[OSSA78] *Unix Programmer's Manual*, Chapter Nroff/troff, Bell Laboratories, 1978.

[PURT86] Purtilo, James M.; Applications of a Software Interconnection System in Mathematical Problem Solving Environments, In *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation*, ACM, July 1986, pp. 16–23.

[QUIN83] Quint, Vincent; An Interactive System for Mathematical Text Processing, *Technology and Science of Informatics*, 2(3), 1983, pp. 169–179.

[SAMM66] Sammet, Jean E.; Survey of Formula Manipulation, *Communications of the ACM*, 9(8), August 1966, pp. 555–569.

[SLAG63] Slagle, J.R.; A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus, *Journal of the ACM*, 10(4), October 1963, pp. 507–520.

[SMIT86] Smith, Carolyn; Soiffer, Neil; MathScribe: A User Interface for Computer Algebra Systems, In *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation*, ACM, July 1986, pp. 7–12.

[SOIF91] Soiffer, Neil Morrell; The Design of a User Interface for Computer Algebra Systems, Doctoral Dissertation Report No. UCB/CSD 91/626, Computer Science Division, University of California at Berkeley, April 1991.

[WORD91] *Microsoft Word for Windows Users Guide*, Microsoft Corporation, One Microsoft Way, Redmond, WA, 1991.

[KNUT81] Knuth, Donald E.; Plass, Michael F.; Breaking Paragraphs into Lines, *Software — Practice and Experience*, 11(11), November 1981, pp. 1119–1184.

[KNUT84] Knuth, Donald E.; *The TEXbook*, Addison-Wesley, 1984.

[KRAS88] Krasnet, G.E.; Pope, S.T.; A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System, *Journal of Object-Oriented Programming*, 1(3), August 1988, pp. 26-49.

[LELE85] A Graphical Interface for Reduce, *ACM SIGSAM Bulletin*, 19(3), August 1985, pp. 17–23.

[LEON86] Leong, B.L.; Iris: Design of a User Interface Program for Symbolic Algebra, In *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation*, ACM, July 1986, pp. 1–6.

[MACS83] *MACSYMA Reference Manual*, Laboratory for Computer Science, MIT, January 1983.

[MART71] Martin, William A; Computer Input/Output of Mathematical Expressions, In *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ACM, March 1971, pp. 78–89.

[MATH89] *MathStation Version 1.0*, MathSoft Inc., One Kendall Square, Cambridge, MA, April 1989.

[MOTI89] *OSF/Motif Programmer's Reference Manual Revision 1.0*, Open Software Foundation, Eleven Cambridge Center, Cambridge, MA, 1989.

[NOLA53] Nolan, J.; Analytical Differentiation on a Digital Computer, Master's Thesis, MIT, May 1953.

[DAVE86] Davenport, James H.; Roth, C.E.; Powermath – A System for the MacIntosh, In *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation*, ACM, July 1986, pp. 13–15.

[EGMO89] van Egmond, S.; Heeman, F.C.; van Vliet, J.C.; INFORM: An Interactive Syntax-Directed Formulae Editor, *Journal of Systems and Software*, 9, 1989, pp. 169–182.

[FATE87] Fateman, Richard J.; TEX Output from MACSYMA-like Systems, *ACM SIGSAM Bulletin*, 21(4), November 1987, pp. 1–5.

[FOST84] Foster, Gregg; DREAMS: Display Representation for Algebraic Manipulation Systems. Technical Report UCB/CSD 84/193, UC Berkeley, April, 1984.

[FRAM89] *Frame Maker Reference Manual, Version 2.0*, Frame Technology Corporation, San Jose, CA, 1989.

[FUMA86] Fumas, George W.; Generalized Fisheye Views, In *Proceedings, CHI '86*, ACM, April 1986, pp. 16–23.

[HART89] Hartson, R.; User-Interface Management Control and Communication, *IEEE Software*, January 1989, pp. 71-77.

[JOHN78] Johnson, S.C.; *Unix Programmer's Manual*, Chapter Yacc: Yet Another Compiler-Compiler, Bell Laboratories, Second Edition, 1978.

[KAHR53] Kahrimanian, H.G.; Analytical Differentiation by a Digital Computer, Master's Thesis, Temple U., May 1953.

Bibliography

[ANTW89] Antweiler, Werner; Strotmann, Andreas; Winkelmann, Volker; A TEX-REDUCE Interface, *ACM SIGSAM Bulletin*, 23(2), April 1989, pp. 26–33.

[AVIT88] *Milo User's Guide*, Paracomp, 123 Townsend St. Suite 310, San Francisco, CA 1988.

[BONA87] Bonadio, Allan; Warren, Erik; *Theorist Reference Manual*, Prescience Corporation, 814 Castro St., San Francisco, CA 1987.

[CHAR91] Char, Bruce W.; Geddes, Keith O.; Gonnet, Gaston H.; Leong, Benton L.; Monagan, Michael B.; Watt, Stephen M.; *Maple V Language Reference Manual*, Springer-Verlag, New York, 1991.

[CLAP63] Clapp, Lewis C.; Kain, Richard Y.; A Computer Aid for Symbolic Mathematics, In *Proceedings, AFIPS Fall Joint Computer Conference* Volume 24, AFIPS Press, 1963, pp. 509–517.

[COWA90] Cowan, W.B.; Wein, M.; State Versus History in User Interfaces, *Human-Computer Interaction – INTERACT '90* pp. 555–560.

The increased availability and capabilities of inexpensive bitmapped display devices, along with their accompanying GUI application software, has made the limitations of traditional CAS interface software very clear. It is hoped that this and related work will help serve as a basis for exploiting the full potential of CAS interfaces that take advantage of contemporary workstation equipment.

The major stumbling block in implementing this sort of functionality is that the interface requires substantial knowledge of the semantics of the expressions being manipulated. For example, although the CAS itself may possess the information about whether or not two subexpressions can be commuted, this fact is not usually available to the interface, which has to make the decision to allow or disallow the manipulation. Another issue that might arise is the question of how to handle multiple solutions arising from a manipulation that isolates a term in an equation. As of this writing, only the Milo [AVIT88] and Theorist [BONA87] interfaces provide basic direct manipulation capabilities, through the virtue of of their built-in algebra engines.

To date, no CAS interface has attempted to provide alternative views of expressions. Both the fish eye and satellite views described in Section 4.1.3 appear to have promise. Some preliminary experiments using Maple library functions to limit the size and content of large expressions have been performed within the Maple user community, but this approach discards information, and does not permit interactive manipulation of the view.

6.3 Conclusions

The design and implementation of an effective user interface for a computer algebra system represents a substantial project, requiring the careful consideration of issues that range over a wide variety of disciplines: human factors, software engineering, data structure and algorithm design, and basic typography figure prominently in the list. It is probable that this diversity of issues has had a negative impact on the evolutionary rate of CAS interface technology, which has lagged behind that of the computer algebra engines themselves.

Enhancements are planned to incorporate both a two-dimensional input facility and interactive editing capabilities in *xiris*; issues related to this project could easily form the basis of another thesis.

The worksheet session model employed in *xiris* also requires further scrutiny. It is not clear that the current implementation is the best way to serve the various different types of users' needs. This model does appear to be headed in a useful direction, namely towards an *active document* paradigm in which the interface can provide a document creation environment with mathematical expressions and graphics that can be dynamically recalculated. However, an effective implementation of this ideal may require a redesign of the interface between the Maple kernel and *xiris*.

On the positive side, *xiris* provides a solid and largely device independent mathematical formatting and rendering engine designed for the efficient handling of the large expressions that can be generated by Maple. Considerable emphasis has been placed on formatting and rendering performance, which will well serve the needs of the planned two-dimensional input and editing facilities. It is probable that this machinery will form a useful basis for output formatting in future Maple user interfaces.

6.2 Future Work: CAS Interfaces in General

One of the more interesting capabilities that CAS interfaces have the potential to provide is the direct manipulation of the mathematical expressions displayed upon the screen. Examples of such manipulations include the moving of a subexpression across the equality sign in an equation, or the rearrangement of the terms within a commutative sum.

generated by computer algebra systems.

- Practical efficiency and implementation issues related to the management of large results, and techniques for maximizing performance while minimizing the use of storage.
- A functional description of the *xiris* interface for Maple, and a presentation of the highlights of its implementation in light of the earlier discussions.

6.1 Evolutionary Directions for *xiris*

Although the current implementation of *xiris* is a useful interface for Maple in its own right, and provides considerable visualization advantages over earlier Maple interfaces, there is still much work to be done before it could be considered to be a complete CAS interface.

One of the major shortcomings in the current *xiris* implementation is that there exists no two-dimensional input facility to permit the user to enter expressions in a visual, "mathematically intuitive" manner; all input must be entered using the traditional Maple command line interface. This is not a serious disadvantage to the experienced Maple user, who might be expected to prefer the entry speed advantage provided by the existing Maple programming language interface, but the omission hinders the usability of Maple for novice and infrequent users.

Another shortcoming, related to the lack of two-dimensional input capabilities, is the lack of support in *xiris* for the interactive editing of formatted mathematical expressions. This includes the selection, cutting and pasting of all or part of a formatted expression. Although it is possible to reformat an expression in character print mode, and then perform character-based selection, this is not an ideal solution.

Chapter 6

Conclusions

This thesis has attempted to present some of the issues inherent in the development of user interfaces for computer algebra systems, with the emphasis on managing the (potentially large) output generated by such applications. Where possible, the practical implementation experiences stemming from the development of a new user interface for Maple were related.

In summary, considerations based on the following topics were presented and discussed:

- General user interface considerations, many of which are generic and can be applied to any type of application.
- Specific user interface considerations for a particular CAS, namely Maple.
- Issues and techniques related to the efficient formatting of mathematical expressions in an interactive environment.
- Issues and techniques based on aiding the user in the visualization of formatted mathematical expressions, with particular attention to the large results

a path representation in the future. A path representation uses the path from the root to the selected node to uniquely identify a subexpression.

majority of the cases are handled by the `fill_childboxes()` routine. In general, the box-type handlers will make recursive calls back to `linebreak()` in order to linebreak any children that are too wide for a line.

Information about whether or not a new line was initialized during each stage of the process is returned explicitly by the linebreaking routines. This fact is used to maintain the "deepest common ancestor" of all boxes for each line. This attribute is not actually required, but it can enhance rendering performance.

Leaf boxes (corresponding to identifiers and numbers) that are too wide for a line require special attention. Because of their atomic nature, they cannot be split across multiple lines using the usual sub-box division technique of other forms. Instead, the line bounding boxes are permitted to intersect the bounding box for leaf nodes; the rendering phase must be able to identify such cases and draw only the appropriate portion of the affected leaf box.

5.5 Sharing

As described in Section 4.3.1, *xiris* will locally share position-independent boxes. The determination of common formatted expressions is determined by hashing the CAS representation of the expression, and verifying the correspondence of the font-sizes.

Selections are represented by a position dependent box pointer, along with the absolute position of the selection in the full virtual line that represents the entire expression. The position information is used to identify which occurrence of a shared subexpression has been selected. This representation has been sufficient for the experiments performed to date. However, it may prove necessary to switch to

```
Boolean linebreak( PDbox box, int ox, int oy ) {
    Boolean newline = FALSE;

    if( box fits on the current line AND
                  box does not require explicit linebreaking )
        add box to the current line box;
    else switch( box->type )
        case LEAF:  newline = linebreak_leaf( box, ox, oy ); break;
        case SUM:   newline = linebreak_sum( box, ox, oy ); break;
        case TABLE: newline = linebreak_table( box, ox, oy ); break;
        ...
        default:    newline = fill_childboxes( box, ox, oy ); break;
    return( newline );
}

Boolean fill_childboxes( PDbox box, int ox, int oy ) {
    Boolean newline = FALSE;

    for( each child of box )
        if( child fits on current line )
            add child to the current line box;
        else if( child fits on a new line )
            initialize new line;
            mark box as root of current line;
            continue;
        else if( linebreak( child, ox + child->x, oy + child->y ) )
            add child to the current line box;
            newline = TRUE;
    return( newline );
}
```

Figure 5.6: Pseudocode Sketch of the *xiris* Linebreaking Algorithm

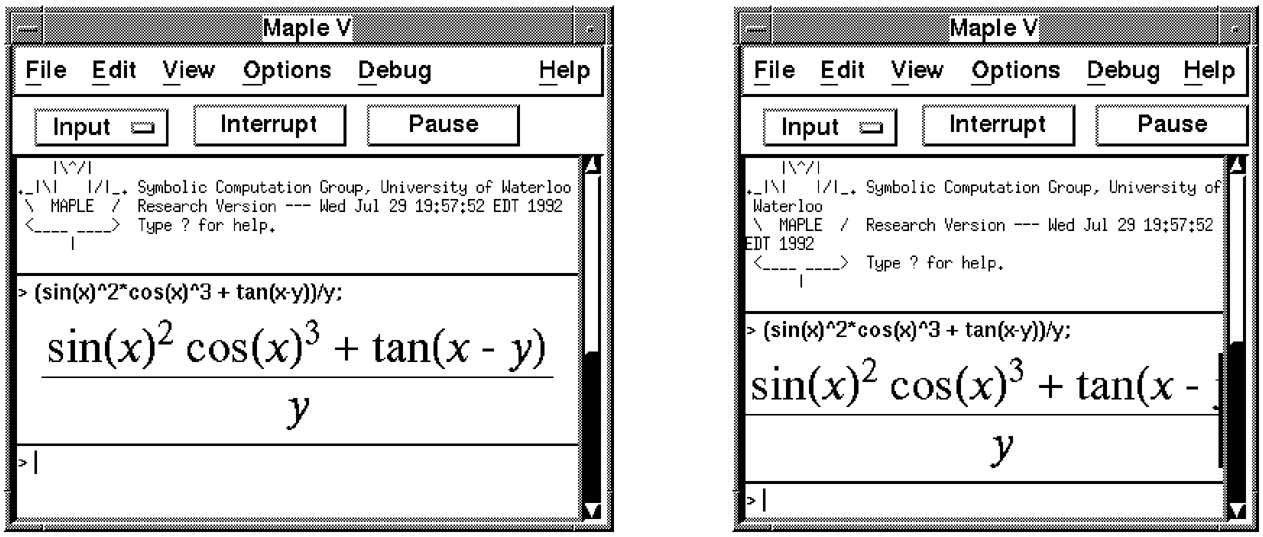
5.4.2 Implementation

The basic goal of the *xiris* linebreaking phase is to construct a set of bounding boxes, called *line boxes*, that overlay the formatted expression. Each of these line boxes represents the extent of the expression that appears on a particular line. The coverage provided by the full set of line boxes is disjoint, while their union encompasses the entire non-empty area of the expression.

As an optimization for the rendering stage, the linebreaking phase also determines the deepest node in the format DAG that is an ancestor of all the boxes or partial boxes that appear on each line. This permits the rendering process to eliminate quickly from consideration those portions of the format DAG that do not appear on the line being drawn.

Note that this mechanism does not enforce any particular method for choosing the line boxes. This facilitates experimentation with different linebreaking policies, as long as they can coexist with the limitations observed earlier. The method used by *xiris* to compute the line boxes is primarily a recursive "first fit" scheme, with some special case improvements. This scheme is inexpensive, and wastes relatively little screen space, making it ideal for interactive use. Figure 5.6 provides a pseudo-code sketch of how it is implemented. The process begins at the root of the expression, and at each stage, the current bounding box is tested to determine whether or not it will fit on the current line. If it will fit, then the current line's bounding box is expanded to encompass the box under consideration. If it will not fit, or if the box is flagged (via its `ALWAYS_LINEBREAK` flag being set) as requiring specific linebreaking attention, then the box is handed to a routine designed to handle that specific box type.

