EQUATION EDITOR FOR MULTIDIMENSIONAL SCIENTIFIC DATA

Andrew Llewelyn

Department of Software Engineering
University of Auckland, Auckland, New Zealand

Abstract

The creation of visualisations from scientific data often requires that the data be transformed to display the information required. No tools currently exist that allow manipulation of the equation in both typeset and tree view. Many of the current applications and toolkits for scientific visualisation are not available on more than one platform, thus a requirement of the application produced for this project is that it is portable to all major platforms. The design for the application was split into several modules to ensure that the code stayed maintainable as the project developed. One of the major goals for the application is extensibility. Due to the limited time available for the implementation of the project, it is not practical to develop an application with all the functionality required for manipulating arbitrary scientific data built into it. The application developed is very extensible, allowing the future addition of new data types, operators and functions. Implementation of the project proceeded in two stages. After user testing on the first version of the application, it was decided to restructure the architecture and implement a second version. This version was found to be more usable. The final version of the application splits the internal representation of the equations into three modules, one for each of the typeset view, tree view and evaluator. The application created meets all the goals of the project.

1. Introduction

When working with scientific data for the purpose of generating visualisations, it is often necessary to develop transformations to alter the data to suit the needs of the representation. The transformations are applied as equations which act on the fields being used to create the visualisation. The task for this project is to develop an application that will allow these transformation equations to be developed quickly and efficiently, and then allow them to be applied to data sets.

1.1. Need for the application

There is currently no single application that provides all of the functionality desired.

1.2. Existing applications with similar functionality

1.2.1. Microsoft Equation Editor [2]

Microsoft Equation Editor (actually a cut down version of the application 'MathType' made by Design Science, Inc.) comes with Microsoft Office. It provides typesetting functionality only, allowing users to create representations of equations within documents for printing. The interface provided for constructing the equation is similar to that desired for the typeset view for the application created for this project. This application is available for Microsoft Windows only.

1.2.2. MATLAB [3]

MATLAB is a very powerful matrix based programmable mathematics tool. It allows the creation of advanced transformations using a custom scripting language. MATLAB does not provide typesetting support. It is, however, capable of creating visualisations based on input data and equations. The programming interface can be cumbersome to use, and has a fairly steep learning curve for new users. MATLAB is available on all major platforms.

1.2.3. Mathcad [4]

Mathcad provides the ability to create and evaluate equations in typeset form. It is a very powerful application which provides most of the features required in the application produced for this project. It does not provide a tree view for manipulating the equations however. Mathcad is available only for Microsoft Windows.

1.3. Goals

The goal of this project is to create an equation editor application with the following characteristics:

- Provide a typeset view of equations
- Provide a tree view of equations
- Allow editing of equations in both views
- Evaluate equations
- Provide an API for extending the application to support new data types, functions and operators

2. Design

The overall design for the application was split into three sections as follows:

2.1. User interface

The design of the user interface is perhaps the most important aspect as far as the user is concerned [1]. As this is the main view they get of the application, most of the impressions they have of the system are based on the user interface. The user interface for the application has to function in ways similar to existing programs to allow new users to learn how to use it rapidly. The interface also has to behave consistently when the user performs an action, and ideally the behaviour exhibited should be what the user would expect to result from their action.

The overall design of the user interface follows the general pattern used by applications such as Microsoft Visio, Eclipse and AutoCAD. Figure 2.1.1 illustrates the layout of the application. The left hand side of the interface is devoted to a sidebar from which equation elements can be dragged. The right hand side is split into two sections containing a typeset view and a tree view of the equations being manipulated. Across the bottom is a text box into which equations can be typed, and then dragged into either the typeset or tree views. The drag and drop aspects of the interface allow new users to get up to speed quickly, and the text interface allows experienced users to get more performance out of the application.

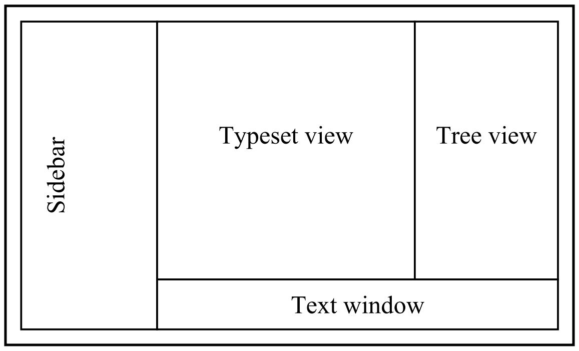

Figure 2.1.1: Layout of the user interface

2.2. Serialisation format

The format used to serialize the equations for storage on disk has to meet several requirements. It has to be:

- Easily understood by humans to allow it to be used for tasks not anticipated by the designers
- Compatible with a wide range of other applications
- Simple to parse into an internal representation so that implementing it does not take a disproportionate amount of the project's time

The format chosen is the same as is used to enter equations into Microsoft Excel and most programming languages such as C/C++, C#, Java, Visual Basic, etc… This format is easily understood by humans, and does not contain extra redundant formatting information. Equation 2.2.2 illustrates the serialised version of equation 2.2.1.

$$\frac{\sqrt{\sin(5x+2)}}{4+6\cos(x)} \tag{2.2.1}$$

sqrt (sin (5 * x + 2)) / (4 + 6 * cos (x)) (2.2.2)

As an added advantage of using this format, the same parser used for reading saved equations can be used to extract equations from the text box in the user interface. This reduces the amount of code that needs to be written to get the functionality of the equation editor implemented, allowing more work to be done in other areas of the application.

2.3. Extensibility

An important requirement for the application is extensibility. To this end, the design of the program must be modular, with clear areas of responsibility between each module. This makes it clear what each modules function is, and ensures the program is more maintainable. The program needs to be written with as few things as possible implemented as special cases, so that the functionality can be extended in a generic manner. An overall view of the modules planned for the application is given in Figure 2.3.1.

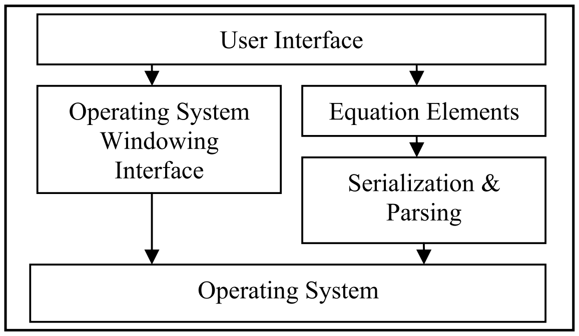

Figure 2.3.1: Overall design of the modules comprising the application

3. Implementation

The implementation of the design has fairly broad requirements to meet in most areas. Both implementations described in this document make use of a structure used by most compilers/interpreters to represent the equations. This structure is the AST (Abstract Syntax Tree). An AST structures the parsed data into a tree of nodes containing all useful information. They are generally dependent on their children for such things as evaluation, size for rendering, etc… This implies a depth first ordering for traversing the tree. The structure of the tree makes the order of operations for an equation implicit (no need to know if multiply is evaluated before or after addition after the tree has been constructed). The order in which children of a node are evaluated is determined by the node itself, and is generally left-to-right.

3.1. Languages used

The main factor considered when deciding on what languages to use for the implementation of the project was portability. To this end, C++ was used to implement the application.

The parsers were constructed using Flex/Bison because they work well with C++ and provide a very powerful and flexible interface for writing LALR(1) parsers. Generation of an AST from a Bison parser is a relatively simple task; in a properly constructed grammar the nodes are parsed in depth first order allowing children to be passed up to higher levels to be used in the construction of parent nodes.

The graphical user interface library is implemented on top of the OpenGL graphics library. The decision to implement a custom graphical user interface library for this project was made because it ensured that the application was more easily portable without having to install large additional libraries such as Tcl/Tk. It also allows complete flexibility in deciding how the user interface can be interacted with, and ensures that it is consistent across all platforms.

3.2. User interface library

To ensure the application's portability and to provide maximum flexibility in the implementation of the editor views and visualisation output, a custom user interface library was written on top of OpenGL/GLUT. The library uses a hierarchical class structure to represent the various controls and windows used by the application. An abbreviated view of this structure is shown in Figure 3.2.1.

Rendering is performed through a Canvas object which abstracts away the underlying graphics library, potentially allowing the interface to be ported to another library. Each Canvas object contains the bounds within which it can draw, and maintains a scissor rectangle around this region to ensure the display cannot be corrupted by an incorrectly implemented control. Fonts are implemented by Font objects, which can be rendered on a Canvas. This allowed wgl to be used to implement the font rendering on Win32 platforms, while still allowing the transparent use of GLUT fonts on other platforms. A similar system can be implemented to take advantage of glx for X-Windows platforms.

Keyboard and mouse input is handled through the RootWindow object. The RootWindow is responsible for tracking which control currently has the keyboard focus, which control is currently under the mouse (for implementing roll over effects), and also handles drag and drop between controls.

