Journal of Universal Computer Science, vol. 13, no. 6 (2007), 806-816
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © *J.UCS*

Programming through Spreadsheets and Tabular Abstractions

Carlos Henrique Q. Forster
(Instituto Tecnológico de Aeronáutica, Brazil
forster@ita.br)

Abstract: The spreadsheet metaphor has, over the years, proved itself valuable for the definition and use of computations by non-programmers. However, the computation model adopted in commercial spreadsheets is still limited to non-recursive computations and lacks abstraction mechanisms that would provide modularization and better reuse (beyond copy and paste). We investigate these problems by identifying a minimal set of requirements for recursive computations, designing a spreadsheet-based language with an abstraction definition mechanism, prototyping an interpreter and evaluating it with examples.

Keywords: Spreadsheet languages, End-user programming, Call-by-need, Lazy structures, Recursion, Abstraction.
Categories: D.1.1, D.1.7, D.2.6, D.3.3, F.1.1, H.1.2

1 Introduction

Spreadsheets are a popular document model for structuring information and interacting with data. With spreadsheets a user can also define small computations with data that allow for instance statistical summarization, simulation and information highlighting provided that this user is able to express these computations in a formula language. Spreadsheet systems have also been used as a data entry interface for other software.

While the textual paradigm of programming daunts the general user, we can see non-programmers using spreadsheets and defining simple computations expressed through spreadsheet formulae. This can be a good starting point for learning to program. A spreadsheet user will perceive desired computations that he is unable to describe in a spreadsheet and feel the urge to move to a textual programming language. It is expected that enhancements on the expressive power of spreadsheets to allow advanced computations would fill the gap between spreadsheet and textual programming.

One criticism about current commercial spreadsheets is their lack of abstraction mechanisms. Users are limited to a set of predefined functions. Modularization is simulated by copy-and-paste of template spreadsheets. This makes spreadsheets hard to maintain. Code replication forces the users to keep track of all copies in order to update them properly when new information arrives or a bug is found. Proper reuse mechanism would allow tested and trustful components to be safely adopted and easily replaced. Additionally, some features can be implemented through spreadsheets but in a very cumbersome way. Long expressions referring to many cells are truly hard to decipher and maintain. Having means to define abstract operations and

creating instances of these operations as needed would make complex spreadsheets more readable just like subroutine calls make programs more readable.

Another criticism is the lack of scalability. Although the supposition of infinite-sized problems can strike us as superfluous, recursive computations are important when we deal with abstractions. Most of the times when the problem is bigger than the spreadsheet were prepared to handle, we need just to insert a number of columns or rows and get a new spreadsheet ready to do the calculation. In the case of abstract spreadsheet definition, nothing should be assumed about the size of the problem, we just need one abstraction to be able to solve the problem and we cannot manually create another spreadsheet instance for each "subroutine call" because the number of instances can grow rapidly.

This paper investigates the problem of defining recursive and abstract computations in the tabular format, preserving important properties such as the spatial structuring of information and referential transparency. Usually an imperative scripting language is included in a spreadsheet system to add support to advanced computations. We intend to follow the reverse path, allowing complex computations to be defined in spreadsheet format and turning the spreadsheet into an environment for scripting application behavior or connecting software components.

We point some application areas for spreadsheet programming. Spreadsheet programming can be used for the reconfiguration of software applications through defining small computations or data flow among software components. Particularly, Information Visualization is an area that would benefit from the reconfiguration capability and the representation of data and data transformation in the tabular format. Also, in Education, teaching of Math-related subjects could benefit from end-user definition of computations [Misner and Cooney 1991].

The paper is organized as follows. [Section 2] of the paper revisits many concepts on which spreadsheets are based and points to a few references from the literature. The respective design decisions for the proposed system are presented in [section 3]. The development of our contribution starts with the identification of a set of primitive constructs that allow a spreadsheet language to represent recursive computations [section 4]. A model of abstraction definition in the tabular format for the spreadsheet language is presented [section 5]. The prototype implementation of an interpreter for the defined language and a spreadsheet editor are described [section 6]. The prototype system is evaluated by implementing some examples [section 7]. In [section 8], we provide a general conclusion of the present work and point future directions.