In practice, there are relatively few distinct box-type specific handlers: the



```
    |\^/|
._|\|   |/|_. Symbolic Computation Group, University of Waterloo
 \  MAPLE  /  Research Version --- Wed Jul 29 19:57:52 EDT 1992
 <____ ____>  Type ? for help.
      |
> (sin(x)^2*cos(x)^3 + tan(x-y))/y;
```

$$\frac{\sin(x)^2 \cos(x)^3 + \tan(x - y)}{y}$$

```
>
```

```
    |\^/|
._|\|   |/|_. Symbolic Computation Group, University of
 Waterloo
 \  MAPLE  /  Research Version --- Wed Jul 29 19:57:52
EDT 1992
 <____ ____>  Type ? for help.
      |
> (sin(x)^2*cos(x)^3 + tan(x-y))/y;
```

$$\frac{\sin(x)^2 \cos(x)^3 + \tan(x -}{y}$$

```
>
```

Figure 5.5: Two-dimensional forms in *xiris* linebreaking

linebreaking techniques is certainly possible, but would be expensive in terms of runtime performance. A more general restatement of this limitation is that the form of the output expression cannot be altered using this linebreaking scheme; the *xiris* implementation relies on the layout phase to choose linear forms where required. This limitation is a problem when the user reduces the window width, as some two-dimensional forms may now be too wide to display. There are several ways to handle this situation. The entire expression can be reformatted, or a horizontal scrolling mechanism can be provided. The *xiris* implementation takes the approach that no action need be taken automatically: all that is done is that a visual cue, (in the form of a vertical occlusion bar at the right margin) is displayed to warn the user that a portion of the expression is occluded. The user can then either explicitly request that the expression be reformatted, or simply make the window wider. Figure 5.5 illustrates this situation, and the *xiris* solution.

gested that an expression formatted for one long *virtual line* can be broken into "segments" which do not exceed the window width; each segment is then displayed on a separate line. Soiffer's description of his algorithm was directed at line cutting rather than line breaking, and did not address the issue of hard linebreaks or the problem of wasted space.

5.4.1 Advantages and Disadvantages

This particular linebreaking scheme provides several valuable benefits. Since the linebreaking process is considerably cheaper than the formatting phase, it is possible to recompute linebreak locations automatically when the user changes the window size, without incurring an excessive delay. This maximizes the amount of information visible on the screen, without requiring any extra effort on the part of the user. The linebreaking controls are also useful for performing a quick determination of what portion of an expression is currently visible in the window; this enhances drawing performance by permitting the rendering phase to draw only those lines that are visible. Finally, this mechanism permits the sharing of subexpressions, even when breakpoints occur at different locations in distinct instantiations, and requires very little storage overhead.

There are drawbacks to this linebreaking algorithm, too. In particular, it complicates the rendering process somewhat, as the rendering code must integrate the information contained in the linebreaking controls with the format DAG structure. Added complexity also appears in the implementation of selection of the displayed expressions with a mouse.

A further limitation of the current implementation is that it only supports surface-level linebreaking (see Section 4.2.3); extending it to support deep-level

$$
\begin{array}{l}
T := \mathrm{table}([\\
\quad \alpha = \dfrac{\sin(x)}{x^2} \\
\quad \beta = \cos(x)^2 \\
\quad \gamma = \pi \tan(x) \\
\quad])
\end{array}
$$

Figure 5.4: Displayed format of a Maple Table structure in *xiris*

It is occasionally required that a “hard linebreak” be inserted in the displayed form of the expression. A hard linebreak forces a new line to begin when it is encountered, regardless of the amount of free space remaining on the current line. Hard linebreaks are not generally required in traditional mathematical notation, but are useful in certain special situations. For example, the *xiris* formatter uses hard linebreaks to place each entry of a Maple table on a separate line (see Figure 5.4), regardless of the current line width. Hard linebreaks are not stored as explicit entities in the formatting DAG. Instead, a flag is set in the appropriate box to indicate that the box must be considered explicitly by the linebreaking code, regardless of whether or not the box's dimensions indicate that it could fit on the current line. This `ALWAYS_LINEBREAK` flag is propagated up the format DAG to the root, where it can be subsequently examined by the linebreaking phase.

5.4 Linebreaking

The *xiris* linebreaking algorithm is an extensively modified version of an approach suggested by Soiffer in his doctoral dissertation [SOIF91, pages 103–105]. He sug-

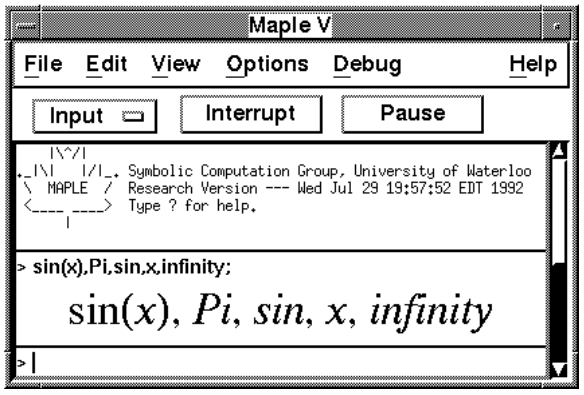
Optional Form	**Description**
∂ vs. d	Notation for differentials.
$\exp(x)$ vs. e^x	Form of the exponential function.
i vs. j	The representation of $\sqrt{-1}$.
Subscript Brackets	Controls the bracketing of subscripts.

Table 5.1: Optional Formatting Forms Provided in *xiris*

The list of the alternative display forms is an ad-hoc one that has evolved over time, and currently consists of the choices illustrated in Table 5.1. Some items, like the choice of ∂ vs. d for the differential operator, are designed to permit customizations based on the type of user (for example, first year calculus students are usually unfamiliar with partial derivatives). Others, like the representation for $\sqrt{-1}$, are aimed at handling the notational differences encountered across various mathematical disciplines..

5.3.2 Linebreaking Requirements

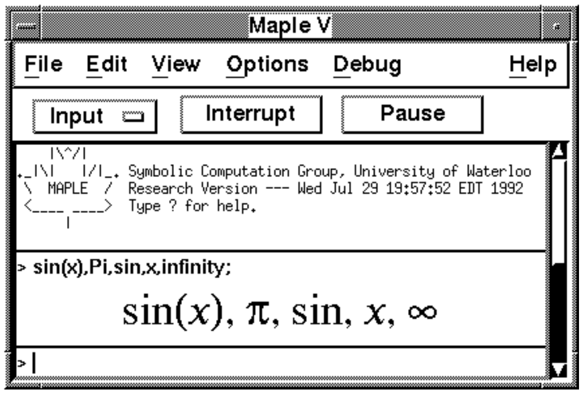
Recognizing those locations in the expression where the natural two-dimensional forms cannot be used (see the discussion on Linear Forms in Section 4.2.3) is performed in the layout phase. This recognization requires knowledge of the output device width; at the beginning of the layout pass, the maximum width for a box that cannot be linebroken is computed using this information. Two-dimensional forms that would exceed this width are formatted in their linear format; this ensures that the linebreaking phase will not encounter boxes that cannot be broken across multiple lines. Thus, the result of the layout phase is an expression formatted as if it were one long virtual line, but that includes no two-dimensional forms that cannot be linebroken to the current output device width.



```
    |\^/|
._|\|   |/|_. Symbolic Computation Group, University of Waterloo
 \  MAPLE  /  Research Version --- Wed Jul 29 19:57:52 EDT 1992
 <____ ____>  Type ? for help.
      |
> sin(x),Pi,sin,x,infinity;
```

$$\sin(x),\ \mathit{Pi},\ \mathit{sin},\ x,\ \mathit{infinity}$$

```
>
```



```
    |\^/|
._|\|   |/|_. Symbolic Computation Group, University of Waterloo
 \  MAPLE  /  Research Version --- Wed Jul 29 19:57:52 EDT 1992
 <____ ____>  Type ? for help.
      |
> sin(x),Pi,sin,x,infinity;
```

$$\sin(x),\ \pi,\ \sin,\ x,\ \infty$$

```
>
```

Figure 5.3: Effects of the Name Translations Mechanism

appears.[1] The second translation record causes `Pi` to be printed with the string `p` using the symbol font, which produces the π glyph.

Currently, all translations are specified using the X11 Resource Manager and are installed when *xiris* is initialized; there is not yet a mechanism to permit users to augment the name translations from within a Maple session, although such a facility is planned.

Alternative Forms

A pull-down menu permits the user to choose between a limited set of "alternative forms". This mechanism permits the user to specify several special format customizations, according to their taste or level of expertise. As with the name translation facility, these customizations affect only the display of the appropriate forms in the high-resolution printing mode; the character printing formats are unaffected.

[1]By default, names appearing in the context of a function call are displayed using a non-italic font, while all other occurrences are rendered in italics.

complicated problem which might be solved by providing customizable formatting is the provision of definable notation for a user-defined operator.

The discussion on formatting directives presented several techniques for providing hooks for fully customizable formatting. While *xiris* does not yet support fully user-programmable formatting, it does provide two mechanisms for handling basic customization tasks.

Name Translations

Name translations permit the user to specify a simple name substitution for Maple identifiers. This mechanism is used to map names like `Pi` to their Greek letter representations, and to display built-in primitive function names in a distinctive typeface regardless of context.

The necessity for a name translation mechanism arises because of the limited mathematical knowledge available to *xiris*. In particular, no semantics are attached to the Maple identifiers that *xiris* sees: for example, it does not know that the sine function is known automatically to Maple, and accordingly, it cannot treat the Maple name `sin` any differently than the name `x`. Figure 5.3 illustrates the effect that the name translation mechanism can provide.

Name translations can be specified dynamically to *xiris* via a simple text string, which specifies the Maple form of the identifier, the translated form of the identifier, and the font to be used for the translated form. The translation directives `sin normal` and `Pi symbol p` implement the translations observed in Figure 5.3. The first one does not specify a translation string for `sin`, so the original name is used. The specification of “normal” as the font forces *xiris* to print all occurrences of the name `sin` in the non-italic font, regardless of the context in which the name

5.3 Formatting

The formatting (or layout) phase employed in *xiris* is a straightforward procedure-based one. All of the formatting procedures are encoded in C, the *xiris* implementation language. As mentioned in the discussion on sharing (in Section 4.3.1), *xiris* shares position-independent boxes, and accordingly, the box structures created during the formatting pass are heterogeneous. In order to provide further savings on memory requirements, even the position-independent boxes are not uniform, but vary in size according to their type. Although all such boxes contain a common set of fields reflecting the box type, its width and height, and some flags, leaf boxes also have string and font identifier fields, while internal boxes have fields to store pointers to their children. Figure 4.5 provides a conceptual illustration of the output generated by the layout phase, although the heterogeneous nature of the position independent boxes is not shown.

The linebreaking algorithm used in *xiris* requires that the child boxes of the internal boxes be stored in an order based on their left to right positions. This permits the linebreaking mechanism to make use of the spatial coherence properties of a box's children to improve runtime efficiency.

5.3.1 Simple Format Customizations

As was noted in Chapter 3, regardless of the number of formats made available by the output formatting procedures, it is inevitable that there will be mathematical forms that are not displayed in a manner that is satisfactory to the user. Some of these cases arise from simple variations in notation between various disciplines: the use of both i and j to represent $\sqrt{-1}$ is one example. An example of a slightly more along with information about the current dimensions of the output device, are used by the linebreaking phase, which constructs a set of linebreaking controls (without altering the input). Finally, both the format DAG and the linebreaking control information is used by the rendering phase to display the lines of the expression on the output device.

This architecture for formatting has several advantages over other implementations. In particular, the separation of the layout and linebreaking phases simplifies the implementation and maintenance of the former, while facilitating experimentation with completely different techniques in the latter. The linebreaking algorithm currently used in *xiris* is basically a simple "first-fit" scheme; however, it would be possible to replace it with a more sophisticated mechanism if desired. Furthermore, it is the separate linebreaking pass that permits *xiris* to recompute breakpoints in outputs automatically when the user changes the size of the application's main window.

Figure 5.2 also visually demonstrates the degree of device independence that the *xiris* formatting mechanism has achieved. Both the layout and linebreaking phases require only font metric information and output device dimensions to produce their outputs; furthermore, although the rendering phase is inherently device dependent, it requires only a few primitive operations be implemented for a particular device. The current *xiris* implementation makes use of this device-independent architecture to image mathematical expressions in both X11 windows and upon Postscript output devices.

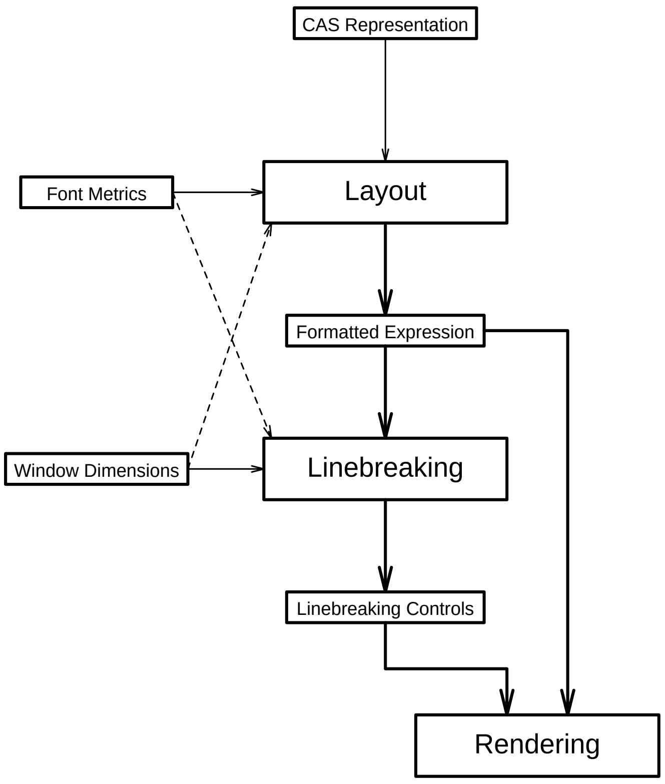
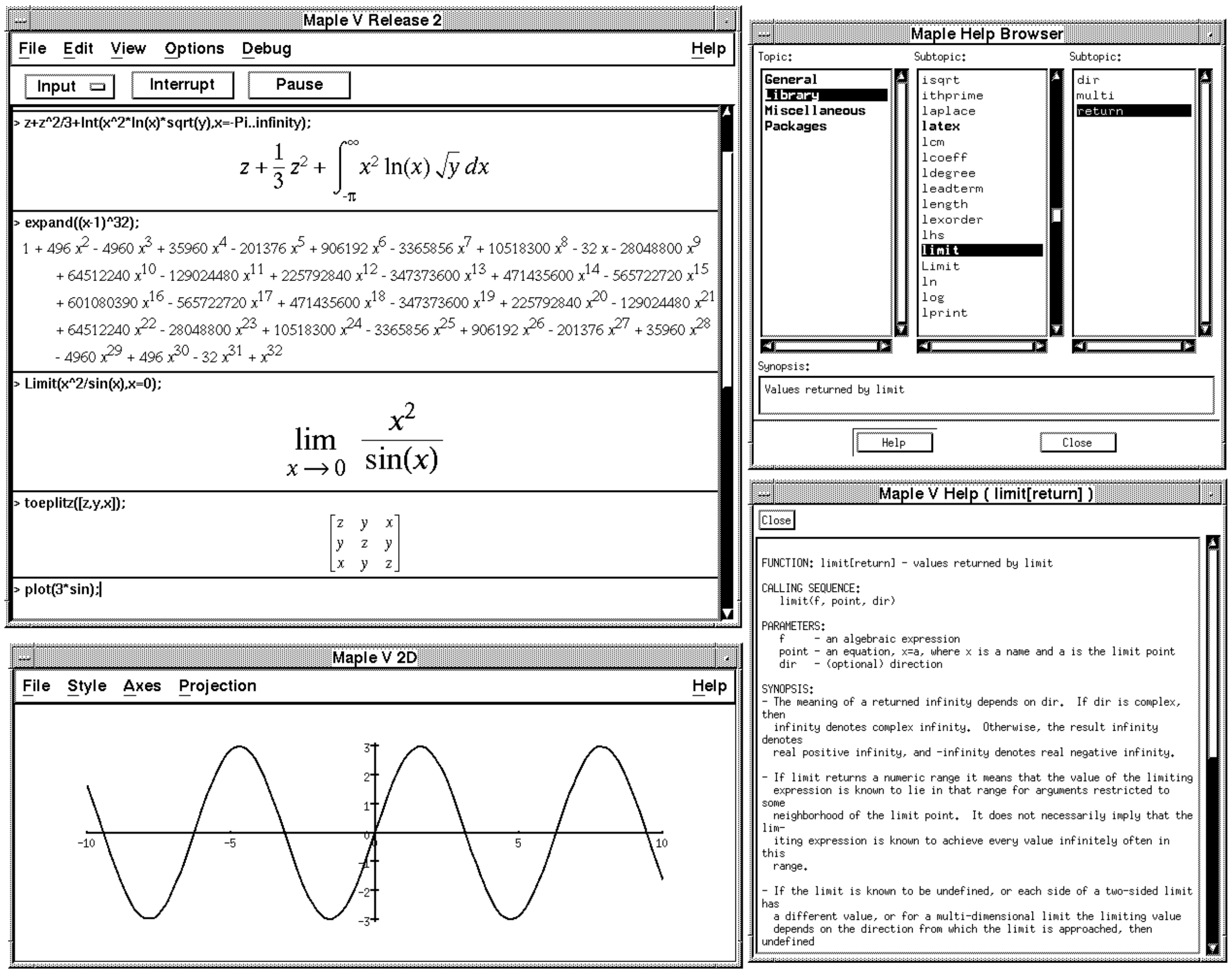