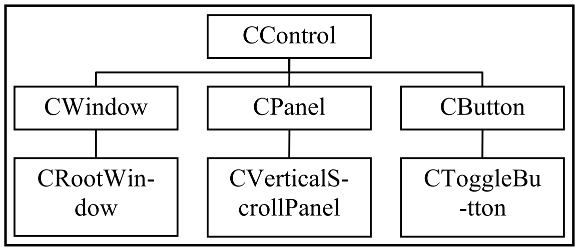

Figure 3.2.1: Abbreviated diagram of the basic class hierarchy for the user interface library

3.3. User testing

As each major part of the application was completed, an informal user test was performed. The target audience was a small group of students, some with a lot of experience with other existing applications, and others with very little. During the testing, the students attempted to construct several simple equations. Their comments on the usability of the interface were recorded and used to improve the next iteration of development.

3.4. Initial approach

The initial approach taken on implementing the design for this project had one parser and AST structure to handle all manipulations of the elements forming an equation. Each element of the equation was responsible for:

- Rendering itself in both the typeset and tree views
- Serialization to a plain text format
- Type checking and operator overload resolution

This system was originally chosen because it allowed the application to be built incrementally. This made it possible to begin development of the application and examination of the problems encountered to be undertaken without requiring the entire architecture to be designed before any coding was done. Unfortunately the architecture that resulted from this method of development suffered from several fundamental shortcomings as outlined in the following sections.

3.4.1. Tree structure too restrictive for typeset view

User testing highlighted several areas in which the tree structure placed too many restrictions on the manipulation of the typeset view. The users tested frequently wanted to split up children of adjacent operator nodes using parentheses. For example, in the equation 3.4.1 it would be intuitive to be able to drop a set of parentheses on either '+' operator and have it surround the adjacent literals. Because of the underlying tree structure there is a set of implicit parentheses as shown in 3.4.2. This prevents the user from adding visible parentheses around the sub expression '6 + 7'.

$$5+6+7 \tag{3.4.1}$$

$$((5+6)+7) \tag{3.4.2}$$

This is clearly undesirable behaviour, but it is at best very difficult to solve cleanly without changing the underlying tree structure to one incompatible with the tree view. The work-around for this problem in the initial version of the application was to drag the entire sub expression into the text window at the bottom, edit the serialized form manually, and drag it back into the equation.

3.4.2. Element nodes too complex

The element nodes had to be complex because they contained the functionality of what is really three separate areas of responsibility, namely typeset representation, tree representation and type resolution/evaluation. Changes made to attempt to resolve issues uncovered during usability analysis were difficult to implement as there was large amounts of coupling between portions of the code, and so alterations often resulted in a the propagation of a large number of changes to supposedly unrelated code.

3.4.3. Disparities between apparent and actual order of operations

Because the same internal tree structure was used to represent both the typeset and tree views, many special cases had to be developed to accommodate situations arising from disparities between apparent and actual order of operations. The most obvious of these are situations that arise when the division operator is used. When division is written in serialized form, it appears as in 3.4.3.

$$5*6/7*8 \tag{3.4.3}$$

The user needs to be able to represent each different variant of this equation that results from parentheses being placed in different locations. However, in each of these equations the parentheses should not be displayed as it is apparent which sub expression is evaluated first. 3.4.4-6 show several examples and their serialized forms.

$$5\times\frac{6}{7}\times8 \Rightarrow 5*6/7*8 \tag{3.4.4}$$

$$\frac{5\times6}{7}\times8 \Rightarrow (5*6)/7*8 \tag{3.4.5}$$

$$\frac{5\times6}{7\times8} \Rightarrow (5*6)/(7*8) \tag{3.4.6}$$

This situation can be resolved by removing parentheses that are around the arguments to the division operator during parsing, and always inserting parentheses around the arguments during serialization. This solution works satisfactorily and has the additional benefit of also being appropriate for the tree view representation.

3.4.4. Parenthesis stripping in tree view

Parentheses necessary for specifying order of operations in the typeset view were also displayed in the tree view because of the shared internal representation. Attempts to resolve this by hiding unnecessary parentheses were enjoyed only limited success. Complications and special cases in the code arise as soon as the user is allowed to manipulate the tree view. This is because the application has to decide what to do with the hidden parentheses in each situation. It also has to decide when to insert new hidden parentheses to maintain consistency with the typeset view.