2 Design Issues Peculiar to Spreadsheet Languages

The design of a language for spreadsheets is quite different from the design of a general programming language being the main differences, the presence of a visual editor and the requirement for fast updates for interactive computing. Another important difference is the organization of information in a tabular structure. We describe in this section some design issues that are considered in our design.

Coordinate-based references are the foundation of the spatial organization of spreadsheets, but imply some design difficulties. On one hand, coordinates can be easily entered through point-and-click in the editor interface. Additionally, coordinate-based formulae can be copied, moved and extended (copied to a larger

area) maintaining appropriate relative references to, for instance, “the same line”, “the line above” or “two columns to the right”. On the other hand, formulae based on coordinate mnemonics are very difficult to read. Consequently, programs are difficult to write or maintain. Verifying correctness of choice between absolute, relative or mixed-coordinate references is pretty difficult. In Visicalc-based spreadsheets [Bricklin and Frankston 1999], absolute coordinates represented such as $A$1, when copied or extended are kept as $A$1. Relative coordinates like A1, when extended to the right becomes B1, when copied to the cell immediately below becomes A2. Mixed coordinates are represented like A$1 or $A1.

The semantics of cell copy, movement and extension operations is a source of confusion. The behavior of those operations can vary concerning how cell references are maintained. Some systems may update cells that used to point to a previous location of another cell. However, copying would be handled differently. In some editors, movement keeps track of which cells refer to the cell being moved. It is confusing because it can modify something very far away from the area where the user is changing. [Lisper and Malmström 2002] point at the problem of concatenation of tables on the same page. If we concatenate two tables side-by-side, the insertion of a new row to the first table would spoil the alignment of the second table. In order to cope with row or column insertion concatenation of separate tables must be diagonal, what is logically correct but functionally and visually inappropriate. This is a good clue that data and programs must be organized beyond two-coordinate axes. A data structure holding multiple tables should be necessary.

Another way to reference cells considers cell names instead of coordinates. The use of names is recommended because references become more readable and provide a form of access similar to tuple structures. Systems based only on names, like dataflow systems, unfortunately rarely make use of the tabular spatial structure and can easily become cluttered both in name space and canvas space. Notable exceptions include Forms/3 [Burnett et al. 2001] and Haxcel [Lisper and Malmström 2002].

While, for the textual programming languages, there is the read-eval-print loop of an interpreter or the compile-eval-print loop of compiler systems, spreadsheets would have a definition-eval-display loop. Traditional spreadsheet systems would evaluate an expression and all of its dependencies on any change. A call-by-need spreadsheet system would evaluate only formulae that are directly or indirectly needed for the current visualization. The formula language of a spreadsheet doesn’t have the attribution operation. The absence of side-effects due to attribution provides the property of referential transparency. This is the property that two identical expressions yield identical values. The preservation of this property along with call-by-need will allow lazy-evaluation, mechanism of evaluation that goes beyond call-by-need by reusing computations so that identical expressions should not be evaluated twice. (See for example [Harrison and Field 1988]).

Recursive formulae through the means of cycles are not generally allowed. A spreadsheet system may permit the definition of cyclic dependent cell formulae. This will constitute an equation system or a constraint system, which may be solved by a constraint solver or through relaxation, where each cell is computed multiple times until some convergence criterion is satisfied or recognized as unreachable.

Some spreadsheet systems allow data structures to be included inside cells or to include spreadsheets as members of data structures such as a folders (list with

hierarchy) or trees. Complex objects such as graphics or images are allowed as cell values in some spreadsheets such as image spreadsheets [Levoy 1994] or visualization spreadsheets [Chi et al. 1998] [Nuñez 2002].

3 Design Decisions

We describe now our design decisions. In order to keep semantics as clear and simple as possible, we recommend using coordinate-awareness providing functions namely row() and col(). Such functions allow the formula of cell to know the coordinates of that cell. We propose that relative cell references be built upon these functions. Copying a formula from one cell to another is the correct cell copy operation. Extension of a cell to an area is implemented simply as copying the formula of the cell to each cell in the area. Alternatively, names are also allowed as absolute references.