Figure 5.2: Information Flow in the *xiris* formatting process

The mechanism through which expressions are formatted and rendered on the display is a three part process. These three parts are the layout phase, the line-breaking phase, and the rendering phase; the first and last phases are described in Section 3.1, while the linebreaking phase chooses breakpoints in the fully formatted expression. Figure 5.2 shows the relationship between these phases, and the inputs and outputs produced at each stage.

The layout phase takes as input the CAS form of the expression to be formatted, and the metrics corresponding to the fonts to be used. It produces a hierarchical box structure DAG representing the fully formatted expression, without linebreaks, formatted as if for an infinitely large display. This box structure,



```
> z+z^2/3+Int(x^2*ln(x)*sqrt(y),x=-Pi..infinity);
```

$$z + \frac{1}{3} z^2 + \int_{-\pi}^{\infty} x^2 \ln(x) \sqrt{y}\, dx$$

```
> expand((x-1)^32);
```

$$1 + 496 x^2 - 4960 x^3 + 35960 x^4 - 201376 x^5 + 906192 x^6 - 3365856 x^7 + 10518300 x^8 - 32 x - 28048800 x^9$$
$$+ 64512240 x^{10} - 129024480 x^{11} + 225792840 x^{12} - 347373600 x^{13} + 471435600 x^{14} - 565722720 x^{15}$$
$$+ 601080390 x^{16} - 565722720 x^{17} + 471435600 x^{18} - 347373600 x^{19} + 225792840 x^{20} - 129024480 x^{21}$$
$$+ 64512240 x^{22} - 28048800 x^{23} + 10518300 x^{24} - 3365856 x^{25} + 906192 x^{26} - 201376 x^{27} + 35960 x^{28}$$
$$- 4960 x^{29} + 496 x^{30} - 32 x^{31} + x^{32}$$

```
> Limit(x^2/sin(x),x=0);
```

$$\lim_{x \to 0} \frac{x^2}{\sin(x)}$$

```
> toeplitz([z,y,x]);
```

$$\begin{bmatrix} z & y & x \\ y & z & y \\ x & y & z \end{bmatrix}$$

```
> plot(3*sin);
```

Topic:	Subtopic:	Subtopic:
General	isqrt	dir
Library	ithprime	multi
Miscellaneous	laplace	return
Packages	latex	
	lcm	
	lcoeff	
	ldegree	
	leadterm	
	length	
	lexorder	
	lhs	
	limit	
	Limit	
	ln	
	log	
	lprint	

Synopsis:
Values returned by limit

```
FUNCTION: limit[return] - values returned by limit

CALLING SEQUENCE:
   limit(f, point, dir)

PARAMETERS:
   f     - an algebraic expression
   point - an equation, x=a, where x is a name and a is the limit point
   dir   - (optional) direction

SYNOPSIS:
- The meaning of a returned infinity depends on dir.  If dir is complex,
then
  infinity denotes complex infinity.  Otherwise, the result infinity
denotes
  real positive infinity, and -infinity denotes real negative infinity.

- If limit returns a numeric range it means that the value of the limiting
  expression is known to lie in that range for arguments restricted to
some
  neighborhood of the limit point.  It does not necessarily imply that the
lim-
  iting expression is known to achieve every value infinitely often in
this
  range.

- If the limit is known to be undefined, or each side of a two-sided limit
has
  a different value, or for a multi-dimensional limit the limiting value
  depends on the direction from which the limit is approached, then
undefined
```

Figure 5.1: A sample *xiris* session

to the current window width using a linebreaking mechanism that is detailed below. Results are linebroken automatically whenever the window width is altered.

A hierarchical help browser subsystem permits the user to examine the extensive help information available on the Maple library functions, while pull-down menus (implemented with OSF's Motif Toolkit [MOTI89]) are used to control formatting and display options. The entire session can be formatted for rendering on Postscript devices.

Figure 5.1 illustrates some of these *xiris* features, and shows a plot window, the hierarchical help browser and a help page, in addition to the main window.

5.2 General Architecture

The only *xiris* implementation at present exists as an X11R4 client, and accordingly can be compiled and run on any Unix system with X11 and Motif libraries installed. The Motif Toolkit is used to provide the standard interface components such as the pull down menus, status information, and the dialogues. All drawing in the main window is performed via Xlib calls.

Currently, *xiris* sits on top of the existing Maple Iris implementation. This is not an ideal arrangement, but practical time constraints precluded a proper bottom-up rewrite of the current Maple Iris. This layered approach is useful in terms of practical maintenance issues, since it permits the existing Iris code to continue to be used for other platforms with very few changes; at the same time, however, this architecture has hampered the implementation of the desired functionality. The planned modularization of the Maple Iris, as discussed on page 20 will facilitate future *xiris* development.

- provide effective visualization of CAS results
- not impair performance to any appreciable extent
- be portable as possible
- be extensible as possible
- have only minimal impact on the existing Maple system
- adhere as much as possible to the principles of interface design discussed in Chapter 2.

Due to time constraints, no attempt has yet been made to improve the Maple input facilities, or to provide any sort of two-dimensional editing mechanism. The design of the current system has attempted to facilitate the addition of such features in the future.

What the *xiris* system does provide is a primitive worksheet model, in which the user's work session consists of a series of horizontal strips or *regions*. These regions have a type attribute, insofar as each region can contain either Maple input, the results of Maple output, or inert textual comments. The overall interface effect is very similar to the dialogue style of other Maple interfaces: the user types Maple statements in the input regions, and the Maple system responds with results which are formatted using the techniques described in Chapter 3 and displayed in the output regions.

A vertical scrollbar on the side of the session window permits the user to scroll back to the beginning of the session and review results, or alter the inputs and re-execute them. Horizontal scrolling is not provided: large results are linebroken

Chapter 5

The XIRIS Implementation

This chapter provides a functional description of the current incarnation of *xiris* and its capabilities, and then discusses some of the implementation details in light of the considerations of the previous chapters. Readers without particular interest in the implementation details may wish to pass over this discussion, although the linebreaking mechanism used by *xiris* and described here is novel and has some valuable properties.

5.1 Functional Description

The current incarnation of the *xiris* interface is but one step on the way to a truly effective interface for Maple. Rather than attempting to provide an inflexible and poorly designed interface with an excess of features, the design and implementation efforts were directed at the construction of a solid base upon which future development can occur in a systematic and controlled manner. Accordingly, the existing *xiris* implementation was designed with the following goals in mind:

valuable when the rendering process cannot keep up with the user's scrolling requests. Avoiding scrolling altogether when possible can also improve performance: rather than automatically displaying and scrolling through all lines of large multi-line expressions, *xiris* displays only the final screen page of a result, and relies upon the user to manually scroll back to view the earlier lines.

the structures being allocated to minimize the impact of the hardware restrictions.

4.4 Rendering Performance

Formatting and displaying large expressions can potentially require a substantial amount of time. While the time required to format and display an expression can often be hidden in the CAS's computation time, interface specific operations that require the reformatting or redisplay of an expression, such as scrolling or redrawing the main window, must occur quickly in order to convey a feeling of responsiveness to the user.

Several techniques can help speed up the process of rendering large expressions on the display. Probably the most obvious one, but also the most effective, is to avoid drawing those boxes which are not visible. The method of determining just which boxes are not visible depends on how large expressions are displayed. Interfaces which use horizontal scrolling can normally discard boxes that fall outside of the current window very quickly, especially if the boxes are ordered from left to right; similar techniques can be used on a per-line basis in systems which use line cutting or breaking.

Speedup techniques that depend upon the output device are also common. Avoiding frequent font changes, and drawing characters as strings rather than individually works well when drawing to an X11 window; another useful X11 optimization is to use the faster opaque character drawing operations (which overwrite their background) rather than their transparent background equivalents.

Jump scrolling can be useful for improving interactive scrolling performance. Jump scrolling "compresses" multiple scrolling directives into fewer ones, and is

4.3.2 Session Memory

Reducing the storage requirements of the display representation may be only part of the memory management solution. A CAS interface needs to be able to create and destroy display representations for what could be very large expressions; if interactive editing of these expressions is supported, then the memory requirements of any particular expression may need to grow or shrink by an arbitrary amount.

The current implementation of *xiris* has not addressed this issue to any great extent, since neither global sharing nor editing of displayed expressions is supported yet. The lack of these two features provides a very convenient simplification for memory management: all storage used to represent any particular expression has the same persistence. This implies that all the required memory can be allocated when the expression is formatted; when the expression is destroyed, all memory can be released at once.

The existing Maple Iris already embodies a memory allocator which supports the fast allocation and deallocation of memory based on its persistence, and the *xiris* implementation makes extensive use of this functionality. The use of this particular memory management system has pointed out two important considerations, although neither is a CAS interface issue *per se*. First, the granularity of the allocations performed by the memory management system is worthy of attention. If the chunk size is too small, performance suffers as repeated memory allocation requests are made of the operating system. On the other hand, if the chunk size is too large, memory is wasted. The second issue concerns address alignment of the structures being allocated: the underlying hardware architecture may impose alignment restrictions on particular data types. Nothing can really be done about this limitation, although it can be worthwhile to consider the sizes and arrangement of dimensional notation, as described earlier in the discussion of linebreaking. As in the case of fontsizes, the format type of the box may need to be maintained. This prevents the undesirable sharing of boxes when different formatting is required for the same subexpression that appears in different contexts.

Parentheses can pose a special problem if they are stored as an inherent part of an expression. This is because a subexpression may require parenthesization in some contexts, and not in others, as in the case of the expression

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

This problem can be alleviated by storing the parentheses with a distinct parent node rather than with the expression itself. Although this increases the depth of the DAG, it permits full box sharing between the parenthesized and non-parenthesized versions of the expression.

Finally, the mechanism for deciding when boxes can be shared is worthy of some discussion. If the CAS form of the expression is a DAG, as it is in Maple, then the form of the DAG itself, coupled with the additional attributes mentioned above, can be used in a hashing scheme to determine commonality of formatting boxes. This is the method used in *xiris*. However, the CAS DAG itself may have some undesirable sharing within it, at least from the formatting standpoint. An example of this arises in the Maple form for procedures: parameters and local variables are represented internally as array references with respect to their parent procedures, and as such can be shared between different procedures. Although the local variable or parameter name so referenced will have different names in different procedures, naive sharing of the formatted forms of these structures will result in formal parameters from one procedure appearing in what appears to be a totally unrelated one.

subexpressions tend to be very prevalent between consecutive results. Soiffer's results show that global sharing can provide an additional 70%–75% savings over local sharing.

The principal problem with global sharing within an interactive environment is one of memory management. Deletion of an expression requires some form of analysis to determine what portions of the global DAG can be dispensed with, while a local sharing scheme can safely dispose of the entire structure without additional consideration.

The current *xiris* implementation does not utilize global sharing.

Other Sharing Considerations

Several other points are worthy of consideration with respect to the implementation of a box sharing mechanism. If multiple fonts are available, then a fontsize attribute must be considered when deciding whether or not two subexpressions are identical for the purposes of formatting; thus, no sharing is possible in the expressions such as

$$\sin(x + y^2)^{x+y^2}$$

whenever the exponents are to be formatted in a smaller font than their respective bases. Although it is conceivable that font size information be moved from the position independent boxes to the position dependent ones to permit sharing in this cas, such a design requires that the bounding box dimensions be scalable according to font size. The nature of the fonts used in the X11 *xiris* makes meeting this requirement impossible.

A similar problem may arise if the formatter supports context-dependent display forms. An example of such a form is the linear representation of an inherently two-

There are several variations on the theme of sharing formatting information; these variants boil down to the question of just how much information should be shared. Soiffer discusses these issues in some detail in [SOIF91]; sketches are presented here.

Type of the Shared Boxes

The decision of what exactly should be shared is a fundamental question when implementing a sharing scheme. Figure 4.5 demonstrates a model in which the position independent boxes are shared. However, it is equally valid to share position dependent boxes instead, or even both types simultaneously. Note that even in a homogeneous sharing environment (in which only boxes of one type are shared), some implicit sharing of the other box type occurs automatically whenever non-trivial subexpressions are present. Soiffer reports that sharing both box types (total sharing) results in the greatest savings in memory usage, although thanks to implicit sharing of both box types, the savings resulting from sharing only one box type are not far behind.

The current *xiris* implementation shares position-independent boxes.