3.4.5. Conclusions

The initial shared internal representation architecture, while being invaluable in enabling usability issues to be discovered early in the project, is unsuitable for the final version of the application for the following reasons:

- Work-arounds are needed for several operations that a user is likely to want to perform frequently
- Many situations need to be special cased in the code resulting in an interface which is not easily extensible due to the high level of coupling between sub modules
- Too many restrictions are placed on how the user can write equations because of the type resolution. This would be better performed as a separate function in conjunction with evaluation

3.5. Revised approach

In order to resolve the issues exhibited by the initial version of the application, a new architecture was developed to implement the functionality required. This revised design is based on three parsers instead of the previous one. Each parser has a separate and distinct area of responsibility as outlined in the following sections.

3.5.1. Tree view

This parser creates an AST containing only the information strictly necessary for rendering a tree view. Parentheses are stripped out as the order of operations is apparent from the tree structure itself. During serialization of the tree structure, the precedence of the operators is examined to determine where parentheses are needed to faithfully reconstruct the tree. Figure 3.5.1 shows trees with the same layout and the parentheses required in their serialized form to reproduce the tree after parsing.

The parentheses are needed in Figure 3.5.1 (a) to ensure that the last '+' operator is evaluated before the middle one. While it would be mathematically correct to leave out the parentheses in this case, a different tree would result after parsing (the last '+' operator would become the root, and hence the last evaluated).

No parentheses are needed in Figure 3.5.1 (b) because the tree structure follows the BEDMAS rules.

Two sets of parentheses are needed in Figure 3.5.1 (c). The first set are needed to ensure that '-' operator is evaluated first, and the second set to prevent the tree from reordering to place the last '*' operator at the root (this is the same situation as in Figure 3.5.1 (a)).

A simple set of rules based on the precedence of each operator and whether it is left or right associative determines where parentheses are needed.

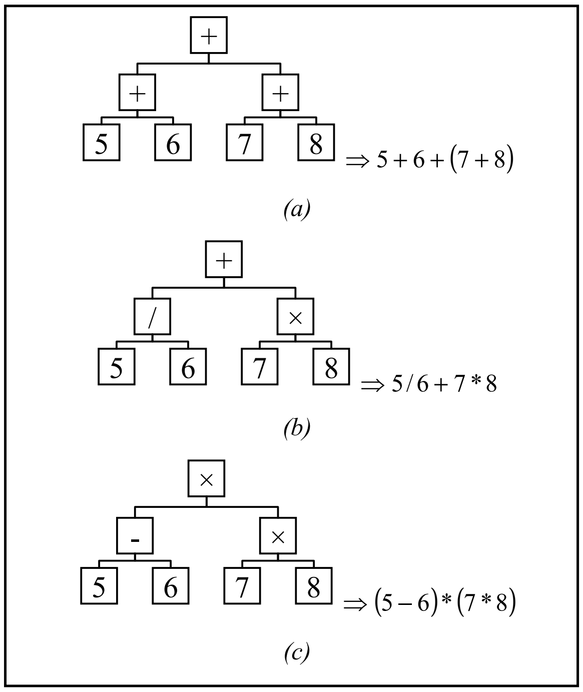

Figure 3.5.1: Situations where parentheses are needed to maintain tree layout during serialization.

3.5.2. Typeset view

The parser for the typeset view creates an AST whose structure is much flatter than that of the tree view. In order to allow more intuitive editing of the equation, a distinction is made between a node consisting of one term (e.g. a literal, variable or function) and a node consisting of a series of terms (e.g. term, '+', term, '×', term). Figure 3.5.2 illustrates the difference between the AST's for the tree and typeset views.

The flattened AST allows the user interface to be written to enable the user to edit the equation in the same way one would edit a line of words formed from letters in a word processor. Because the structure isn't restricted to a tree like it was in the initial version, the user is able to drop a set of parentheses on top of the '+' operator and cause the sub expression '6 + 7' to be evaluated first.

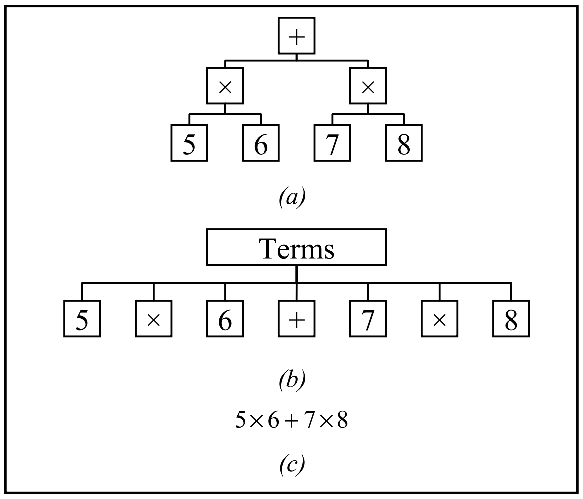

Figure 3.5.2: AST structure for the tree view (a) and typeset view (b). The serialized form is shown in (c).

