ViSSh: A Data Visualisation Spreadsheet

Fabian Nuñez and Edwin Blake

Collaborative Visual Computing Laboratory
Department of Computer Science, University of Cape Town
Private Bag, Rondebosch 7701, South Africa
{fabian,edwin}@cs.uct.ac.za

Abstract. We describe a data visualisation system which uses spreadsheets as its user interface metaphor. Similar systems implemented in the past were hampered by the contradiction between an imperative formula language and the declarative spreadsheet framework. We have analysed spreadsheets from a data visualisation point of view, and built a system that is an improvement over past efforts. Our prototype combines the following three techniques: we store lists of values in each spreadsheet cell; we use a functional programming language as the formula language and we make use of lazy evaluation. The novel combination of these techniques makes our system consistently declarative in nature, and gives it several advantages such as small, uncluttered visual programs, the ability to deal with potentially infinite datasets and the use of advanced functional language features.

1 Introduction

Our prototype system for data visualization "ViSSh" (*V*isualisation *S*pread*Sh*eet) uses a purely declarative spreadsheet paradigm for data visualization. ViSSh has been designed to explore the benefits to the user of a consistent application of a pure declarative programming paradigm. We note that spreadsheets are in essence declarative (since cells store either constants or side-effect-free formulas). One can also observe that the traditional programming systems for visualization (Modular Visualisation Environments or MVEs [2]) which use a visual data flow programming model also have a declarative flavour – data flow leads to pure declarative formalism.

A further insight on which our approach is based is that a spreadsheet is, of itself, a data visualization of sorts and as such its use in data visualization is a consistent extension of the basic spreadsheet paradigm.

Spreadsheets possess a number of useful properties for data visualization, *viz.*:

A. They are declarative in that cells store constants or functions without side-effects.
B. They are naturally tidy and uncluttered no matter how large they grow, while the same cannot be said for data flow graphs.
C. They derive a lot of power from their mutability: changes are easily made since all cells are available for editing.

However, they suffer from a number of drawbacks, mainly related to the way they scale with the size of the problem:

1. Large spreadsheets may be tidy but one can only view a small part of them.
2. Very large data sets are unwieldy since each cell may only contain a single value.
3. The large size of datasets commonly visualised would slow down recalculations.

We have developed extensions to the basic spreadsheet paradigm that overcome the above while retaining a pure declarative approach, and implemented them to demonstrate their viability. In essence, our extension of the spreadsheet paradigm consists of the following: storing lists of values in each cell, using a functional programming language and making use of lazy evaluation. Our prototype implements these extensions, as well as addressing some less fundamental usability problems we came across during its development. For point (1) above, we adopt an *overview window*, that acts as a map of the spreadsheet to aid navigation. Very large data sets (2) are naturally accommodated by using *lists* rather than scalars as the primitive data type in each cell and (3) is overcome by using lazy evaluation in the recalculation process. Sect. 5 further elaborates on these points. Additionally, we discovered that providing users with an *automated data flow view* of the corresponding spreadsheet (see Fig. 2) greatly helps users with the editing process. Since our spreadsheet is purely declarative there is a one-to-one correspondence between this dataflow view and the spreadsheet.

2 Previous Work

Most current data visualisation systems are based on the dataflow model. This includes systems such as *Iris Explorer*, *apE* and IBM's *Data Explorer* (see [2] for an overview). The dataflow paradigm is declarative in nature, and allows users to build non-trivial programs without extensive programming knowledge.

Spreadsheets have seldom been applied to data visualisation. Levoy [9] has described a system in which spreadsheet cells are populated not only by numbers and formulas (written in *Tcl*), but also by graphical objects (such as images or movies) and user interface objects (such as buttons or sliders). Levoy found that, compared to the commonly used dataflow model, spreadsheets are "more expressive, more scalable, and easier to program"[9]. However, Levoy's use of Tcl as a formula language leads to a mismatch in the programming paradigms used. Spreadsheets use a declarative programming paradigm, while Tcl is clearly imperative in nature. In addition to this, Levoy encourages the use of imperative programming and side effects (e.g., having cells directly modify other cells). Chi, Riedl, Barry and Konstan [3] extend the work of Levoy by not constraining the system to images, but instead allowing the user to visualise any type of information. Although this work does remove some of the limitations in Levoy's implementation, it still suffers from the same paradigm mismatch since it also uses Tcl.