Local or Global Sharing

The scope of the sharing must also be addressed. Global sharing permits boxes to be shared among all the expressions contained within a session, while local sharing requires common formatting subexpressions to be contained within the same expression. Thus, global sharing results in one large DAG spanning the entire session, while local sharing provides a DAG forest, with one DAG per expression. The nature of CAS sessions make global sharing very attractive, since common

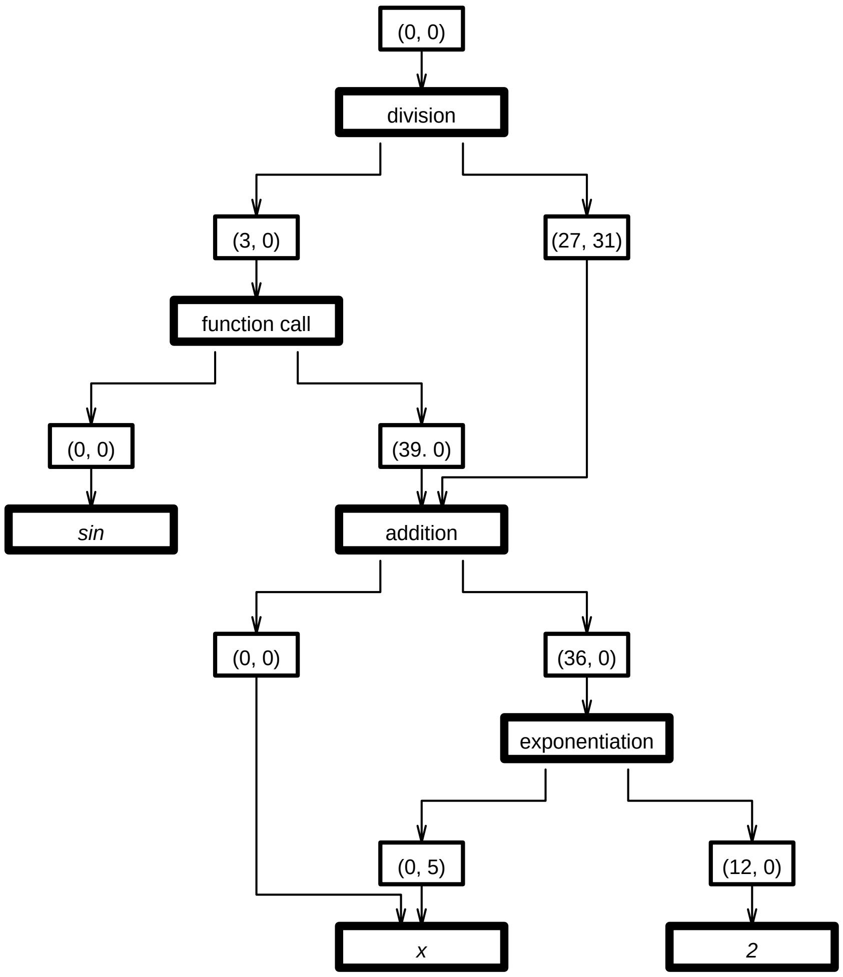


Figure 4.5: A formatting DAG for $\frac{\sin(x+x^2)}{x+x^2}$

4.3.1 Sharing

One of the most effective methods for controlling the growth of memory requirements is to *share* common formatting information. In a sharing scheme, common subexpressions need only be formatted once, and the hierarchical box structures representing the formatted expression become nodes in a directed acyclic graph, or DAG. This technique will frequently provide a substantial payback in storage savings, since common subexpressions tend to appear with considerable frequency in mathematical expressions of appreciable size.

To date, only Soiffer's MathScribe and *xiris* implement box sharing, although Maple's TTY-based Iris takes advantage of common subexpressions to improve formatting performance. Since the latter ultimately produces strings of ASCII characters, it does not maintain a formatted representation DAG as the other systems do.

In order to implement sharing, one important modification is required to the formatting methods described in Chapter 3: positioning information must be separated from the remainder of the bounding box, and maintained in a distinct structure. This results in two basic data types, which Soiffer calls Boxes and PBoxes. The former correspond to the bounding box structure outlined on page 32 without the location information; these are also known as *position-independent* boxes. PBoxes, or *position-dependent* boxes, need only consist of the location of, and a pointer to a position-independent box. An illustration of a DAG structure that might result from this separation of information is shown in Figure 4.5.

Note that Figure 4.5 utilizes a *parent-relative* positioning representation for the location of the position dependent boxes; the (x, y) pair positions the child box's origin relative to the parent box.

the results of structured linebreaking.

4.3 Memory Management

The most significant disadvantage of using high resolution formatting techniques for the display of mathematics in CAS interfaces is the increased amount of storage required. This section describes why the additional storage is required, and some techniques for handling the problems associated with this aspect of CAS interface design.

The additional storage required in a true graphical user interface (GUI) environment over a traditional TTY-based one stems from two factors. First, most existing CAS interfaces use strings of fixed-width ASCII characters to represent their output; these strings of text require relatively little storage space compared to the hierarchical box structures described in Chapter 3. The maintenance of these structures is frequently necessary even after the initial rendering of the expression has been performed, since most windowing systems require client programs to be able to reconstruct the contents of their windows upon demand. Furthermore, if the interface makes the entire history of the session available through some form of scrolling mechanism, then it becomes necessary to maintain the structures for arbitrary periods of time.

Second, TTY-based interfaces do not generally support the selection, or "picking" of displayed expressions with pointing devices (such as a mouse); thus, these interfaces do not need to store the information required to resolve such references. To a large extent, this information is already available in the hierarchical box structures, so maintaining picking information does not require any further storage beyond that needed for rendering the expression.

```
          2
    t * (x  - x + 1)

+

    32 * sin(x - 1)
```

(a) One level of linebreaking

```
        t
    *
          2
        (x  - x + 1)
+

        32
    *
        sin(x - 1)
```

(b) Two levels of linebreaking

Figure 4.4: Results of Structural Linebreaking

the risk of appearing inconsistent, even within the same expression.

Structural linebreaking illustrates some of the problems that can arise from making extreme use of indentation. Structured linebreaking splits expressions that are too wide for a line by placing each subexpression on a separate line, at an increased level of indentation. The process is then applied recursively to the subexpressions. Although this linebreaking scheme tends to show the structure of an expression very well — indeed, Soiffer points out that the visual display resembles a tree — it wastes considerable amount of vertical display space, and is problematic once the indentation level exceeds the width of the display. Figure 4.4 shows an example of during the formatting process, which permits subexpressions to be formatted in the required form on the first attempt.

Another technique that can avoid the reformatting of subexpressions is a re-use strategy. In such a scheme, only the notation associated with the two-dimensional form need be changed, while the subexpressions themselves maintain their original format forms. As an example, consider the conversion necessary to switch between the forms in Figure 4.3 — only the relative placement of the numerator and denominator boxes need change, along with the nature of the separating notation. This re-use technique is not applicable in the general case, however, since it relies upon the system's ability to linebreak linear forms without reformatting them.

Indentation

The technique of varying the amount of indentation for broken lines can be a useful visualization cue. Overuse of indentation can lead to problems, however, and can actually obscure the form of the expression being linebroken.

Several choices exist for selecting indentation levels for even relatively simple expressions. Soiffer observes that indentation is not treated consistently in mathematical literature, although he does suggest some rules of thumb [SOIF91]. In particular, the use of at least one level of indentation for a broken line is very important, as it draws the reader's attention to the fact that the expression is continued over more than one line. Indentating an additional level whenever a subexpression is broken is also a common choice, although this can quickly lead to excessive indentation in complicated expressions, resulting in wasted display space and reducing the available width for the expressions themselves. Mechanisms that avoid this problem by varying further indentation based on the current level run breakpoints. This implies that an attractive choice early in the expression has the potential to result in poor ones later.

The linebreaking mechanism used in TeX [KNUT81] is designed to address this problem of choosing uniformly good breakpoints over entire expressions. Antweiler, Strotmann and Winkelmann use this scheme as the basis for linebreaking in their TeX-REDUCE Interface [ANTW89], which typesets REDUCE formulas in TeX. This method makes multiple passes over the full expression, examining all potential breakpoints and assigning a *demerit* to each. The sequence of linebreaks that minimizes the total demerits over the complete expression is deemed to be the best choice of breaks.

It is apparent that this linebreaking scheme is considerably more complex than a simple, "first-fit" type of method. Antweiler *et al* reports in [ANTW89] that their linebreaking implementation requires five times the amount of CPU time used to simply format the result, even with their simplifications to the full TeX linebreaking model, although they observe that that this time increase is linear in the length of the expression being broken. Since their implementation is not aimed at handling the display of expressions interactively, the time penalty for better linebreaking is not a particularly important problem. In an interactive CAS interface, optimal linebreaks are not necessary, especially at the cost of increased formatting time.

Further efficiency considerations arise from the need to convert two-dimensional expressions to their linear forms, if it is deemed that a breakpoint must fall within them. Since this conversion frequently requires the reformatting of the affected node's subexpressions, it is conceivable (although rare) that exponential time (in the number of subexpressions) could be required to complete the formatting and linebreaking of the particular node. This growth in formatting time can be controlled to a large extent by making linebreaking decisions between subexpressions vantages of traditional mathematical notation are lost. As an example, consider a definite integral in which one of the bounds of integration is extraordinarily wide, which forces the use of the functional notation used in the input process. In Maple, this appears as

$$int(x^2, x = 0..\text{very wide upper limit})$$

The cognitive effort required to recognize the nature of this expression, especially when it is embedded within a larger expression, is quite high. However, the necessity of resorting to the linear form can be avoided here if the subexpression causing the problem is labelled, as described earlier. In Maple, the resulting form would be similar to

$$\int_0^{\%1} x^2 dx$$

$$\%1 := \text{very wide upper limit}$$

which facilitates recognition of the nature of the integral expression, although at the cost of requiring the user to direct their attention elsewhere to determine the true nature of the integral's upper limit.

Choosing Breakpoints

The task of selecting the breakpoint locations within the expression tends to be difficult from two perspectives. The first of these is a matter of effective visualization and aesthetics, while the second is primarily one of execution efficiency.

From the visualization perspective, ideal breakpoints should be chosen such that vertical space is not wasted excessively, and mathematical content is not overly obscured. Maintaining these criteria over the entire expression can be difficult, since the location of any particular breakpoint influences the choice of subsequent

$$\frac{\text{very wide numerator}}{\text{very wide denominator}}$$

$$\text{(very wide numerator)/(very wide denominator)}$$

Figure 4.3: Two-dimensional and linear forms of a quotient

within such forms; Soiffer calls this approach a “surface-level” linebreaking scheme. Pure surface-level linebreaking methods must handle these wide, unbreakable forms using another mechanism, such as horizontal scrolling.

In order to linebreak expressions at arbitrary places (a “deep-level” linebreaking approach), it must be possible to choose breakpoints within two-dimensional forms, which in turn requires that they possess an equivalent linear form. As an example, consider the case of quotients. The preferred representation generally shows the numerator vertically aligned and centered over the denominator, as in Figure 4.3, but if the width of either the numerator or the denominator exceeds that of the display, then it is necessary to resort to the obvious linear form, in which both the numerator and the denominator can be recursively broken over lines as required.

Generally speaking, it is probably best from a consistency point of view if the linear forms correspond to the notation used to enter the expression; thus, in Maple, the linear exponentiation form would use a caret between the base and the exponent.[1]

The obvious disadvantage of using the input form is that the visualization ad-

[1]It’s worth pointing out that exponentials have a middle ground between requiring a completely linear format and their ideal two-dimensional representation: as long as a breakpoint is not chosen between the base and the exponent, it is frequently possible to use the traditional superscript notation and dispense with the caret entirely.

lines. On the other hand, trimming the unused space from each piece can make visualization more difficult – consider the case of attempting to distinguish the trailing portion of a radical from that of a denominator. Covering the top half of Figure 4.2 and attempting to derive useful information from the bottom half serves to illustrate the visualization problems that can arise from line cutting.

4.2.3 Linebreaking

Linebreaking is the most natural method for handling large expressions, insofar as it is the technique that is most often used in traditional pencil and paper mathematics. As in line cutting, linebreaking breaks the full expression into pieces which fit on the screen (or page), and places each of the pieces on a new line. However, while linecutting procedures generally cut the expression into pieces in a naive manner – without regard to the semantics of where the cut points fall – linebreaking techniques attempt to locate "breakpoints" appropriate to the content of the expression. Linebreaking has historically been provided by many CAS systems as a solution for large expressions.

The following discussion on linebreaking is divided into three main sections. They are concerned with the handling of two-dimensional forms, how to choose effective breakpoints, and the issue of managing indentation of broken lines.

Linear Forms

Expressions which are represented in an inherently two-dimensional form, such as quotients and exponents, pose particular problems for linebreaking. It is clear that choosing a breakpoint within such a form amounts to nothing more than a particularly naive line cut. One solution to this problem is to simply disallow linebreaks

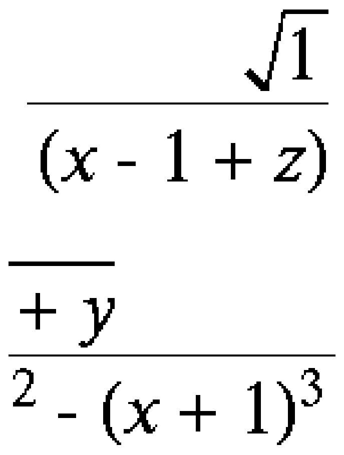


Figure 4.2: An example of a poor line cut location

permits more of the expression to be viewed at once.

Line cutting retains the natural two-dimensional mathematical form of the expression, and the advantages of the simplified layout pass. Furthermore, it is amenable for use with hardcopy output devices.

The principal difficulty with line cutting is choosing cut points in such a way that visualization is not impaired. Arbitrary cut points have the potential to split characters across lines, or worse, separate elements which must be juxtaposed to preserve semantic content; Figure 4.2 shows an example of a potentially misleading line cut. Soiffer suggests that a "glue" based scheme similar to that employed by TEX in conjunction with operator precedence information could choose effective line cuts. No known system has attempted to implement such a mechanism, however.

Another problem with line cutting is that it has the potential to waste vertical screen space on certain forms of expressions. If each piece of an expression retains the full height of the overall expression, but only one or two of the pieces actually use that full height, then excess whitespace is produced on all of the remaining portions of the expression not visible are clipped by the boundaries of the viewing window.

Two disadvantages of scrolling are apparent from a visualization standpoint. First, scrolling does not make effective use of display screen space on a per expression basis. If an expression is only a few screen widths wide, an alternative scheme (such as linebreaking) can make the entire result visible at once and alleviate the need for scrolling entirely. Furthermore, the comprehensibility of larger expressions tends to benefit from the additional context that is made visible when linebreaking is used. The second major disadvantage of scrolling is that it does not lend itself to handling hardcopy output.

The overall effectiveness of horizontal scrolling depends on the scale of the problem that the underlying CAS is designed to handle. The DERIVE system, which is intended for modest computations, provides a horizontal scrolling mechanism that is generally sufficient for the size of the results it is capable of producing. MathScribe also utilizes horizontal scrolling, but as it is designed to interface with algebra systems that are capable of producing large results, it provides a less satisfactory solution than other techniques.

4.2.2 Line Cutting

Line cutting preserves some of the simplicity of scrolling and makes better use of screen display space, at the cost of creating some visualization ambiguities. In the simplest implementation, line cutting formats an expression naturally (as in the scrolling mechanism described above) and then cuts the result into screen-sized widths, displaying each portion on a separate line. This eliminates the need for horizontal scrolling (but typically incurs the need for vertical scrolling), and

4.2 Displaying Large Expressions

Despite the best efforts to reduce the quantity of information to be presented when displaying a mathematical expression, expressions must frequently be rendered in their entirety. If it is the case that the expression will not fit completely on the workstation screen, some mechanism must exist to provide the user with piecewise viewing access to the entire result.

Some possible techniques for handling this problem, namely horizontal scrolling, line cutting and line breaking, are discussed below. It is worth noting that the majority of mathematical expressions tend to be wide, not tall (matrices are an obvious exception); the effectiveness of both line breaking and line cutting tend to depend on this observation.