Selection is handled in a similar way to how it is handled in Microsoft Word. If the user starts dragging the cursor from in the middle of a terms node, it will select adjacent terms until the end of the terms node is reached. If the user continues to drag past this point, the selection origin moves to the parent node and extends to include the entire terms node. The user can thus select any set of adjacent terms for manipulation. Figure 3.5.3 illustrates how the selection moves as the mouse is dragged across the equation.

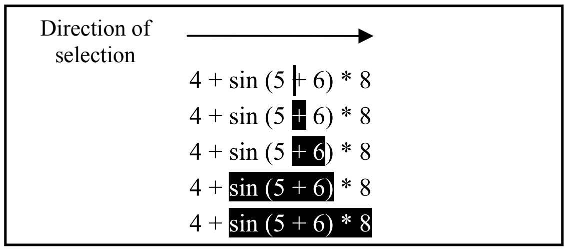

Figure 3.5.3: Selection of multiple adjacent terms within the typeset view

3.5.3. Evaluator

This parser creates an AST which contains only the information required to resolve the types of each node, and to evaluate the equation against a data set. The structure of the tree is the same as that of the tree view, with parenthesis nodes stripped out.

3.5.4. Parser interactions

All interactions between the parsers occur through the serialization interface. The AST's generated for the tree and typeset views are capable of serializing themselves – either in their entirety or a sub tree. When a sub tree is dragged from one view to another, it is the serialized form of the AST that actually gets communicated between the views.

Error checking performed by the evaluation parser is passed to the other views by specifying where in the serialized form the error occurs. All nodes in the views that are parsed from a section of text containing an error are marked as an error so that the user can make the appropriate corrections.

3.5.5. Conclusions

The revised structure for the internal representation of the equations allowed us to develop the user interface to be more intuitive in the way it responds to interactions. Word processor style interactions are now possible within the typeset view. Unnecessary parentheses are hidden in each of the views. No arbitrary restrictions are placed on the users actions within the editor by attempts to maintain type consistency because the types are not validated until evaluation time.

4. Results

The final application produced for this project provides all the core functionality to allow a user to create and evaluate simple equations. The application provides a good base to build upon to develop a fully fledged equation editor. During the development of the application it was reinforced that the design of program must be flexible at all times; results from user testing can often result in large changes being required to the architecture.

5. Future work

The following items are recommended as things to be developed for the application in the future:

- Addition of new data types, operators and functions to allow more advanced equations to be developed
- Development of a library to allow dynamic extension using a scripting language
- Addition of visualisations, including a representation of the visualisation at each stage of the evaluation of the equation within the tree view
- Implementation of OLE to allow equations to be embedded within Microsoft Office
- Implementation of a library to support output to TeX/LaTeX for typesetting

6. Conclusions

The scope of the project was well chosen as it allowed development of the core of the application without having to implement a large number of functions and operators. The application is very extensible allowing the addition of new data types, functions and operators in the future. The user testing performed showed that the application behaves intuitively to new users, and allows rapid editing for more experienced users.

Acknowledgements

The author would like to thank the supervisor Dr. Burkhard Wuensche for providing the support and guidelines to make this project successful. The author would also like to thank his partner Andrew Brown for the work he has done on the project.

7. References

[1] Associate Professor Robert Amor, "lecture01 Introduction to HCI" [online document] 2005 [cited 2005 September 9] Available HTTP: https://www.se.auckland.ac.nz/courses/SOFTENG450/lectures/lecture01%20Introduction%20to%20HCI.pdf

[2] Design Science, Inc., "Design Science: MathType vs Equation Editor" [online document] 2005 [cited 2005 September 9] Available HTTP: http://www.dessci.com/en/products/mathtype/mt_vs_ee.htm

[3] MathWorks, "The MathWorks - MATLAB® - The Language of Technical Computing" [online document] 2005 [cited 2005 September 9] Available HTTP: http://www.mathworks.com/products/matlab/

[4] Mathsoft, “Mathcad” [online document] 2005 [cited 2005 September 9] Available HTTP: http://www.Mathcad.com/products/Mathcad12/