Our model of spreadsheet editing and computation assumes that cells only accept formulae as input. We assume also that spreadsheet formulae cannot modify cells, only the editor is able to modify the formula of a cell. We define an editor as the only type of entity able to modify cells so that interactive computation should be obtained through the mediation of an editor. Complex objects may be represented or abstracted as spreadsheet data and it is the role of an editor to provide the correct visualization for this data.

We consider that evaluation of complex computations may not be deliverable in time, so our proposed editor has two modes: a formula-viewing mode and a value-viewing mode. Formulae are evaluated only when their values must be displayed (or are referenced by some other cell or evaluation chain whose value will be displayed). In formula-viewing mode, values are not displayed and then evaluation does not happen. This behavior of performing only the presently needed computations corresponds to the desired call-by-need behavior. Sharing the evaluation of expressions is implemented through caching of computed values of cells while intermediate values passed as function arguments are not cached and their resources are freed by a garbage collector. This behavior approaches lazy evaluation that could be properly obtained if graph rewriting was implemented, which was used for example by [de Hoon et al. 1995].

Spreadsheets are first-class objects and can be contained in spreadsheet cells. The cell reference operation is extended to provide access to cells of inner spreadsheets and outer spreadsheets. The coordinate-awareness functions can provide not only the coordinates of the cell being evaluated, but the location where its spreadsheet is stored and so on until a top-level spreadsheet is reached.

At the moment, cyclical references are not allowed and, when detected, are reported with an error symbol being delivered as the value of the cell. We provide other forms to describe recursive computations that we find more appropriate and are described in the next sections.

4 A Sufficient Model of Computation through Spreadsheets

We describe a computational model for spreadsheets. This model was created based on an experience of implementing a Turing Machine in a spreadsheet language. Therefore, the constructed model is expressive enough to represent any computable function. However, this sufficiency does not help to make program development easy. Readability of code, clear intuitive semantics and other human factors should be considered in the design of a spreadsheet language.

Coordinate cell references are important to exploit the spatial structure of information. Lets make all cell references through the use of a function ref(i,j) where i and j are respectively row and column numbers. So an absolute reference such as $B$1 is written as ref(1,2). An editor can adopt a syntax sugar of seeing or accepting ref(1,2) as $B$1. Now we consider coordinate-awareness constants i and j for which row and column numbers are automatically assigned. A cell self-reference would be ref(i,j), while a reference to the immediately above cell would be ref(i-1,j) and the cell immediately to the right is ref(i,j+1). If current cell is C5, ref(i,j+1) can be displayed by the editor as D5. If we enter D5, the editor converts it to ref(i,j+1). If we enter C$4, it is converted to ref(4,j). When considering copying a cell with this kind of representation, there is nothing to change in the formula. The same happens when extending and moving. The semantics is then very clear and closer to traditional textual programming languages.

Our view of this kind of representation is to understand all cells as functions. While a spreadsheet S is a function mapping a (row,column) pair into a cell, a cell C is a function mapping (row,column) into a value. So, a cell with formula ref(i,j+1) holds indeed a function λij.ref(i,j+1) in lambda-calculus notation. The value of this cell is obtained applying this function to the cell coordinates such as λij.ref(i,j+1) row column. Approaching cell definition as a function of its location allows the preservation of referential transparency. Complaints that relative references would damage referential transparency are, therefore, not effective for this design.

Indirection is also accomplishable and important for the implementation of conditionals. Conditionals need not to be a primitive if we have indirection such as ref(ref(1,1),ref(1,2)). Implementation of a conditional can be seen in [Tab. 1].

ANS=IF(COND; THEN; ELSE) for example ANS=IF(A>B; A; B)		
	J=1	J=2
I=1	COND: A>B	ELSE: B
I=2	ANS: ref(ref(1,1)+1, 2)	THEN: A

Table 1: Implementation of conditional.