4.2.1 Scrolling

From a technical standpoint, simple scrolling is the easiest way to provide the user with full access to large expressions. In such a scheme, the expression can be formatted "naturally" without regard to display device size limitations; this both simplifies the layout phase and preserves traditional two-dimensional mathematical notations. The result is a single line expression over which the user can scroll in both the vertical and horizontal axes. When the expression is not taller than the display window (as is frequently the case), the vertical scrolling capability can be dispensed with.

Neither the rendering phase nor the process of selection is complicated by scrolling. When a scrolling operation is executed, only the x and y positions of the expression's root box need be changed before re-rendering from the root; those tends to be ineffective for wide, shallow expressions. Another possibility would be to provide a magnification window which can be moved interactively by the user; unfortunately, performance constraints may make this approach impractical.

Fish Eye View

A view which has the potential to be more useful than the straightforward zoom view is the "fish eye" view, originally suggested by Fumas [FUMA86]. In a fish eye view, full details are provided at the point upon which the user has focused their attention, while other details are elided according to a *degree of interest*, or DOI function. Fumas suggests the following function for trees

$$DOI_{fisheye}(tree, f, x) = -(d(tree, f, x) + d(tree, root, x))$$

in which x is a node in the *tree* with the user's attention focused upon the particular node f. The function $d(tree, x, y)$ returns the path length between x and y in *tree*. In the function above, the two terms favour nodes that are close to either the focus or the root of the tree, respectively. Soiffer points out that this DOI function is not well suited for large mathematical expressions, which tend to be wide and not deep, and instead suggests that physical distance be used instead of tree distance. Furthermore, the function should be tuned to increase the weights for the first and last terms of wide expressions, and discriminate among delimiters according to their mathematical importance.

An effective fish eye view should permit the user to change the focus quickly and easily, in order to allow panning over the entire expression; it may be difficult to obtain adequate interactive performance for this task.

may or may not be desirable from the user's standpoint.

Iris does not support elision, mainly because it does not provide support for convenient selection within output expressions. User controlled elision is planned for *xiris*, although it has not yet been implemented.

4.1.3 Alternative Views

One of the advantages that CAS interfaces have the potential to provide over traditional pencil and paper methods is an easy way to look at problems or results from another angle. No CAS interface to date has made a serious attempt to provide "visualization views", although some ideas along these lines have been suggested before.

Satellite View

A "satellite view" of a mathematical expression is nothing more than a rendering of the entire expression so that the entire expression is visible at once. For large expressions, details will not be visible in such a rendering; however, the intention is that the viewer may be able to extract structural information about the overall form of the expression from its general shape.

This scheme is probably useful only for expressions that contain notations that have visually distinctive shapes, such as matrices. Once expressions become too large, visual cues become too small to be useful, unless a specific attempt is made to make them stand out. One possibility for improvement might be to display the mathematical notation (such as operators) associated with the upper n levels of the expression tree clearly, and omit the details of the operands; however, this scheme (but not necessarily identical) constructions when the reader can readily deduce the form of the expressions being elided, as in $x^2+2x^4+3x^8+\cdots+nx^{2n}$. The latter usage is particularly common when dealing with large or arbitrary sized regular matrices, since matrices tend to require large amounts of space for their representation.

Unfortunately, CAS interfaces are rarely able to make intelligent guesses about what portions of expressions are eligible for elision. This is because there are virtually no heuristics that are generally applicable over a wide range of problem domains. Early versions of MathScribe provided automatic elision functionality, but Soiffer reports that users did not find it very valuable.

User-controlled elision is a useful tool in CAS interfaces. Typically the user must select the subexpressions to be elided in some manner (often with a mouse or other pointing device), and then request that the interface elide the selection. This can be inconvenient for subexpressions that do not fit entirely upon the screen, and requires some care on the part of the user to ensure that they do not deceive themselves. On the other hand, the elided expressions can be expanded again if desired, which constitutes an advantage over the paper and pencil method.

Elision need not overly complicate the formatting and display process, although attention need be given to several details. Obviously all user-initiated elision or expansion operations require a redisplay operation at the very least, and possibly a partial reformatting as well, depending on the sophistication of the display representation structures. MathScribe uses Soiffer's incremental updating methods [SOIF91, pages 78–81] for maintaining bounding boxes to handle this problem. Matrices require special attention, especially to handle elision of entire rows or columns rather than just individual entries. Finally, it is worth noting that if a DAG structure is used to represent bounding box information, manual elision within one instance of a common subexpression can affect all occurrences, which

```
> inverse(vandermonde([x,y,z,t]));

 [       y z t            t z x            t y x              y z x       ]
 [     - -----          - -----            -----              -----       ]
 [        %1               %2               %3                 %4         ]
 [                                                                        ]
 [ t y + t z + y z  x t + z x + t z    x t + y x + t y    y x + z x + y z ]
 [ ---------------  ---------------  - ---------------  - --------------- ]
 [       %1               %2                 %3                 %4        ]
 [                                                                        ]
 [     t + y + z        x + t + z        x + t + y          x + y + z     ]
 [   - ---------      - ---------        ---------          ---------     ]
 [        %1               %2               %3                 %4         ]
 [                                                                        ]
 [         1                1                  1                  1       ]
 [       ----             ----             - ----             - ----      ]
 [       %1               %2                 %3                 %4        ]

                          2              2              3    2
%1 :=    t y x - y z t - x  t + t z x - x  y + y z x + x  - x  z

                             2            2              3    2
%2 :=   - t z x + t y x - x y  + y z x - y  t + y z t + y  - y  z

                             2            2              3      2
%3 :=   - t z x + t y x + x z  - y z x + z  t - y z t - z  + y z

                             2              2            3      2
%4 :=   - t y x + y z x + x t  - t z x + y t  - y z t - t  + z t
```

Figure 4.1: Subexpression Labelling in IRIS

possibility is to use the *width* and *depth* characteristics of the tree or DAG representing the expression as the basis upon which to make labelling decisions. Maple's Iris interface uses an estimate of the displayed width of a subexpression, along with a few other heuristics to decide whether or not a subexpression should be labelled; in addition, the user controls a subexpression width variable which determines the degree of labelling. Figure 4.1 shows the results of Iris labelling; *xiris* provides identical labelling capabilities.

In addition to selecting the subexpression to be labelled, the CAS interface must also choose a variable name to represent it. To avoid conflicts with user defined variables, it is generally desirable to choose such label names from a distinct name space. Iris (and *xiris* as well) label expressions with names of the form %*number*, reserving all such names as labels. Of course, once the labelling has been performed by the CAS interface, the user must be permitted to access the subexpression with its label. Furthermore, the CAS itself need not be informed about the labelling process at all if the interface handles the task of translating label references in user input back to the appropriate subexpression; Soiffer [SOIF91] points out that failing to do so (thus permitting the CAS to generate results in terms of the substituted name) can lead to subtle errors if functional dependencies become hidden by the substitution.

4.1.2 Elision

Elision of detail (also known as "collapsing output") is another technique used by both mathematicians and CAS interfaces to enhance the visualization process, by replacing those portions of an expression which are not relevant to the problem at hand with ellipses. Another common use for this notation is to represent repetitive implicitly know the resultant form of the expression being manipulated, and can write the results of the manipulation in a manner which conveys this information naturally.

When manually performed computations become complicated, mathematicians generally rely on two similar techniques to enhance the comprehensibility of their work. The first of these is the labelling of complicated or common subexpressions, while the second is elision of detail. Both techniques can be used by CAS interfaces with varying degrees of success. Finally, the interface can provide alternative views which may be beneficial to the visualization process.

4.1.1 Labelling Subexpressions

Labelling (or renaming) is performed by making a simple substitution of a new variable for a subexpression in the displayed version of the expression, and then displaying the relationship between the variable name and its value separately. Labelling is most effective for large subexpressions occurring more than once within the overall expression, but can be useful even when a complicated subexpression occurs only once, since such a substitution can aid in visualizing the overall form of an expression.

Labelling is not difficult to implement in the CAS interface from a technical standpoint; a simple symbol table can handle most book-keeping tasks associated with such substitutions. The more difficult task is choosing the subexpressions that are to be candidates for labelling. When using paper and pencil, mathematicians can make effective choices by making use of their additional knowledge of the problem domain at hand; the CAS interface rarely has this information available, and must make labelling decisions based upon the form of the expression alone. One impact excessively upon the size of the problems that can be handled by the CAS.

Displaying large expressions can also impose performance penalties within the interface, where "performance" is considered to be the user's perception of the responsiveness of the interface. Performance can be considered from two standpoints. First, there is the time required for formatting a result produced by the CAS; from the user's perspective, this delay is usually buried in the CAS computation time. Second, there is the delay time of refreshing the display when the user scrolls the display to see another portion of an expression, or an earlier result; perceived sluggishness in this operation can not be hidden, and must be minimized in order to reduce user frustration.

This chapter addresses each of these three potential problem areas in turn; the implementation details of how *xiris* handles these issues are deferred until Chapter 5.

4.1 Visualization

When discussing the subject of effective *visualization* of mathematical expressions, we are talking about the degree to which the user comprehends the mathematical content of the expression being viewed, generally from a structural standpoint. As a contrived example, consider the output generated by Maple when given the input expression `expand((x - 1)^300)*t`. While the result expression is a product, this information is not readily apparent from the displayed output, which the user may have to inspect with some care to determine this fact. This problem does not occur when using traditional pencil and paper methods because the user performs and controls all simplifications and manipulations manually, which implies that they are unlikely to accidentally perform large expansions unnecessarily. Furthermore, they

Chapter 4

Handling Large Expressions

Large expressions pose problems for the CAS interface from several standpoints. The most important problem for the user is one of visualization; complicated mathematical expressions (and especially wide ones) can be difficult to manage using traditional means such as pencil and paper, but forms that are generated by a CAS can be considerably more complex. The user of the pencil and paper has the advantage of knowing what portions of the expression at hand are of primary interest; unfortunately, the CAS interface typically does not possess this information, so it is difficult to make intelligent decisions about how to display large expressions in a reasonably useful manner.

Other major problems posed by large expressions are primarily technical in nature. CAS systems themselves tend to be frugal with their memory requirements for algebraic structures, since this facilitates solving larger problems. However, the overhead posed by maintaining a complete hierarchical box structure as outlined in Chapter 3 can easily expand these storage requirements by a factor of three, making effective memory management a serious issue if the interface is not to handled by testing the integrand to determine if it is itself an integral, and choose the size of the symbol accordingly, but of course, if the mathematical semantics of the integrand is not available, this scheme cannot work. The alternative is to use a single fixed sized symbol of integration, as does TEX in the examples below.

$$\int_{i=0}^{n} x_i$$

$$\int_{i=0}^{n} \frac{x-y}{1+\frac{x-z}{t-2}}$$

Some special formatting cases cannot be reasonably handled — some because they cannot be anticipated, and others because they occur infrequently or represent such inappropriate notation that they are not worth implementing code to handle them. As an example, consider an excessively complicated expression that is used as a bound on a definite integral. While perfectly meaningful from a mathematical standpoint, such expressions are too awkward and confusing to be written as such in normal practice; instead, the substitution of a dummy variable for the complicated bound is normally performed. Such a substitution can be made by the CAS interface (see the discussion on labelling in Chapter 4), but depending on the implementation cost, it may be more practical to let the user handle these infrequent cases manually.

Expressions which are inherently multi-line in nature may also require some form of special treatment. Matrices are an example of such an object, in which each row is placed on a logical line of its own. Small matrices which can be displayed on the screen in their entirety present no difficulties, but large ones which are either too wide or too tall have the potential to present difficulties if a linebreaking scheme is employed to handle wide expressions. Chapter 4 discusses linebreaking and its ramifications in more detail.

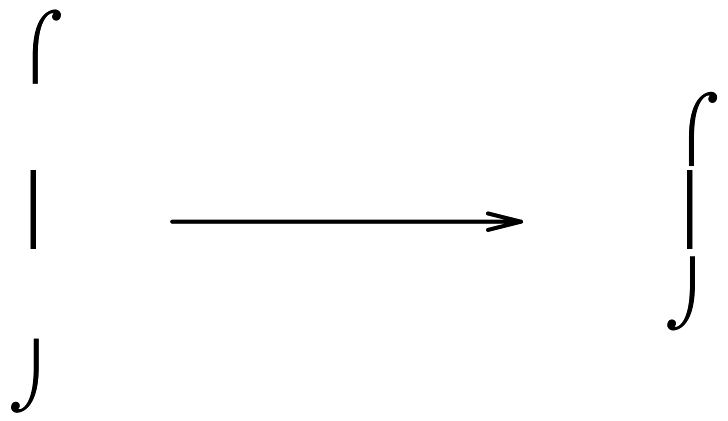


Figure 3.4: Glyph Creation by combining bitmapped characters

3.2.3 Specialized Formatting Cases

Although carefully designed formatting directives will usually be able to do a good job laying out most expressions, there are always particular expression forms or special cases that will cause difficulties. In most cases, these expression forms cases will result in substandard or aesthetically unpleasant formatting; in the worst case, the expression may be displayed in an ambiguous or deceiving manner.

Generally, once these exceptional cases are recognized, the only recourse is to enhance the capabilities of the formatting directives responsible. This may be impossible if the directives are encoded in a primitive or constraint-based system, and can be difficult for procedural encoding methodologies as well. As an example, consider the case of formatting integrals. Many systems that format mathematics will vary the size of the integral sign according to the size of the integrand, or, in practical terms, the size of the bounding box of the integrand. In the case of a double integral, this implies that the outside integral sign will be larger than the inner one, since the outer one encompasses not only the integrand of the inner integral, but the integral symbol and the limits as well. This special case can be

3.2.2 Device Independence

Display device independence is relatively easy to achieve when using a formatting scheme similar to the one discussed above. In particular, almost all device dependencies are isolated to two areas: first, in the encoding of the fonts to be used for rendering (this information is obtained from the metrics of Table 3.1), and second, in the detailed knowledge of how to render text at a given position.

Unfortunately, some mathematical notation requires drawing capabilities beyond the simple rendering of characters. For example, the fraction bar of a large quotient can be implemented as a single call to a line drawing graphics primitive, while a radical sign may be generated through a series of such calls. In general, any notation that varies with the size of the expression being annotated will usually require something other than the use of a set of bitmapped fonts. Hence, routines that render variable sized brackets, integral signs, radical signs and fraction bars typically require access to other drawing primitives.

Scalable fonts provide an easy way to handle some of these cases, but if they are not available, bitmapped character composition can be used. Character composition is a very simple concept: multiple elements of a bitmapped font are combined in order to create a single glyph which is perceived by the user as one character unit. Sometimes additional graphics primitives are used in conjunction with the font elements; as an example, *xiris* creates variable-sized integral signs by combining a vertical line segment with two characters from the mathematical symbol. The two symbol font characters represent the top and bottom portions of the integral symbol, while the joining line segment can be of whatever length is appropriate to construct an integral sign of the desired size. Figure 3.4 illustrates this concept.

3.2 Output Issues

The techniques and issues discussed in the previous section are generally applicable to most systems which are capable of formatting mathematics. However, in the case of formatting the output of CASs, other considerations quickly become apparent.

3.2.1 Large Expressions

One of the greatest strengths of any CAS is the ease with which it can generate and handle large expressions. With very little input, a user can generate expressions of virtually unlimited size. As a somewhat contrived example, consider that the expansion of $(x-y+t+v)^{50}$ contains more than 23,000 terms, with many coefficients exceeding 20 digits. Of course, such expansions are rarely of interest; however, by accident or design, expressions too large to be displayed at once are frequently the results of CAS computations.