The functional programming paradigm stipulates that functions may only read their arguments and generate only the result, without making any changes to their surroundings. Spreadsheet formulas on most commercial spreadsheets normally adhere to this paradigm, if one considers the formula to be the function, the cells referred to by the formula as the function's arguments and the value displayed in the cell occupied by the formula as the function's result. De Hoon [4] has shown that a spreadsheet can indeed be built from a purely functional perspective, and describes his implementation using the functional programming language *Clean*.

Since traditional spreadsheets make use of declarative formula languages, they do not suffer from the paradigm mismatch described above. However, it was not until the functional paradigm was used in conjunction with spreadsheets that the compatibility of spreadsheets and the functional paradigm was demonstrated. De Hoon [4] has constructed a text-based spreadsheet, which uses the functional language *Clean* [1] as its underlying formula language. This spreadsheet has some interesting properties, such as the ability to evaluate functional expressions symbolically. This depends on the choice of functional language, but does demonstrate the compatibility of both paradigms.

3 An Abstract View of Spreadsheets

Isakowitz *et al* [8] have analysed spreadsheets using concepts from the field of databases, and have come up with the concept of a *dual* view of a spreadsheet. On one side, there is the *physical* layout. This refers to the way cells are laid out, grouped, *et cetera*. On the other side there is a *logical* view of the spreadsheet. This handles abstract aspects of the spreadsheet, such as data dependencies between cells. The logical view could be likened to a database's schema, and in fact performs many of the same duties. Using an algorithm described in [8], one can break up a spreadsheet into several properties, called *schema*, *data*, *editorial* and *binding*. These four properties, taken as a whole, make up the entire spreadsheet, i.e. *spreadsheet* = *schema* + *data* + *editorial* + *binding.* These properties are defined as follows:

- The *schema* property stores a formal definition of the spreadsheet's formulas.
- The *data* property is the set of constants on which the *schema* property operates.
- The *editorial* property is what remains in the spreadsheet after the *schema* and *data* properties have been extracted: labels, comments, etc.
- The *binding* property is what defines the logical to physical mapping of the other three properties, using row and column addresses.

The logical structure of a spreadsheet is defined by the *schema* and *data* properties, while its physical structure is described by the *editorial* and *binding* properties.

It can be seen that Isakowitz's *schema* property can be likened to a dataflow system, since it keeps track of the dependencies between spreadsheet cells, and hence indicates the flow of data between these cells.

3.1 A Functional Look at the Logical Properties

Isakowitz and Schocken separate the concepts of *schema* and *data*, suggesting that functions and the data they operate on are different. Most functional languages, however, treat functions and data as equivalent entities. A re-examination of the logical structure of a spreadsheet, this time from a functional point of view, may reveal some interesting properties of spreadsheets.

In spreadsheets, cells can contain either constant values or formulas. A formula can be easily seen to be simply a different representation of a function. Any constant can also be trivially expressed in term of functions, simply by defining a function that always returns the same constant. Therefore it is possible to describe the contents of a spreadsheet cell entirely in terms of functions. From this, it follows that a spreadsheet can be seen as simply a grid of functions. For the sake of simplicity, each could in turn be called A1, A2, A3, . . . , B1, B2, B3, . . . , etc. If one considers a spreadsheet which contained a “`1`” in cell B1, a “`2`” in cell B2 and the formula “`=SUM(B1:B2)`” in cell B3, then these functions could be written (using the Scheme language syntax) as follows:

```
(define (B1) 1)
(define (B2) 2)
(define (B3) (+ (B1) (B2)))
```

The implications of this observation are not immediately obvious, yet they are quite fundamental. Since a spreadsheet can be completely defined in terms of named functions, it can be adequately described only in terms of functions.