In the considered case, the comparison operator > returns 0 for FALSE or 1 for TRUE. This value can be added to a row or column index and used to select the proper cell. Implementation of conditional with indirection is similar to parameter selection in lambda-calculus. In lambda calculus, TRUE and FALSE are combinators that select the first or the second parameter.

We opt to use two types of recursive definitions. The first one is the "ellipsis" tabular recursion; the second one is function-call recursion which is defined in [section 5] under the name of tabular abstraction for which some examples are discussed in [section 7]. Tabular recursion allows construction of infinite tables and provides a means to specify induction. A three dots symbol (…) in a cell implies that this cell and all cells to the right of it have the same formula which is the formula of the cell immediately to the left of the current cell. The vertical counterpart is a three columns symbol (:::). A diagonal ellipsis would be represented by ***. See, for example, the implementation of factorial and Fibonacci functions in [Tab. 2].

ANS=FACTORIAL(N)				
	J=1	J=2	J=3	J=4
I=1	1	mul(ref(row(),sub(col(),1)), col())	…	
I=2	N:	ANS: ref(1,ref(N))		

	J=1
I=1	1
I=2	1
I=3	add(ref(sub(row(),1),1),ref(sub(row(),2),1))
I=4	:::

Table 2: Implementation of factorial and Fibonacci functions.

To illustrate that this model is sufficient to describe any computable function we implemented a Turing Machine, first in Excel, then in our prototype. In [Tab. 3] and [Tab. 4] there is an implementation of a TM that accepts the language $\{1^n 2^n \mid n \geq 0\}$. The symbol 5 is the blank symbol for the TM tape. Each row of spreadsheet (a) contains an instantaneous description of the TM, the first element is the machine state, the second is the head position (column number) and the remaining elements are tape symbols.

A Turing Machine requires an infinite-length tape and may enter an endless loop, requiring that the spreadsheet that simulates it be infinite in both axes. Infinite-length spreadsheets can be defined by the use of the ellipsis operator. Due to the call-by-need evaluation scheme, this does not lead to infinite computations because only the values that are needed for calculation or presentation are evaluated. Our spreadsheets can be seen as lazy structures.

Although, in the Turing Machine example, all computations are represented by the infinite-length spreadsheet, this doesn't mean that the computations are carried out. We will only see a computed value if displayed by an editor, so we have to "scroll down" the spreadsheet until we see the TM stopped. If the TM enters an endless loop we would theoretically need to scroll down forever. From this observation, we conclude that a function to force the evaluation of cells is needed. We propose a "scan" function that will traverse the spreadsheet forcing evaluations until a criterion is met. In the case of the TM, this function will traverse the spreadsheet until the TM stops. If TM never stops, "scan" should also theoretically never stop.

a)Instantaneous Descriptions

A (state)	B (head)	C (tape)...					
1	3	1	1	2	2	5	5
2	4	3	1	2	2	5	5
2	5	3	1	2	2	5	5
3	4	3	1	4	2	5	5
3	3	3	1	4	2	5	5
1	4	3	1	4	2	5	5
2	5	3	3	4	2	5	5
2	6	3	3	4	2	5	5
3	5	3	3	4	4	5	5
3	4	3	3	4	4	5	5
1	5	3	3	4	4	5	5
4	6	3	3	4	4	5	5
4	7	3	3	4	4	5	5
5	8	3	3	4	4	5	5
#TRUE	8	3	3	4	4	5	0

5 is the blank symbol

This machine accepts $\{1^n 2^n \mid n \geq 0\}$

b) Symbol to Write on Tape

symbol	*=1*	*=2*	*=3*	*=4*	*=5*
state=1	3			4	
state=2	1	4		4	
state=3	1		3	4	

c) State Transition Table

symbol	*=1*	*=2*	*=3*	*=4*	*=5*
state=1	2	#FALSE	#FALSE	4	#FALSE
state=2	2	3	#FALSE	2	#FALSE
state=3	3	#FALSE	1	3	#FALSE
state=4	#FALSE	#FALSE	#FALSE	4	5

d) Tape Head Movement

symbol	*=1*	*=2*	*=3*	*=4*	*=5*
state=1	1			1	
state=2	1	-1		1	
state=3	-1		1	-1	

e) Excel Formulae

A3=ÍNDICE(State!$A$1:$E$5;Tape!$A2;ÍNDICE($A2:$Z2;1;$B2))