Large expressions pose problems for the CAS interface designer from several standpoints. On one hand, there are technical issues to be considered; the formatting and display of large expressions places heavy demands on the resources of the system formatting the expression – a straightforward implementation of the formatting scheme suggested above requires memory quantities proportional to the size of the expression being formatted. On the other hand, big expressions pose problems from a visualization standpoint; the larger an expression, the more difficult it is to comprehend its general form, and thus identify patterns or other structurally-based content. Because large expressions pose large problems for the CAS interface, Chapter 4 is devoted to their discussions.

bounding box structures lends itself well to isolating the device-dependent rendering routines from the remainder of the system. All that the rendering routines require is information about *what* to display, and *where* to display it, and both are readily available from the box structures.

When using interactive display devices, such as a workstation screen, it may be desirable to use tricks such as double buffering to reduce display flicker. Double buffering is a simple technique in which the image being rendered is constructed in off-screen memory, and then moved to the display as a unit. Most modern graphical environments provide primitives for supporting double buffering; if a platform does not, the degree of functionality required for our purposes is easily simulated.

The majority of the displayed image of the mathematical expression is built up from strings of characters; indeed, there is surprisingly little mathematical notation that is not character-based. Hence, virtually all drawing is ultimately performed by the system-provided text rendering graphics primitives. Their use is generally straightforward, although a minor consideration is whether or not the characters are to be rendered with an opaque background. If so, the character cell is filled with the background colour, and the character drawn on top; the alternative scheme draws the character directly without the background fill. There may be a performance penalty associated with one scheme or the other; *xiris* makes use of the first method, since it is faster in its implementation environment, X11. Unfortunately, the opaque background technique complicates rendering text adjacent to slanted characters, since when used naturally, the bounding boxes of italics overlap. *xiris* solves this problem by using a coarse box granularity, which has the benefit of permitting the strings of italic characters making up a single leaf box (representing an identifier, for example) to be rendered with one call to the system's text rendering primitive.

Font Property	**Description**
MIN_SPACE	The minimum size of an inter-word space.
NORM_SPACE	The width of a normal space character.
SUPERSCRIPT_X	Horizontal offset for superscripts.
SUPERSCRIPT_Y	Vertical offset for superscripts.
SUBSCRIPT_X	Horizontal offset for subscripts.
SUBSCRIPT_Y	Vertical offset for subscripts.
UNDERLINE_POS	Vertical offset for underlining.
UNDERLINE_THICK	Thickness of the underline.

Table 3.1: Font Property Information Used by *xiris*

Table 3.1 contains a list of the font properties used by *xiris*. Unfortunately, even the items on this relatively small list are not always defined by the fonts available under X11; in such cases, approximated values must be assumed for the missing properties.

3.1.2 The Rendering Pass

The rendering pass walks the data structures created by the layout pass and displays the contents of the bounding boxes at the appropriate relative locations on the display. Generally speaking, rendering is fairly simple compared to the layout process, and there are correspondingly fewer issues worthy of consideration. Complications can arise when dealing with large expressions; these are addressed in Chapter 4.

The actual process of drawing on the workstation display screen (or other output device) is, of course, system dependent; however, the recursive traversal of the

the CAS version, code may be required to convert back to the CAS representation.

A design decision required with respect to bounding boxes is the definition of the degree of granularity to use. One approach is to define individual boxes for each character in a string; a more efficient technique for output considers strings to be the primitive unit of construction, rather than characters. A disadvantage of the latter approach is that it complicates the picking process within the primitive strings.

Fonts and Font Dimensions

Ultimately, the majority of the rendered expression is displayed with elements of one or more fonts. The sizes of the bounding boxes for primitives such as variable names and numeric constants have to be determined based on the sizes of these font elements. Consider the process of formatting a simple numeric constant; since the constant will be rendered as a string of digits, the attributes of the bounding box enclosing the string depend solely upon the characteristics of the font in which the string is rendered.

In addition to the basic dimension information, other facts about the input fonts are useful, such as the location of super and subscripts, and the thickness and location of underlining. Italic fonts and fonts with serifs have additional width information, such as the degree of slant which has to be considered when juxtaposing them with upright fonts. TEX makes use of more than 20 font parameters when handling mathematical fonts; however, most of these parameters are not made available on the display fonts used by current workstations. The relatively low resolution of workstation display hardware makes their absence a moot point, unless some form of sub-pixel text antialiasing is employed.

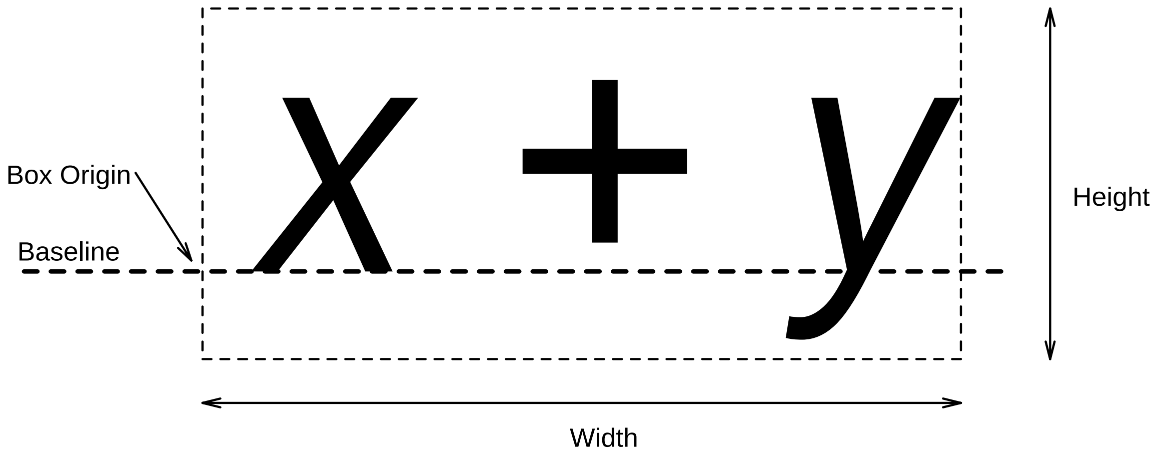


Figure 3.3: Attributes of a Bounding Box

Alignment The location of the box's baseline. This value is used for the side-by-side alignment of boxes, and corresponds to the baseline of plain text contained within the box.

Children A list of the sub-boxes contained within this box.

Figure 3.3 illustrates the definitions of some of these attributes.

It can be worthwhile to fold the mathematical semantics of the CAS tree node being represented into the bounding box structure, rather than maintain two totally disparate data structures. One obvious technique is to place a pointer in each box that refers back to the appropriate node in the CAS representation; complications can arise, however, if there exists no direct correspondence between the two forms. For example, since Maple represents quotients such as $1/x$ internally as x^{-1}, there is no node corresponding to the formatted numerator in the CAS representation. Another approach is to simply add an operation field to the box representation that records the appropriate CAS primitive. This scheme eliminates the need for both representations, but depending on how faithfully the display expression parallels

CAS interface.

Constraint-based Constraint-based formatting uses predefined relationships between box attributes to locate specific boxes at execution time. Typically this involves solving a series of equations in order to satisfy the constraints, which might incur a significant performance penalty at runtime. The INFORM formula editor [EGMO89] makes use of this technique; in addition to specifying the structure of expressions with a grammar, information encoded in the same grammar defines the display layout for the expressions. Specifications in the grammar define how the children of a particular type of expression are to be located, and how the dimensions of the resulting bounding box are to be computed. This scheme is not extensible at run time, however, since the grammar file has to preprocessed by a parser generator to create source code that is part of the INFORM system.

Bounding Boxes

The minimal set of information required to be stored as part of a bounding box typically includes the following attributes, although various implementations will alter their semantics to some degree:

Location The location of the box, usually represented as the X, Y coordinates of the box's origin. The bounding box's origin is frequently defined as the intersection of its baseline and its left side. The X, Y coordinates may be relative to the parent box, a sibling box, or even to the root of the entire expression.

Dimensions The dimensions of the box (width and height), which determine the extent of the bounding box.

```
layout_Exponential( dag, fontsize, parent )
ALGEB   dag;       /* CAS form of expression */
int     fontsize;  /* Current fontsize in effect */
pdbox_p parent;    /* Attachment point of formatted expression */
{
    pdbox_p base, expon;

    /* Allocate storage for a box structure, (two children). */
    parent->box = allocBox( IB_EXPON, fontsize, 2 );

    /* Format the exponent and the base separately.  Note that
     * the exponent is formatted to be one size smaller. */
    layout_expr( dag[1], fontsize,
                         base  = &parent->box->kids[0] );
    layout_expr( dag[2], fontsize + 1,
                         expon = &parent->box->kids[1] );

    /* Compute the dimensions of the resultant box, and the
     * location of its baseline (copied from the base), and
     * locate both the exponent and base boxes within it. */
    parent->box->h = expon->box->h + SUPERSCRIPT_Y(fontsize);
    parent->box->w = base->box->w + SUPERSCRIPT_X(fontsize)
                                  + expon->box->w;
    parent->box->c = base->box->c;
    base->box->y = h - base->box->h;
    base->box->x = 0;
    exponent->box->y = 0;
    exponent->box->x = base->box->w + SUPERSCRIPT_X(fontsize);
}
```

Figure 3.2: Formatting Procedure for Exponentation

cannot be described completely satisfactorily. Second, even relatively simple notations may be difficult to codify with the primitives. Soiffer suggests that a WYSIWYG construction tool can help alleviate this problem.

Procedure-based The procedure-based approach is probably the most commonly used technique in current CAS interfaces. MathScribe, MathStation and Maple all use this method, in which formatting procedures are written with full control over the underlying box structures. This technique provides considerable flexibility and the most control over the formatting process, but is not extensible unless the interface's implementation language supports runtime loading and linking for user procedures. MathScribe (using Lisp), and MathStation (PostScript) can thus support user-written formatting procedures, but neither the existing Maple interface, Iris, nor *xiris* can. An example of a formatting procedure from *xiris* is displayed in Figure 3.2.

Macro-based Macro-based formatting is similar to the procedure-based techniques in terms of how the notation format is defined, but are similar to primitive-based methods insofar as they do not permit direct access to the underlying box structure. Furthermore, although they are inherently extensible, macro-based techniques require considerably more in terms of runtime resources, since both memory and execution time requirements are greater than for their procedural equivalents.

TEX is the best known example of a macro-based formatting system, and consists of a set of primitives and a macro language that permits user-defined sequences to be constructed. While TEX is very capable when it comes to formatting mathematical expressions, it is not clear that a purely macro-based formatting methodology would perform adequately in an interactive

Formatting Directives

The formatting directives embody exactly how the given expression is to be displayed. Typically, these instructions are encoded and executed based on the type of expression at hand. For example, the process of formatting the expression x^2 will exercise the directives for formatting a variable, a constant integer, and finally, an exponentation operation.

Unfortunately, mathematical notation is not static. Various mathematical disciplines frequently use differing notational schema, while entirely new notations for new mathematical objects are often developed. Hence, limiting the formatting directives to a static set is not desirable; it should be possible to enhance or replace the functionality represented by the basic body of formatting directives. Soiffer [SOIF91] classifies four techniques for the codification of extensible formatting directives; these approaches are summarized below.

Primitive-based A primitive-based technique defines a set of basic formatting operations (the primitives) and uses them to implement all of the formatting functionality of the system. Users are able to override or augment existing formatting capabilities by constructing new directives based on the system-provided formatting primitives, but direct access to the underlying box structure is not available. Mathematica provides a basic, character-based version of this functionality for enhancing notation on ASCII terminals. Soiffer proposes a list of primitives suitable for implementing primitive-based formatting on a bitmapped display screen, but to date, no CAS interface system appears to have employed this scheme.

Two shortcomings of this technique are apparent. First, unless the provided primitives are very comprehensive, there are likely to be notations which

4. A bounding box is created to encompass the arrangement of child bounding boxes created in the previous step.

The bounding boxes of most leaf nodes and annotations can be determined from the bounding box of their string representation when rendered in the desired font. Some objects need to be handled as special cases: for example, the bounding box of a root sign depends on the dimensions of the bounding box that is to appear under it.

The Input Expression

The form of the input expression is not of particular importance to the operation of the layout stage, as long as it has knowledge of the format. Of course, encoding the knowledge of the internal representation of a CAS expression in the formatting routines limits the reusability of the code; this is usually a concern only if the CAS interface is designed for interoperability between different CASs. In [SOIF91], Soiffer discusses issues related to generalized translations between mathematical notation and internal CAS formats. For our discussion of output formatting, we presume that a simple mapping from the internal CAS format to an abstracted form (upon which the layout phase will operate) will suffice.

In the development of *xiris*, it was decided to forgo the added complications imposed by converting Maple's internal representation of mathematical expressions to a common abstract form. Instead, the *xiris* formatting routines operate directly upon the Maple form of the expression. This has the benefit of increased formatting performance – not only is the intermediate translation step omitted, but additionally, full advantage can be taken of the Maple DAG structure [CHAR91] by formatting common subexpressions only once.

intention to display) the formatted expression structures the *rendering* phase.

3.1.1 The Layout Phase

The layout phase is responsible for the construction and positioning of the hierarchical bounding box structures that represent the expression; it is this pass that embodies the actual formatting process. Input to this pass consists of three major items. First, the expression to be formatted is obviously required. Second, some form of direction as to how particular mathematical forms are to be displayed is needed. Finally, information about the dimensions of the fonts used to render the characters and symbols making up the expression is necessary. As output, the phase produces a data structure (typically a tree or a directed acyclic graph) composed of bounding box structures which can be used as input to the rendering phase for final display. The box-structure representation of the expression, which we call a *display expression*, is also required for the operation of *picking*, or determining the results of tracker-based selection (for example, selections made with a mouse).

Although the details of constructing the display expression vary between implementations, the basic strategy is a simple one. The input expression tree (or DAG) is traversed in postfix order, and at each internal node, the following actions occur:

1. Any children of the node are formatted recursively.

2. Bounding boxes are constructed for any annotations required by the node that are not represented in the input data structure.

3. The bounding boxes of the children and annotations are arranged with respect to each other.

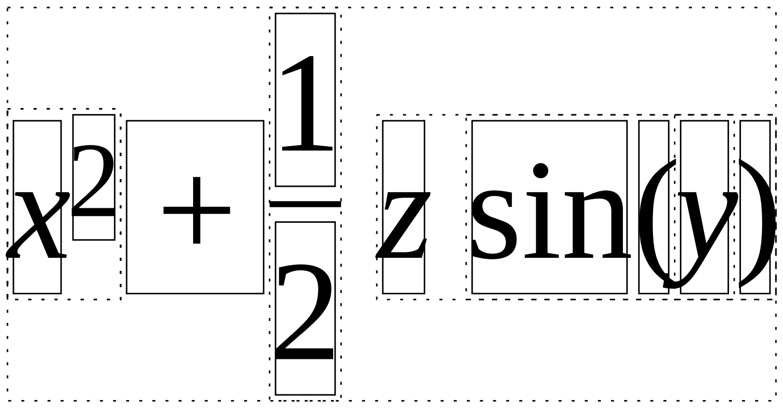


Figure 3.1: Hierarchical Arrangement of Bounding Boxes.