Of course, some information is lost in the process, namely the spatial relationships between the cells. These relationships are used in two ways, namely to make relative references to cells (e.g., “add the contents of the cell just above this one to the contents of the cell just to the left of it”), and to manipulate groups of cells as ranges. However, this functionality can be implemented in other ways, for instance, consider the following implementations of relative references and cell ranges. Note that both of these are used only when *editing* the spreadsheet and do not affect *computations* in any way.

Relative References. Relative references can be implemented by applying simple transformations to the formulas as they are moved or copied to other cells. For example, a formula such as `=sum(A1:A3)` would become `=sum(B1:B3)` if copied one cell to the right of its current location. A mechanism would be required to “anchor” some cell references so they are not transformed (for example, the cell reference `$B$4` in *Microsoft Excel* does not change when the cell referring to cell B4 is moved). This functionality would not be a part of the functional part of the spreadsheet, but of the cell editor.

Cell Ranges. Cell ranges can be implemented as compound types. For example, the cell range named `B2:B4` could be expressed as the Scheme list `(B2 B3 B4)`. Again, since this relies on cell adjacency, the responsibility lies with the cell editor.

4 A Redefinition of Spreadsheets

In Sect. 3.1 it was observed that a spreadsheet can be described as a rectangular grid of functions, instead of a mixture of functions and data. Therefore the logical view of a spreadsheet referred to by Isakowitz and Schocken [8] need not be broken down into *schema* and *data*, but can in fact be described as a coherent whole.

Since all inter-cell references are calls to named functions, spreadsheets have no direct need for a grid structure, and the grid organization of spreadsheet cells thus comes into question. It seems then that a spreadsheet can be described simply as a set of functions that are ordered on a rectangular grid. However, it should be remembered that the names that are given to functions are completely arbitrary (as long as they are consistent). This means that the grid layout of spreadsheets is in fact a user interface feature, and is not necessary for the spreadsheet’s functioning. Note that this remark is not meant to demean the importance of the grid, but instead to highlight the separation that exists between the user interface and the underlying functional mechanism.

Isakowitz and Schocken's logical view of a spreadsheet has already been explained as a set of functions, while the physical view (i.e. the layout of the cells) has also been shown to be separate from the underlying functional structure. Therefore, since the physical layout is separate from the "real" structure, yet aids in its comprehension and manipulation, the physical layout itself can be classed as a form of data visualisation. Therefore, a spreadsheet can be defined as being a visualization of a finite set of functions, which taken as a whole solve a given problem. In other words, *a spreadsheet is an interactive system for manipulating and visualising declarative functions.*

This definition is quite useful, since it implies that since spreadsheets are already (simple) data visualisations, all that is needed to build a complex, interactive data visualisation based on spreadsheets is merely an extension of the basic spreadsheet paradigm.

5 The Extended Paradigm

We have extended the spreadsheet paradigm to address the needs of data visualisation systems, while maintaining the simplicity and declarative nature of traditional spreadsheets. Although some of the techniques described have been used individually in the past (e.g. Levoy's *Spreadsheets for Images* [9] and Eriksson's *Scheme in a Grid* [5]), by combining all of these features we have made an improved system.

Most programming systems deal with the concept of processing multiple data items by using ideas such as recursion or iteration, i.e. taking the elements of a set and processing them one after the other. While these methods have many advantages, they are ill-suited to the spreadsheet paradigm, where looping constructs cannot be elegantly expressed. Instead, numbers in traditional spreadsheets are manipulated individually, and where some form of grouping is desired, cell ranges can easily be formed by the user, and operations such as finding the average of a given set of numbers can be accomplished by idioms such as `=average(B12:B72)`. Although this works quite well for financial statements, data visualisations normally deal with data sets containing tens of thousands of items. At this level, the range paradigm breaks down due to the unwieldiness of thousands of rows or columns containing thousands of items.

Our extended spreadsheet paradigm circumvents this problem by extending the spreadsheet paradigm to allow sets of items (in the form of lists) to be stored in each cell, as opposed to single items. Most non-circular data structures can be implemented using only lists; many programming languages provide lists as their only form of data abstraction. The concept of ranges is no longer needed – every single cell *is* a range.