B3=ÍNDICE(Move!$A$1:$E$5;Tape!$A2;ÍNDICE($A2:$Z2;1;$B2))+$B2

C3=SE(COL()=$B2;ÍNDICE(Write!$A$1:$E$5;Tape!$A2;
ÍNDICE($A2:$Z2;1;$B2));C2)

Table 3: This is an implementation of a Turing Machine in Excel. We suppose lines and columns are infinite. Three basic formulae are given in (e), additional formulae is produced by the extension mechanism.

	J=1	J=2	J=3	4	5	6	7	8
I=1	state:grid(…)	write:grid(…)	move:grid(…)					
I=2	1	3	1	1	2	2	5	…
I=3	ref(state, ref(sub(row(),1),1), ref(sub(row(),1), ref(sub(row(),1),2)))	add(ref(sub(row(),1),2), ref(move, ref(sub(row(),1),1), ref(sub(row(),1), ref(sub(row(),1),2))))	if(eq(col(), ref(sub(row(),1),2)), ref(write, ref(sub(row(),1),1), ref(sub(row(),1), ref(sub(row(),1),2))), ref(sub(row(),1),col()))	…				
I=4	:::	:::	:::	***				

Table 4: Implementation of a Turing Machine in our prototype.

5 A Model for Abstraction Definition

We propose an abstraction model very similar to the one in Forms/3 [Burnett et al. 2001]. The abstraction scheme resembles the delegation mechanism used as inheritance in prototype-based languages such as JavaScript. Every spreadsheet is a candidate for abstraction. It happens by overriding the formulae of some cells of a template spreadsheet and collecting the results of computed cells of the new formed spreadsheet.

We call tabular abstraction the use of a spreadsheet or table as a function. Our liberal proposal permits that any cells of a template spreadsheet be overridden. The override of a spreadsheet consists of a spreadsheet with the substitution cells and a reference to the template spreadsheet whose cells are being overridden. The new spreadsheet formed this way will have cells whose formula will evaluate based on the other overridden cells. We use the following syntax: extend(template_spreadsheet, override_spreadsheet) which evaluates to a spreadsheet that can be kept inside a cell. A spreadsheet can extend itself using the ref() function without arguments. In terms of lambda-calculus, we create an abstraction defining which cells become parameters and already apply the abstraction to values and formulae that override the cells.

The difference of our proposal to [Burnett et al. 2001] is that tabular organization is enforced and depended upon. Coordinate-awareness functions can provide access to the cells of the outer (or the "calling") spreadsheet as absolute references or references relative to the location of the extend() function. As a result, spreadsheet cells can be also overridden with ellipsis, allowing a sort of infinite argument list. Return of multiple values is done by accessing computed values of the spreadsheet instances. Infinite list of values can then also be returned.

6 Implementation

We implemented the prototype of the interpreter and the editor in Java. Java was chosen as development platform for several reasons: we already have a garbage collector; it is easier to write a graphical editor; it is easy to write a FFI to extend the system using class files and the reflection mechanism; it is multiplatform; it is free.

Spreadsheets were implemented as Java Hashtables. The objects used for keys are coordinate pairs and, for elements, a class containing a formula string, the expression tree of the parsed formula and a cache for the cell value. Each spreadsheet also has a hash table of named cells (mapping symbolic name to coordinates) and a linked list of ellipsis cells. When the value of a cell is requested, the spreadsheet checks if the cell value was cached. If it is not the case, then it checks if the cell is influenced by some ellipsis cell, returning the corresponding ellipsis reference or returning the cell expression tree otherwise. When any formula is changed, all cached values are discarded.