3.1 Basic Formatting

Mathematical notation is inherently two-dimensional. The traditional representations of exponents, fractions and matrices all require the juxtaposition of symbols in two dimensions to convey their meanings. One common scheme used to handle the task of formatting a mathematical expression uses two passes. The first creates a *bounding box* (or just "box", for the sake of brevity) for each subexpression, and then positions the subexpressions' bounding boxes appropriately. The overall effect that results is a hierarchy of boxes within boxes, with the outermost box containing the entire expression, and is illustrated in Figure 3.1. The second pass then walks the structure embodying the bounding box hierarchy, and renders the box contents at the appropriate locations. This well-known technique was used by Martin [MART71] in the late 1960's, and has been used by many systems that format mathematics since, including Knuth's TEX system [KNUT84].

For the purposes of this discussion, we call the construction of the bounding boxes the *layout* (or formatting) phase, and the subsequent traversal of (with the

Chapter 3

Formatting CAS Output

This chapter discusses general techniques and issues related to the rendering and display of the mathematical output generated by CASs, and the approach used in the implementation of *xiris* in particular. More specific details of the formatting methodology used in *xiris* are discussed in Chapter 5.

The primary goal of the output formatting process is to show the results of CAS computations to the user in a manner that facilitates easy comprehension. This implies presenting results using the basic notation traditionally used in mathematical publications, in order to reduce the cognitive load imposed by displaying results in a non-standard format. Unfortunately, mathematical notation is not particularly standardized, making some level of user control over notation desirable.

Many of the difficulties encountered in formatting CAS output stem from the fact that the generated output can be arbitrarily large. Large expressions pose technical problems for the implementor, and visualization problems for the user. These issues are addressed at length in Chapter 4.

into the standard Maple interface.

Other CAS Interface Features Other CAS interfaces have distinct and useful features that may be worth incorporating into the Maple interface.

Affecting Existing Users It is often undesirable to implement sweeping changes in the interface that require existing users to completely relearn the program, or require substantial changes to the algebra engine.

The last point is particularly important when the new interface is intended as a replacement for an existing one. Maple, as a relatively mature program that is available on a wide variety of platforms, is no exception, insofar as a fairly diverse set of interfaces already exist to support Maple versions running on Unix, VAX/VMS, MS-DOS, and the Macintosh, to name a few.

On the other hand, the practical advantages of a single, portable interface across many platforms are readily apparent. Users need learn the capabilities of a single model, while documentation costs can be drastically reduced. Hence, the benefits of a common interface over multiple platforms outweighs the requirements that users learn a new, consistent interface.

present more kernel state information to the user. An effective implementation of this solution is hampered by the degree of kernel/interface separation that exists in Maple, described earlier. A second solution would be to make the interface function in the manner of a "spreadsheet" program, in which any and all commands occurring after (or depending upon) the current one in the session are re-executed in order, thus bringing the session log "up to date." The main problem with this solution is that such automatic recomputation of all the subsequent commands could require excessive amounts of computation time, much of which is probably unnecessary. Unfortunately, it is very difficult to determine which results require recomputation and which do not when an arbitrary line of input is changed.

The problem of providing adequate state information to the user in worksheet-based Maple interfaces has not yet been solved.

2.3.3 Environmental Considerations

Environmental considerations take into account the constraints imposed by the real world. Many of these considerations are based on tangible issues such as implementation cost, while others are less concrete. Software engineering decisions tend to be driven largely by these considerations.

Commonality It is desirable to have a common Maple interface that runs on as many platforms as possible.

Portability Related to commonality, the interface should be as portable as possible, in order to facilitate code reuse over multiple platforms.

Maple Features Since very diverse Maple interfaces already exist on various platforms, it is desirable to incorporate the good ideas from these implementations exacerbates the problem. Techniques such as using shared memory instead of Unix pipes could alleviate this problem, although byte-stream I/O will likely be required for any heterogeneous kernel-Iris execution combinations.

The Worksheet Interface Model

Interest has been expressed by members of the Maple user community in the notion of a "worksheet" based interface model for Maple. Such an interface acts as a combination of a rudimentary word processor and medium for "scratch pad" mathematical calculations; indeed, the Maple implementation for the Macintosh already embodies a modest amount of this functionality. Some attention must be paid to the details of implementing such an interface model, however.

Cowan and Wein [COWA90] observe that many graphical user interfaces present application state information to the user, but do not make the session history available; on the other hand, command line interfaces generally provide the history, but without the state feedback. The line-oriented, sequential programming language model used by Maple falls into the second category, and its traditional glass TTY interface displays the commands and their results sequentially upon the display. When the user is given the ability to edit and re-enter computations at arbitrary locations in the session, it becomes easy to create a worksheet that can "lie" to the unwary observer. This comes about when the sequence of inputs to the Maple kernel (and the kernel's subsequent state, such as which names are assigned) cannot be deduced from the sequential examination of the interface's session log: the visible "history" may not correspond with the kernel's state.

Two potential solutions to this problem come to mind, although neither one is completely satisfactory from the Maple standpoint. First, the interface could direct use by most applications, it fits the distinct Maple kernel and Iris design very well. Accordingly, the Maple design permits considerable flexibility in the interchange of interfaces, both of different types and over different platforms.

This separation is not without its disadvantages, particularly in the case in which the kernel and the interface run as distinct processes. Indeed, the problems outlined below need not occur on platforms in which the computation engine and the interface components are compiled into the same address space.

One obvious weakness in the current design stems from the necessity of communicating state information from the kernel to the interface. Currently there is no way for the Iris to query the kernel's state — for example, to determine which function names are loaded in order to provide name completion.

A related problem (again based on the fact that the kernel has a state distinct from that of the interface) is the difficulty of saving a Maple "session". In order to fully recreate a work session, both the interface state and the kernel state need to be saved; in this context, the complete kernel state must be transmitted to the Iris. Similarly, the Iris needs to be able to restore the kernel's state when loading a saved session.

Finally, the process-based separation technique raises some efficiency considerations. Computational results generated by the kernel can be very large; these results must be transmitted to the interface for presentation to the user over a byte-stream communications channel. In environments in which the kernel and the interface execute on the same machine, two copies of the structure are created, one in each process. Displaying plots is even worse — current interfaces simply copy the data structures representing the plot to a third process, which actually performs the rendering. The fact that plot data structures tend to be large only established word processors available: implementation of yet another one, albeit with sophisticated symbolic computation features, is probably not the best way to serve the user community. A better approach would be to provide rudimentary outlining and text manipulation capabilities in a Maple interface, along with flexible mechanisms for exporting results to established full-functioned word processors.

Perhaps the only effective solution to addressing the major uses of Maple outlined here is multiple, special purpose interfaces. The division of the Maple kernel and the user interface facilitates this approach; furthermore, if the existing Maple Iris were to be broken down into smaller and more autonomous functional units, then these components could be "mixed and matched" to provide multiple interfaces while minimizing programming efforts. For example, it is easy to imagine replacing the existing parser which accepts the full Maple language with one that accepts only single-statement Maple expressions, and accordingly does not require the trailing semicolon.

Kernel/Interface Separation

As was mentioned earlier, current implementations of Maple that run on Unix-based systems separate the Maple kernel and the interface into two distinct processes which communicate via pipes. This design facilitates (and indeed, enforces) complete interface and application separation.

The notion of separating the application and the interface is a well-known UI design principle based loosely on the Smalltalk Model-View-Controller paradigm [KRAS88]. This idea is the basis of Hartson's discussion on runtime architecture design [HART89], in which the application is divided into a computation component and a dialogue component. Although Hartson's model tends to be too general for

Uses of Maple

One of the most important considerations in UI design is to be cognizant of the purpose or uses of the application. In some cases, this is a fairly straightforward issue — the application may have a narrow but well-defined scope, or a very small user community with specific needs. Interactive "tool" applications can be more difficult to pin down, and to a large degree, Maple falls into this second category.

In essence, Maple is nothing more than an interactive, interpreted programming language for symbolic computation. The interpreted nature of the language facilitates its use as an interactive symbolic calculator, however, and many users never use Maple for anything more sophisticated than that. Current Maple interfaces make no attempt to address this dichotomy of use; rather, they force the user to either recognize the fact that a programming language is being used for even the most simple interactive calculation, or condemn users to a perpetual distrust of the system. As an example, consider the following transaction, typical of any user's first interaction with Maple: the user types `2 + 3` and presses the Enter key; Maple responds with another prompt, but with no sign of the expected result. That the programming language requires a semicolon at the end of statements, and that newlines are permitted within statements are the facts that the user needs in order to understand just what has happened here, and why no result has appeared.

Maple is used by researchers from many disciplines, and embedding the results of Maple computations in scientific papers being prepared for publication is a frequently performed (but rather awkward) task for this type of user. This fact, and the feedback from a rudimentary "notebook" style Maple interface available on the Apple Macintosh has demonstrated that some users want to use Maple as a "mathematical word processor". One danger here is that there are already a myriad of name ambiguties. Such a mechanism can be useful in CASs that provide a programming language interface, or a large set of system-defined functions. The Maple system in particular meets these criteria; providing name completion for the 2000 or so functions in the Maple library would facilitate the use of longer, more descriptive function names, which in turn helps the current problem of naming inconsistencies and overloading in Maple.

The desirability of providing keyboard command sequences to accomplish all interaction tasks is a generally accepted interface principle. This permits experienced users to perform tasks without moving their hand from the keyboard to a pointing device, which can be a time-consuming operation. Although some operations are usually faster and easier to perform with a mouse, a keyboard interface can provide more control and precision in delicate operations, such as the selection of a small displayed object. Most user interface toolkits provide a standard interface paradigm for mimicing mouse use with the keyboard.

2.3.2 Application Specific Considerations

It goes without saying that the nature of the application itself drives many interface design decisions. This section describes some of the important attributes of the Maple system that have influenced the design of *xiris*. Some of these factors are generally applicable to most CAS interfaces, while others are concerns that stem from the architecture of Maple itself.

Multiple Command Paths One technique for handling the problem of addressing the needs of different types of users is to provide more than one way to accomplish tasks. This permits novice users to be "single-stepped" through a complicated task, while experienced users can take advantage of a faster method with minimal prompting.

Although this can be an effective strategy, it is not without its drawbacks: implementation and documentation costs are increased, while the interaction of multiple methods can confuse users. Furthermore, the burden on the user is increased, since although the user is free to use whichever method they prefer, they have to learn both methods before being able to make use of this flexibility.

Command Accelerators One simple but effective technique for addressing the needs of both the novice and the experienced user is the menu accelerator. Menu accelerators are commonly available in most window system toolkits, and permit the user to make menu selections using a single keystroke or keystroke combination without actually displaying the menu. For example, the Apple Macintosh uses the Option key to accomplish this task. Menu accelerators avoid many of the problems of multiple command paths, and work well for commonly used commands that appear on menus.

Command name completion can be a useful accelerator mechanism, although its utility is dependent upon the nature of the application. To invoke name completion, the user types the first few characters of the name, and then presses a "completion" key: if the prefix entered by the user uniquely identifies a name known to the system, the remaining characters are filled in automatically. Additional prompts for the user can be provided to resolve

Support for Advanced Users

Although the point was made earlier that all user types must have their needs addressed, the subject of facilitating the use of the application by advanced users is brought up here again. As with many of these interface design considerations, there are a myriad of possible techniques for providing customization and shortcuts, and their effectiveness is often determined by the nature of the underlying application. Some ideas that appear to be relevant to CAS interfaces and the Maple system in particular are presented below.

Configurability Frequent users that have achieved a reasonable level of familiarity with an application and its interface will typically want to exercise some control over interface customization in order to facilitate the completion of regularly performed tasks. One well-known and inexpensive method for providing basic interface programmability is the notion of assignable keyboard commands. Such a scheme permits the user to bind frequently used commands or command sequences to single keystrokes (often in conjunction with a modifier such as the Control key); more sophisticated interfaces allow the creation of keyboard macros in which any command or data input action can be recorded and subsequently played back.

One problem with this form of configurability is that once a user has customized their interface to any appreciable degree, it becomes virtually unusable by anybody else. Macros (or other interface sub-programs) tend to be difficult to construct; their effectiveness depends on both the imagination of the user and the quantity of repetitive actions they need to perform. However, once such customizations have been put into place, they can improve usability on a personal basis by a considerable amount.

actions, and the best way to make them accessible to the user is highly dependent upon the nature of the application, and the way in which the application is being used.

In most CAS interfaces, it is desirable to place emphasis on making navigation between results easy, and to facilitate viewing of results that may be too large to display at once. As CAS interfaces become more sophisticated, the direct manipulation of expressions displayed on the screen will likely make up a large part of the common interactions performed by the user; designing the semantics and bindings of direct manipulation actions requires careful attention to be effective.

Visualization of Results The mental assimilation of the results produced and displayed by the interface of an application frequently places considerable cognitive demands upon the user. The amount of mental processing required by a user to comprehend any particular result depends both upon the nature of the result itself, and the manner in which the interface presents it. Thus, representing results as "naturally" as possible can greatly facilitate usability, both by reducing the workload on the user, and by decreasing the possibility of user errors in interpreting results.

Virtually all CAS interfaces tend to be weak in the area of visualization of mathematical expressions themselves, although considerable effort has been expended in the area of rendering mathematical plots effectively. Improving visualization is one of the themes of this thesis, and is discussed in detail in Chapter 4.

Layering of functionality is present at the application level in the Maple system via the Maple *package* concept, which permits the loading of a set of related mathematical functions by user request.

Consistency The benefits of providing a high degree of consistency in user interfaces is one which has received considerable attention from the software industry over the past six or seven years. Inter-application consistency has been promoted by development toolkits and runtime environments on both microcomputers and workstations alike, with a reduced learning curve and increased user confidence being cited as two of the prime benefits.

An effective user interface must maintain internal consistency as well. This implies that the commands and their resulting behaviours should be as consistent as reasonably possible; this permits users to effectively predict the form and results of new and unfamiliar commands based on the knowledge gained from earlier experiences with the interface.

In addition to being consistent, choices for command names and menu selections should share several other properties. Ideally, they should tend to favour infrequently used words or phrases, with specific meanings, that aid discrimination between included and excluded operands. As an example, for file manipulation commands, "duplicate" would be preferable to "copy", and "rename" to "move".

Naming conventions in the Maple library could stand improvement from the standpoint of consistency.

Facilitate Common Actions Although it almost goes without saying, making common actions and commands easily accessible is very important in reducing user stress levels. The identification of common command or manipulation

The problem facing the user interface designer is to facilitate the use of the application by users in all three categories, while at the same time maintaining a sufficient degree of consistency so that users can "switch categories" smoothly as their proficiency level increases.

Imposed Cognitive Load

Whenever users interact with an application, some portion of their mental energy is expended on the interaction tasks necessary to achieve the results they desire. Thus, an important part of making an application "easy to use" is minimizing the cognitive load imposed by using its interface. To a large extent, the way in which this may be accomplished is highly dependent upon the nature of the application itself; nevertheless, there are several basic principles that serve the general case.

Layered Functionality Users (and especially novices) are easily overwhelmed when presented with big menus, long lists of commands, or large numbers of buttons: the meaning or effects of their underlying functionality are easily lost in the shuffle of visual complexity. One way of handling this problem is to remove or split off the superfluous functionality: although not everyone will agree, we would assert that multiple programs (with consistent interfaces) are superior to a single monolithic program embodying the same capabilities.

For inherently complicated applications, layering their functionality can work well. This technique makes less frequently used commands available at a lower level in a hierarchical command structure, rather than clustering all commands at a single level. Careful organization of the hierarchy is required, in order to facilitate the location of infrequently used commands.

principal goal is clear: maximizing the value of the user's time spent using the application.