In Sect. 4 we showed that a spreadsheet can be seen as an environment for the development of functional programs. Therefore the basic paradigm would not be altered if, instead of the formula language normally associated with spreadsheets, a full-featured functional language were used (since spreadsheet formulas do not have any side effects, they can be considered as a special-purpose functional language). By using a functional language the user benefits from its generality and extensibility, as opposed to being constrained by the primitives provided by the spreadsheet software.

Part of data visualisation is to eliminate information from a dataset, in order to obtain the underlying patterns in the data. Therefore, some data often does not make it all the way from the database to the screen. In those cases it would be useful to know what values are not going to be needed by later calculations, in order to save time by simply not calculating them. *Lazy evaluation* defers calculations until they are needed – for example, a lazy evaluation of *f(g(x))* will pass *"g(x)"* to function *f*, instead of evaluating *g(x)* and passing *f* the result; it will be *f*'s responsibility to evaluate *g(x)*. If for some reason *f* does not need *g*'s result, that value will not be calculated. Therefore, if datasets are processed using lazy evaluation, we will have the ability to deal with large datasets in an efficient manner (e.g., if a bitmap is generated and then scaled down to 25% of its size, only one out of every four pixels will actually be generated). However, this must be transparent to the user; recalculations must appear to happen as before.

To summarise, we have extended the spreadsheet paradigm by the addition of three techniques: storing lists of values in each cell, using a functional programming language and the use of lazy evaluation.

6 The Prototype

To test our ideas we have built a prototype, called ViSSh (see Fig. 1). A ViSSh spreadsheet consists of a set of typed cells, each of which is represented on the spreadsheet grid by a small user interface. Each type of cell performs a different function, taking as input a list of values and returning another list, the contents of which functionally depend on the input list and the type of the cell.

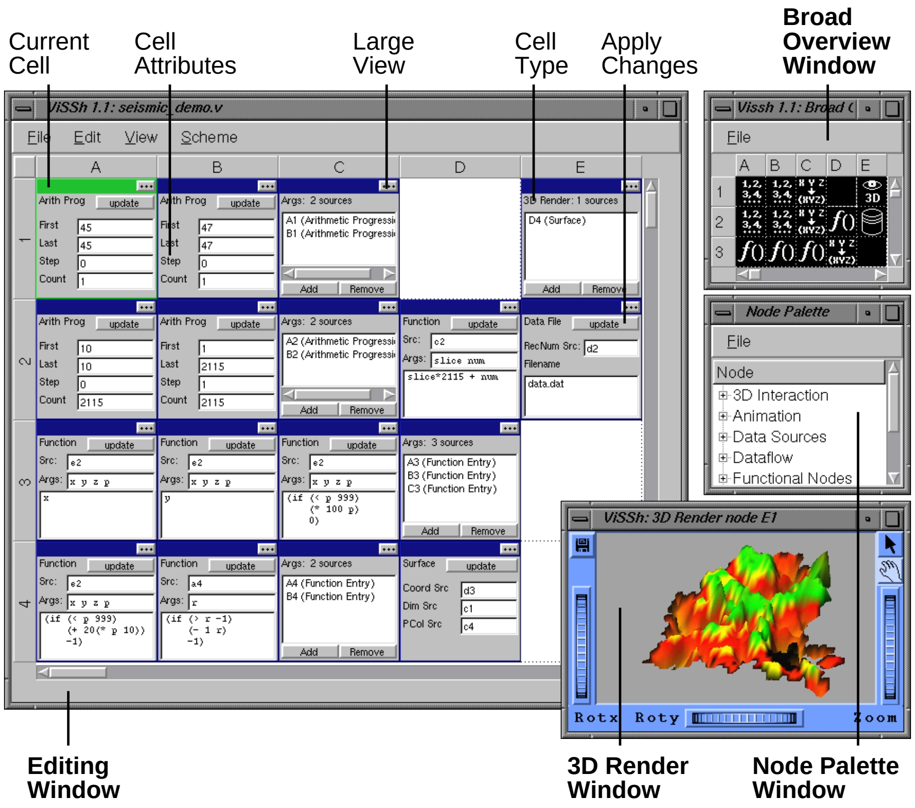
ViSSh attempts to provide users with a useful data visualisation system, using spreadsheets as its user interface metaphor. It uses the functional language Scheme (a dialect of the functional language Lisp) as its formula language.