Spreadsheet persistence is made by a linear description of spreadsheets in the formula language. This allows also the inclusion of a spreadsheet as a cell element inside another spreadsheet. The syntactical construction is grid(row,column,formula,…). This construction is not evaluated as a function; its parameters are considered "quoted".

The editor has two modes: a formula-view mode and a value-view mode that can be switched through a button widget. Formulae for the cells can be entered in either mode. A cell is selected by a single click. A ctrl+click combination produces the expression of relative reference to the cell, while a ctrl+shift+click produces the absolute reference. Double-clicks on cells that contain a spreadsheet will open an editor window with that inner spreadsheet. The textual representation of a spreadsheet can be obtained by an option on the menu and can be pasted as the content of any spreadsheet cell.

The lexical analyzer was written with the aid of Java regular expressions. The parser is a top-down depth-first backtracking parser based on recursive function calls. A symbol table (based in a hash table) is kept to make quick comparisons of symbolic strings. The evaluation of some integer expressions was implemented: addition, multiplication, arithmetic negation, subtraction, division, comparison (equal, greater than etc.). In the case of a spreadsheet inside of another, an expression ref (0, 2, 7, 0) refers to the cell at row 7 column 0 of the spreadsheet located in the cell at row 0 column 2. This function can receive as arguments integer coordinates, names of cells and spreadsheets, so that the expression ref (ref (0, 2), 7, 0) has the same meaning of the one above. Functions col() and row() return the coordinates of the cell being evaluated and admit a numeric parameter n to refer to the n-th outer level coordinates. The function up() returns the spreadsheet that contains the spreadsheet with the cell being evaluated.

7 Examples and Evaluation

We implemented several experiments including Pascal triangle with tabular recursion [Fig. 1], functional-recursive Fibonacci and factorial functions [Fig. 2] and the already mentioned TM. The computation of Fibonacci is well-known as costly to compute when we not reuse previously computed values. We implemented efficient computation of Fibonacci series through value cache. The cache is limited to cell values and is accessed during the evaluation of the ref() function. These basic examples show the expressiveness of the language. The option to create windows to show inner spreadsheets for intermediary computations is illustrated in [Fig. 2] and was very important while constructing the spreadsheets.

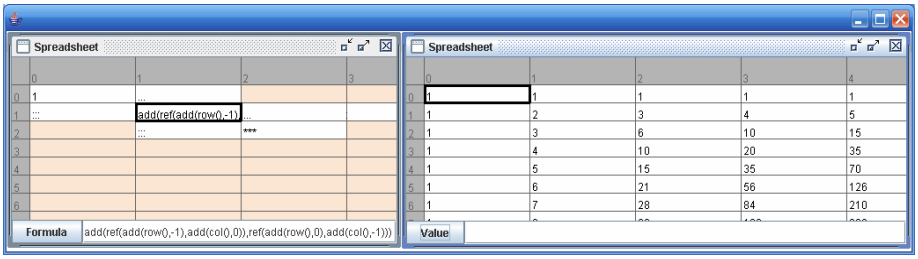

Figure 1: Pascal triangle spreadsheet.