Addressing Different Types of Users

The fact that no two users are identical tends to complicate interface design decisions. One possible approach to handling this problem of addressing the needs of users with different skill levels is to classify the spectrum of users into three categories.

Novices Novices are those users who have little to no experience with the application itself, although it is usually presumed that they have some knowledge of the subject matter manipulated by the application.[1] They may or may not have experience with other (possibly unrelated) applications.

Infrequent Users Infrequent users are well versed with the capabilities of the application, but do not use it on a regular basis. Such users typically know what they want to accomplish within the context of the application, but do not necessarily remember the exact steps required to achieve their goals.

Frequent Users Frequent users are not only familiar with the application and its limitations, but use the program with sufficient frequency that they do not require appreciable guidance from the system as they work. These users are the so-called "power users", although even the most sophisticated user may occasionally require some assistance when performing rare or unusual operations in a large application.

[1]Educational applications are one obvious exception.

in the representation of the expression given in (2.1). For the sake of comparison, the *xiris* rendering of the same expression is visible in Figure 5.1.

2.3 Interface Design Considerations

The overall design of any user interface is typically governed by a host of considerations, which can be loosely divided into three sets. The first of these are general interface considerations, and can be thought of as those guidelines which embody "good interface design principles." The second set, application specific considerations, consists of application dependent interface factors — considerations which are driven by the nature of the application for which the interface is being developed. Finally, the third set contains practical factors that are completely external to the application and its interface. These environmental considerations generally take the form of real-world constraints, such as imposed implementation deadlines or portability requirements.

The principal design considerations encountered when designing *xiris* are discussed below, and are classified according to category. As in most interface designs, conflicts between these considerations exist, especially between those in different categories.

2.3.1 General Interface Considerations

A considerable body of literature exists on the subject of principles of user interface design; only a few highlights are presented here. It is interesting to observe that some design conflicts arise even within these general considerations, although the

```
z+1/3*z**2+Int(x**2*ln(x)*y**(1/2),x = -Pi .. infinity)


                    infinity
                       /
           2           |       2        1/2
z + 1/3 z  +           |      x  ln(x) y     dx
                       |
                      /
                    - Pi
```

Figure 2.1: Character-based Output Forms in Maple

and ASCII character-based two dimensional output. Input is parsed by an LALR(1) parser compiled from a YACC grammar [JOHN78], and while common operators are handled naturally, most mathematical semantics are expressed with a function call-based notation, in which normal infix operators can be embedded more or less naturally. As an example, the expression

$$z + \frac{z^2}{3} + \int_{-\pi}^{\infty} x^2 \ln(x)\sqrt{y}\; dx \tag{2.1}$$

is entered with the following Maple command line (the spaces are optional):

```
z + z^2/3 + Int(x^2*ln(x)*sqrt(y),x=-Pi..infinity);
```

Two formats for output are provided. The first displays results in a "lineprinted" form, which is analogous to the input format accepted by the Iris parser. This format is useful for feeding the output back into Maple. The second output format is an ASCII character-based two-dimensional form, in which traditional mathematical notation is approximated as closely as possible within the limits imposed by standard glass TTY terminals. Figure 2.1 demonstrates both of these output forms existing library functions and adding completely new functionality. At the same time, it limits the computational resources required by the Maple system to the minimum needed to handle the algebraic problems at hand – a valuable feature on microcomputer systems with limited memory.

2.2 The Existing Maple Interface

The benefits of the separation of the CAS functionality from the interface components was recognized early on in the development of the Maple system. A design plan for a conceptually distinct interface for Maple [LEON86] was presented at the 1986 Symposium on Symbolic and Algebraic Computation; the emphasis of this report falls on the advantages and practical considerations of a complete and clean separation of the interface and the algebra engine. The existing Maple interface, known as Iris, implements a subset of this design.

The greatest advantage to this complete functional separation of the user interface from the underlying algebra engine is the flexibility that it affords. Not only can new interfaces be developed with total independence from changes in the CAS, but multiple interfaces targeted at different types of users can be provided, each using the same algebra engine. Where practical, the Maple kernel and Iris can run as separate processes connected via a system-dependent communications channel (as an example, Unix implementations use pipes for this purpose); this can be easily extended to permit the kernel to run on one machine, and the interface to run on another, with the two communicating over a network communications channel. Such an arrangement permits the two processes to run in environments appropriate to their respective tasks.

The user interface presented by Iris is a basic one based on command line input

2.1 Maple

Maple remains an on-going research project of the Symbolic Computation Group at the University of Waterloo. The design and implementation effort was initiated in mid-1980; the goal was to capitalize on experiences gained from earlier CAS systems, such as Reduce and MACSYMA, while at the same time taking advantage of advances in software engineering to construct an effective and portable system. The result is a relatively small kernel which interprets the Maple mathematical manipulation language; the majority of the algebraic knowledge represented by the Maple system is embodied in a large set of library functions written in the high-level Maple language.

Until recently, Maple's primitive data structures and the basic operations performed upon them (such as numeric arithmetic and elementary simplifications) were implemented in a locally-developed macro-preprocessor language called Margay. On any particular target machine, the Margay macros were translated into a platform-dependent dialect of the C programming language and compiled as a traditional C language program; the resulting executable is known as the Maple kernel. The use of the Margay preprocessor considerably enhanced the degree of portability achieved in the design of the Maple kernel, especially in the early days of non-standardized C compilers. The kernel has been ported to a wide variety of operating environments, including the Atari ST, the Commodore Amiga, IBM VM/CMS, and many Unix platforms. Recent versions of Maple have dispensed with the macro preprocessing step, and are implemented directly in C.

Basic I/O functionality is provided by the kernel, which permits the dynamic loading of library functions on a demand basis. This scheme permits considerable flexibility, by providing a convenient means for enhancing the capabilities of

Chapter 2

Maple Interface Design Considerations

This chapter is divided into two distinct parts. The first portion deals with Maple itself, and provides the reader with some background on the design of the Maple system and its existing interface.

The second part of this chapter is devoted to a discussion of some of the considerations which motivated the design decisions made in the development of the *xiris* interface for Maple. Some of these considerations arise from the nature of the Maple system itself, while others stem from generally accepted principles of interface design.

A description of the capabilities of the *xiris* interface is deferred until Chapter 5.

A "top down" approach is taken within the body of this thesis, by first addressing the design considerations for the overall interface, and the factors upon which the design decisions were based. Subsequent chapters deal with specific sub-areas of the interface, namely output formatting and techniques for handling large expressions, with emphasis on the particular problems posed by the nature of the underlying CAS.

At the same time, discussions of CAS interface design principles occur at two distinct levels. At the conceptual level, factors that influence the overall design of the user interface are presented; these factors can loosely be considered the "classical interface issues." These discussions are mainly confined to the opening chapter. The second level of considerations are the practical issues governed by the particular problems that CASs impose on interfaces in general. These are addressed in the subsequent chapters in the context of the affected subarea of the interface.

A functional description of *xiris* as it stands currently, and some of the more interesting details of its implementation, are presented in Chapter 5. This chapter also contains a description of the *xiris* linebreaking mechanism, which utilizes a novel variation of a previously unimplemented linebreaking algorithm.

A slightly different category of CAS interface is represented by the related MathCAD and MathStation [MATH89] systems. Both have interfaces that permit the user to freely intermix text, equations and graphics on the same or over multiple lines. A "spreadsheet" recomputation model is used to update any calculations and graphics that appear below any changed inputs: since neither system embodies a complete CAS, the recomputation times are not excessive. MathStation permits the user to define their own customized formats for operators (through writing PostScript code). Although both systems have been connected to Maple to provide full symbolic computation capabilities, neither can format the medium and large expressions that a full-fledged CAS can produce.

Theorist [BONA87] is a new but small CAS with a fairly sophisticated interface that runs on the Macintosh. It supports a variety of methods for entering mathematical expressions, but the most interesting feature of its interface is the support for the direct manipulation of the displayed expressions: subexpressions can be selected and moved within the overall expression, or substituted into variables. The interface endeavours to ensure that only legal manipulations are allowed, although the question of determining legality is difficult to solve even with a locally available CAS.

1.2 Scope of this Thesis

This thesis touches upon a variety of topics in the discussion of CAS user interfaces, and presents them in light of an implementation of a new user interface, *xiris* for the Maple system. The emphasis is placed on the formatting and presentation of the mathematical expressions generated by the algebra system, since to date, the *xiris* implementation does not include any sophisticated input or editing features.

as troff (with eqn) [OSSA78] and TEX [KNUT84] can handle sophisticated mathematical layout, at the cost of having to learn their cumbersome linear syntaxes. Many of the more recent interactive document processors such as FrameMaker [FRAM89] and Microsoft's Word for Windows [WORD91] support the WYSIWYG ("what-you-see-is-what-you-get") layout and formatting of equations. These systems do not communicate with CAS engines, however, and are not designed to handle large expressions.

Attempts have been made to take advantage of the mathematical formatting knowledge provided by these document processing systems. These efforts have been primarily directed at the batch oriented systems, and fall into two general categories. First are those systems that generate eqn or TEX forms of their CAS output; their intention is to facilitate the creation of publication-quality documents from CAS sessions. The principle problem faced in these systems is handling the linebreaking of large expressions: Antweiler, Strotmann and Winkelmann's TEX-generating engine for Reduce [ANTW89] is one of the few systems that addresses this problem. The second category attempts to use document processing systems to perform interactive formatting of CAS results; these efforts, such as Foster's DREAMS [FOST84] system for MACSYMA tend to suffer from poor performance, and have generally been abandoned.

Leler and Soiffer's 1985 Reduce pretty printer [LELE85] was the first system since Martin's work in the 1960's that could handle both input and output, and served as the basis for Soiffer's MathScribe [SMIT86] system. MathScribe is the most ambitious CAS interface developed to date, and supports a variety of input techniques, bitmap-based formatting and display of traditional mathematical notation as well as the interactive editing of displayed expressions. MathScribe does not break large expressions over multiple lines.

up some of the shortcomings of traditional CAS interfaces. Furthermore, the advent of readily available, low-cost bitmapped display terminals, coupled with higher user expectations with respect to general interface quality and capabilities has led to an increased demand for better CAS interfaces.

1.1 Related Work

A comprehensive treatment of CAS interface development efforts is provided by Soiffer in the introduction to his dissertation [SOIF91]; only the highlights, along with some of the more recent work in the field are presented here.

Initial attempts to improve CAS interfaces focused on improving the quality of their output. The earliest effort in this direction was Clapp and Kain's Magic Paper [CLAP63] system, developed in 1963, which provided two-dimensional formatting and notation for mathematical expressions. One early system that was particularly capable (even by today's standards) was Martin's Symbolic Mathematics Laboratory [MART71] which could display mathematical expressions on a vector-based display using multiple fonts, and permit the user to select subexpressions with a light pen.

The interactive systems of the late 1970's and 1980's, such as MACSYMA [MACS83] and Maple, provide only line-based input and character-based formatting of mathematical expressions; Figure 2.1 shows an example of this formatting style. Until very recently, this form of output represented the "state of the art" of the mathematical formatting provided by CAS interfaces.

Over this same time period, techniques for the computerized typesetting of mathematical expressions have evolved steadily. Batch oriented text processors such experiences obtained from the design and partial implementation of a new interface, called *xiris*, for a particular CAS system, Maple [CHAR91].

Despite the relative maturity of computer algebra systems, most of their interfaces have provided only rudimentary input and display capabilities. By modern standards, many of these interfaces can be classified as primitive. Users enter expressions in a linear, "programming language" style using only those characters available on a standard keyboard, while results are presented as lines of ASCII character-based text.

It is probable that a combination of factors has been responsible for this apparent lack of evolution in CAS interfaces in general. In the case of Maple, it is apparent that its own programming language paradigm has been a particular influence in delaying the production of a high quality interactive interface. The Maple system consists of a relatively small kernel, which implements primitive speed-critical operations and acts to interpret libraries of user code; these libraries embody the majority of Maple's functionality. Both developers and users tend to use an external editor to enter Maple programs or complicated series of calculations, and then invoke Maple upon their program file to test and debug the results. This methodology has alleviated the need for effective input (and to a lesser degree, formatted output) facilities within Maple itself. Furthermore, development efforts by the University of Waterloo's Symbolic Computation Group have – not surprisingly – focused on improving Maple's symbolic computation capabilities, rather than upon interface improvements. Finally, the design and implementation of an effective CAS interface requires a variety of interdisciplinary skills, including knowledge of software engineering and human-computer interaction factors, as well as familiarity with the CAS and the particular problems it poses for the interface.

The increased use of CASs as interactive mathematical scratchpads has shown

Chapter 1

Introduction

Computer programs that manipulate mathematical expressions symbolically are not a particularly recent innovation. Libraries of routines for handling symbolic computations were developed as early as the mid-1960's [SAMM66], while programs designed to handle specific problem types, such as symbolic differentiation and integration, date from even earlier [NOLA53, KAHR53, SLAG63]. Modern computer algebra systems (or CASs) are considerably more powerful, and handle a wide range of problems over a diverse set of mathematical domains. Soiffer provides a brief overview of many of these systems and their capabilities in [SOIF91].

Although most modern CASs provide some degree of programmability (typically in the form of some sort of embedded programming language), an interactive "calculator" mode is usually made available in order to provide accessibility to the casual user as well as to facilitate mathematical scratchpad calculations. For the most part, the effectiveness of these systems has been hampered by the lack of user interfaces that adequately address the issues peculiar to CAS requirements. This thesis attempts to outline and consider these interface issues through the practical

List of Figures

List of Tables

Contents

Acknowledgements

Many people have helped to make this thesis and the software a reality. I would like to thank the members of the Symbolic Computation Group for their patient explanations of Maple internals, as well as their helpful comments on the evolutionary progress of the interface software. David Clark deserves special mention—his knowledge of the existing interface software was invaluable.

My thanks also go to the Computer Graphics Laboratory, which provided encouragement and a comfortable development environment as well as financial support through the kindness of John Beatty. I would also like to gratefully acknowledge the scholarship support received from Alias Research.

My thesis readers, Richard Bartels and Keith Geddes, deserve particular thanks for their valuable feedback on this document. To my supervisor, George Labahn, I owe a particularly large debt of thanks. In addition to providing the vision, funding, and encouragement, George gave me unlimited freedom to explore the issues, yet was always available to discuss the problems as they arose.

My parents deserve much credit for the immeasurable support they have provided for my studies throughout the years. Finally, my wife, Irene, has contributed infinite patience and understanding, even when the "extra term" stretched into an extra year. I couldn't have done it without her support.

Abstract

In recent years, the evolution of user interfaces for Computer Algebra Systems has lagged behind both the advances in the algebra engines which they serve, and the general developments seen in user interface principles as a whole. The increased availability of bitmapped display devices and the graphical user interface software running upon them has served to demonstrate the limitations of existing computer algebra system interfaces.

This thesis discusses some of the design and implementation issues inherent in the construction of an effective user interface for computer algebra systems. The emphasis is upon the efficient handling of the output of algebra systems, and effective techniques for presenting the mathematical content to the user, particularly when the expressions are large. A new user interface for the Maple system is described in light of the issues discussed, and a new variation on an algorithm for breaking long mathematical expressions over multiple lines is presented.

Mathematical Output Presentation in User Interfaces for Computer Algebra Systems

by

Timothy Richard Tyhurst

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1993