6.1 Dealing with Large Datasets

The ability to deal with large datasets is crucial in a viable data visualisation system. ViSSh implements the extended spreadsheet paradigm described in Sect. 5, and as such is capable of handling potentially infinite datasets. Unlike Levoy's "Spreadsheets for Images" [9], ViSSh does not display the contents of each cell. This is both because of the potentially huge volumes of data (making the traditional "cells display their own contents" technique impractical in the general case), and because there is no single "right" way of representing an arbitrary data set stored in a cell. Instead, there are specialised cells that are used to view data in different ways. For example, in Fig. 1 the contents of cell D4 (a 3D surface) are displayed by cell E1 (a 3D renderer).

Functional Programming Model Whereas most spreadsheets use a fairly rudimentary language to express their formulas in, ViSSh uses the programming language Scheme. This gives users a lot more expressive power (as well as allowing the use of data abstractions, see Sect. 5), and allows extension of the system by writing new functions in Scheme. These are dynamically loaded into the spreadsheet at runtime and can be used by spreadsheet cells. To enhance usability we have extended the parser to allow expressions in infix form, as opposed to the native Scheme (e.g., `m*x+c` or `(+ (* m x) c)`).

One of the advantages of a functional programming system is its enhanced modularity [7], which from a user's point of view translates into a system that easily allows



	A	B	C	D	E
1	Arith Prog First 45 Last 45 Step 0 Count 1	Arith Prog First 47 Last 47 Step 0 Count 1	Args: 2 sources A1 (Arithmetic Progressi B1 (Arithmetic Progressi		3D Render: 1 sources D4 (Surface)
2	Arith Prog First 10 Last 10 Step 0 Count 2115	Arith Prog First 1 Last 2115 Step 1 Count 2115	Args: 2 sources A2 (Arithmetic Progressi B2 (Arithmetic Progressi	Function Src: c2 Args: slice num slice*2115 + num	Data File RecNum Src: d2 Filename data.dat
3	Function Src: e2 Args: x y z p x	Function Src: e2 Args: x y z p y	Function Src: e2 Args: x y z p (if (< p 999) (* 100 p) 0)	Args: 3 sources A3 (Function Entry) B3 (Function Entry) C3 (Function Entry)	
4	Function Src: e2 Args: x y z p (if (< p 999) (+ 20(* p 10)) -1)	Function Src: a4 Args: r (if (> r -1) (- 1 r) -1)	Args: 2 sources A4 (Function Entry) B4 (Function Entry)	Surface Coord Src d3 Dim Src c1 PCol Src c4	

Fig. 1. This illustrates ViSSh, our Data Visualisation Spreadsheet. The spreadsheet depicted here generates and displays an interactive 3D representation of seismic disturbance data read from a database. See Sect. 7 for a brief description of its operation.

code reuse. In many cases data visualisation benefits greatly by code reuse, and most data visualisation systems allow this practice by the use of some form of function libraries. ViSSh allows several spreadsheets to be linked together in a functional manner. This is done by the use of three specialised spreadsheet cells which, as a group, implement the idea of function calls in the context of spreadsheets. The user's view of functional linkage between two spreadsheets is quite simple: The *called* spreadsheet has an "Argument" cell, which collects the arguments that the function implemented by the spreadsheet takes (recall that each cell generates a list). It also has a single "Result" cell, which exposes the final result calculated by the spreadsheet. The *caller* spreadsheet has a "Subsheet" cell which behaves much like a function evaluation cell, but which "calls" the helper spreadsheet by evaluating its Result cell. This causes the normal evaluation process to propagate through the helper spreadsheet, until the Argument cell gets evaluated. This cell pulls in the arguments passed from the master spreadsheet, and the results then propagate forward until they reach the Result cell. The results then are given to the SubSheet cell, which returns them as its own results.

Navigational Aids A common problem with spreadsheets is that users can become "lost" when navigating through large spreadsheets which contain many similar subsections. This problem is especially acute in the case of ViSSh, since each cell is rather large and hence the number of them that can be displayed at any one time is fairly small (typically 7×5 cells in a 1024×768 display). This problem is tackled using two different tools, one for each aspect of the problem: grid navigation and dataflow.