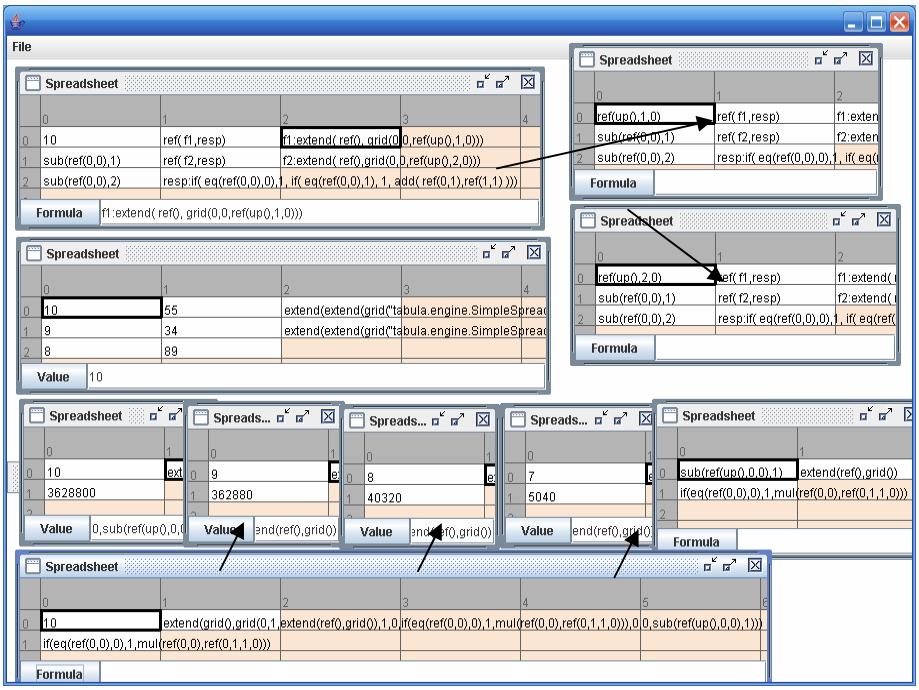

Figure 2: Recursive factorial and Fibonacci spreadsheets. Arrows show how spreadsheet windows open to allow visualization of intermediate results.

8 Conclusion

We presented a programming model based on spreadsheets instead of traditional text file programs. Although spreadsheets are already computationally expressive for many tasks, we intended to improve them to allow the definition of recursive computations and better modularity and reusability of components. We justify our design through informal theoretical discussion and examples of its expressiveness.

We expect the developed environment should meet several application needs, improving the expressive power of users when dealing with computations and allowing non-programmers to enter code in the intuitive form of spreadsheets as long as they are able to understand and use the simple formula language.

The intent of this work, in the present stage, is the exploration of concepts. We are not yet concerned with providing an optimal implementation and the language and the editor are still not appealing for end-users. The designed language is targeted to a prototype system valuing completeness, simplicity and ease of implementation. A lot more of syntax sugar, basic functions and constructs should be added in order to deliver the systems to users. In the future, we plan to improve the interpreter by implementing graph rewriting, better data structure construction functions and a

design for the "scan" operation. We intend to apply the prototype in education and information visualization scenarios.

References

[Bricklin and Frankston 1999] Bricklin, D. and Frankston, B.: "VisiCalc: Information from its creators". (1999) http://www.bricklin.com/visicalc.htm

[Burnett et al. 2001] Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J., and Yang, S.: "Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm", Journal of Functional Programming 11(2), March (2001), 155-206.

[Chi et al. 1998] Chi, E. H., Riedl, J., Barry, P. and Konstan, J.: "Principles for information visualization spreadsheets". In IEEE Computer Graphics and Applications (Special Issue on Visualization) July/August. (1998) IEEE CS, p. 30-38.

[Field and Harrison 1988] Field, A. J. and Harrison, P. G.: Functional Programming, Addison-Wesley (1988).

[de Hoon et al. 1995] de Hoon, W., Rutten, L., van Eekelen, M.: "Implementing a Functional Spreadsheet in CLEAN", Journal of Functional Programming 5(3), July (1995), pp 383-414.

[Levoy 1994] Levoy, M.: "Spreadsheet for images". In Computer Graphics (SIGGRAPH '94 Proceedings), SIGGRAPH, ACM Press (1994) volume 28, pp 139-146.

[Lisper and Malmström 2002] Lisper, B. and Malmström, J.: "Haxcel: A Spreadsheet Interface to Haskell", Proc. 14th International Workshop on the Implementation of Functional Languages, Madrid, September (2002), p 206-222.

[Misner and Cooney 1991] Misner, C. W. and Cooney, P. J.: Spreadsheet Physics. Addison-Wesley, (1991).

[Nuñez 2000] Nuñez, F.: An Extended Spreadsheet Paradigm for Data Visualisation Systems, and Its Implementation, M. Sc. thesis Department of Computer Science, Faculty of Science, The University Of Cape Town (2000). URL citeseer.ist.psu.edu/543469.html