Grid Navigation. To find out which part of the spreadsheet grid is being viewed though the editing window, as well as locating the current cell cluster in relation to the rest of the spreadsheet, the user can call up a window which contains a miniature version of the spreadsheet (See Fig. 1). Each cell contains a small (32×32 pixel) icon describing the function of the cell in the larger spreadsheet, wherein the area of the larger spreadsheet that is visible at any time is shaded in the small-scale spreadsheet. Since this can display a much larger portion of the spreadsheet (about 30×20 cells in a 1024×768 display), the spreadsheet navigation problem is greatly reduced by the use of this window. It can be scrolled independently of the main spreadsheet window, and clicking on one of the cells in the smaller window scrolls the main spreadsheet so that the corresponding cell becomes visible. The "Broad Overview" window acts as a road-map, with the icons representing the function of each cell being used as landmarks.

Dataflow. The other aspect of spreadsheets that can cause much frustration is keeping track of which cells depend on what other cells. ViSSh solves this problem by providing the user with a window which contains a dataflow diagram in which the nodes are iconic representations of the spreadsheet cells (see Fig. 2). The same icons that are used in the "Broad Overview" window to describe spreadsheet cells are used by this window. Like the "Broad Overview" window, the dataflow window can be used to navigate around the main spreadsheet window by clicking on a cell icon; the dataflow diagram is automatically kept synchronised to the main window. This dataflow diagram provides users with the advantages of the dataflow paradigm, while the spreadsheet editing environment shields them from the cluttering associated with medium to large dataflow editing environments.

7 A Practical Example

During its development, ViSSh has been tested with a variety of data sets. One of these consists of seismic disturbance data, recorded from locations in Southern Africa. Figure 1 illustrates the spreadsheet used to visualise this data. The flow of data through the spreadsheet is shown in Fig. 2. The cells in this spreadsheet can be classified into 3 functional groups, namely: extracting data from a file; generating a 3D surface with this data and displaying the 3D surface. The cells tasked with extracting the data are those in the range A2:E2. Cells A2:D2 generate record references (record-field pairs), which are used to extract data from the database by cell E2. The creation of the 3D surface is handled by two groups of cells. Cells A3:D3 form the 3D mesh using the data read in by cell E2, while cells A1, B1, A4, B4, C1 and C4 generate the vertex colours used by the surface, based on availability of data (the dataset has a number of "missing" data points, which must not be rendered) and the displacement value stored in the database. Rendering of the 3D surface is performed by cell E1.

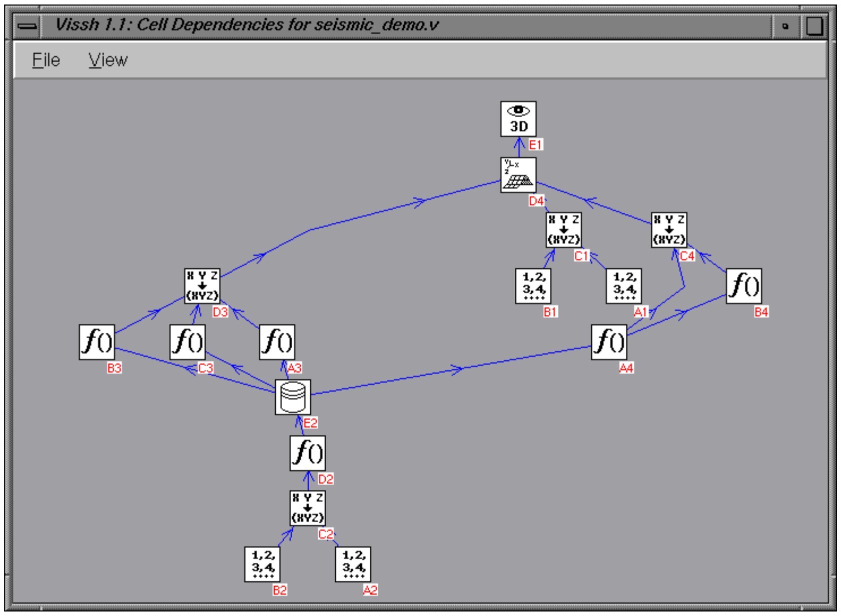


Fig. 2. This illustrates the intercell dependencies of the spreadsheet in Fig. 1. Each cell is represented by a small icon, using the same imagery as the "Broad Overview" window in Fig. 1. The arrows indicate the flow of data; clicking on an icon will make the corresponding cell be the current cell in the main spreadsheet window.

8 User Experiences

ViSSh is being currently tested by the authors and others. This testing has involved visualisations of the effects of different ATM network routing algorithms, as well as seismic disturbance data as described above. We have also compared ViSSh to Khoral Research's *Cantata*, and found that ViSSh was generally easier to use. We believe this is mainly because the spreadsheet layout made the visual program less cluttered than Cantata's dataflow model, thus easing the editing process. Additionally, the Scheme programming language made the formulas, especially those involving conditionals, compact and concise. Using the "Broad overview" and Dependencies windows (see Sect. 6.1 for details) also eased the visualisation task. The latter gives users the advantages associated with dataflow diagrams (see Fig. 2).

In future, in addition to doing further user testing, we will analyse ViSSh using Green's *Cognitive Dimensions Framework* [6]. This will give us a means with which to objectively compare ViSSh to other data visualisation systems.

9 Conclusion

In this paper we have looked at various attempts at using the spreadsheet paradigm for data visualisation. We have noted where this paradigm fails for data visualisation, and extended the paradigm by adding the following: firstly, each spreadsheet cell does not contain a single value, but a list of values. This allows us to store thousands of values in a spreadsheet, while keeping its size manageable. Secondly, we have used a full-featured functional programming language as the formula language. Using this instead of the formula languages used in most commercial spreadsheets provides the user with more expressive power, even allowing constructs such as high-order functions which are not expressible with most imperative languages. Finally, the lists of values are manipulated using lazy evaluation, which allows for the efficient manipulation of potentially infinite lists. In addition to this, we have added navigational and dataflow debugging facilities to circumvent problems inherent in the spreadsheet paradigm. Although others have implemented these separately, we believe that our main contribution is the combination of the three, together with the underlying theoretical framework for doing so. We have built a prototype to test the extended spreadsheet paradigm, and demonstrated its use for real-world data visualisation tasks. During our testing, we found that ViSSh spreadsheets were easier to edit than dataflow diagrams, mostly due to the absence of the clutter that normally occurs when dataflow diagrams are frequently modified. This result agrees with Levoy's findings regarding spreadsheet versus dataflow user interfaces [9]. Additionally, our dataflow debugging aids allow the transfer of skills from existing Modular Visualisation Environment (MVE) users.

10 Acknowledgments

We would like to thank Dr. Gary Marsden for his help and encouraging advice in the writing of this paper, Dennis Burford for the Kaap Vaal Seismic Experiment data and the South African National Research Foundation for their support.

References

1. T. Brus, M. van Eekelen, M. van Leer, M. Plasmeijer, and H. Barendregt. Clean - a language for functional graph rewriting. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*. Springer, 1987.
2. G. Cameron. Special focus: Modular visualization environments (mves). *Computer Graphics*, 29(2), May 1995.
3. E. H. Chi, P. Barry, J. Riedl, and J. Konstan. Principles for information visualization spreadsheets. *IEEE Computer Graphics and Applications*, 18(4):30–38, July/August 1998.
4. W. de Hoon, L. Rutten, and M. van Eekelen. Implementing a functional spreadsheet in clean. *Journal of Functional Programming*, 5(3):383–414, July 1995.
5. U. Eriksson. Scheme in a grid. Online HTML document, 1999. `"http://siag.nu/siag/"`.
6. T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.
7. J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.
8. T. Isakowitz, S. Schocken, and H. C. Lucas. Toward a logical/physical theory of spreadsheet modeling. *ACM Transactions on Information Systems*, 13(1):1–37, 1995.
9. M. Levoy. Spreadsheets for images. In *Computer Graphics Proceedings*, Annual Conference Series, pages 139–146. ACM SIGGRAPH, July 1994.