Implementing Function Spreadsheets

Peter Sestoft (sestoft@itu.dk)
IT University of Copenhagen, Denmark

ABSTRACT

A large amount of end-user development is done with spreadsheets. The spreadsheet metaphor is attractive because it is visual and accommodates interactive experimentation, but as observed by Peyton Jones, Blackwell and Burnett [6], the spreadsheet metaphor does not admit even the most basic abstraction: that of turning an expression into a named function. Hence they proposed a way to define a function in terms of a worksheet with designated input and output cells; we shall call it a *function sheet*.

The goal of our work is to develop implementations of function sheets and study their application to realistic examples. Therefore, we are also developing a simple yet comprehensive spreadsheet core implementation for experimentation with this technology.

Here we report briefly on our experiments with function sheets as well as other uses of our spreadsheet core implementation.

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming; D.3.2 [**Programming Languages**]: Language Classifications; D.3.4 [**Programming Languages**]: Processors; K.8.1 [**Personal Computing**]: Application Packages

General Terms

Human Factors, Languages, Performance

1. MOTIVATION AND RELEVANCE

The overall goal of this research is to experiment with new spreadsheet features and spreadsheet technology. This position paper focuses on the ability of end users to define new functions while staying within the spreadsheet metaphor, as proposed by Nuñez [5] and Peyton Jones, Blackwell and Burnett [6]. We use the term *function sheet* for a worksheet that defines a function.

What is the relevance of function sheets to reliability in end-user software engineering? Many skilled and highly educated people create complex spreadsheet models, because the spreadsheet metaphor is intuitive and has a low entry barrier. However, even the simplest form of abstraction — from formula to function — is not supported inside the spreadsheet metaphor, only by external languages such as VBA. When users eschew VBA, they have to fall back on duplication of formulas and the introduction of extraneous columns or rows, which leads to redundancy and scope for error.

By offering functional abstraction within the spreadsheet metaphor, function sheets permit users to avoid redundancy and to hide irrelevant detail, and hence lead to much more robust models. As in other software engineering, this may give rise to libraries of sharable function sheets that are documented, tested and reviewed, and more reliable than ad hoc spreadsheet formulas.

Even more importantly, it is much easier for an end user to customize a single function sheet than to locate and consistently modify all copies of a formula in a spreadsheet model. Hence function sheets would boost reliability of spreadsheet model maintenance.

Our specific goal therefore is to provide an implementation of function sheets that permits realistic experiments.

2. INTERPRETIVE IMPLEMENTATION

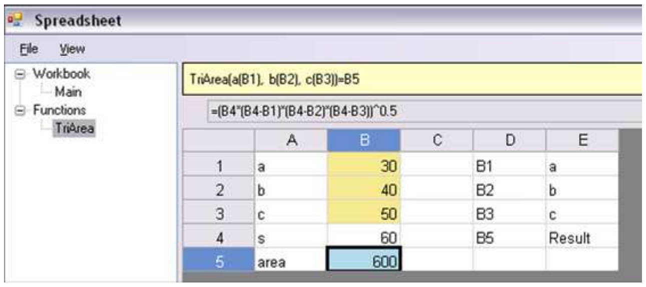
An early implementation of function sheets was developed by Cortes and Hansen [1] under my supervision and based on our spreadsheet core implementation (see section 4).

The implementation demonstrated the utility of function sheets, including array formulas and recursive and higher-order function sheets, drawing on examples from the life insurance industry. Actuarians, when informally interviewed, found the concept understandable and highly useful.

However, although this early implementation can evaluate function sheets, it lacks a user interface for editing them, is cumbersome to use, and does not provide competitive performance. Hence we have recently started creating a faster implementation with a better user interface; see section 3.

Figure 1 is a screenshot of a function sheet to compute the area $\sqrt{s(s-a)(s-b)(s-c)}$ of a triangle with side lengths a, b and c, where $s = (a+b+c)/2$ is computed in the function sheet's cell B4. The function can be called from an ordinary worksheet using an expression such as `=TriArea(A2,B2,C2)`.

The function is trivial to express as a function sheet, but expressing it in an ordinary worksheet would either require an extra column to hold s, and hence pollute the end user's



TriArea(a(B1), b(B2), c(B3))=B5

=(B4*(B4-B1)*(B4-B2)*(B4-B3))^0.5

	A	B	C	D	E
1	a	30		B1	a
2	b	40		B2	b
3	c	50		B3	c
4	s	60		B5	Result
5	area	600			

Figure 1: Function sheet `TriArea` calculates the area of a triangle. Cells B1:B3 are input cells, B4 contains the semiperimeter s, and B5 is the output cell. The B5 formula is shown in the formula bar.

working area with this irrelevant detail, or would involve this formula:

```
=SQRT((A2+B2+C2)/2*((A2+B2+C2)/2-A2)
      *((A2+B2+C2)/2-B2)*((A2+B2+C2)/2-C2)
```

whose correctness is not entirely obvious due to the duplication of the expression for s.

Nevertheless, using a function sheet to compute the area of a triangle is primarily a matter of convenience and clarity. By contrast, the examples shown in figures 2 and 3 would be impossible to express as pure spreadsheet formulas.

2.1 Recursive function sheets

Recursive function sheets were explicitly ruled out in the work of Peyton Jones, Blackwell and Burnett [6, 4.1], who argued that they did not fit with the cognitive model and that there is little need for recursive functions on numbers. We disagree on the latter point and hence do permit recursive function sheets.

Recursion and higher-order functions are well-known abstractions to mathematically inclined spreadsheet users, such statisticians and physicists, although many of their current tools, such as SAS [3] and the Fortran programming language, can express these abstractions only through arcane encodings.

Figure 2 shows a recursive and higher-order function sheet `Integrate` that computes the integral $\int_a^b f(x)dx$ of function f by adaptive numerical integration.

The bounds a and b and the function f are parameters of the function sheet. To compute the integral, `Integrate` computes two different approximations to it. If they are almost equal, we have found the integral. Otherwise the interval $[a, b]$ is split in two at $c = (a + b)/2$ and `Integrate` calls itself recursively to compute the integrals of both halves, and then adds them:

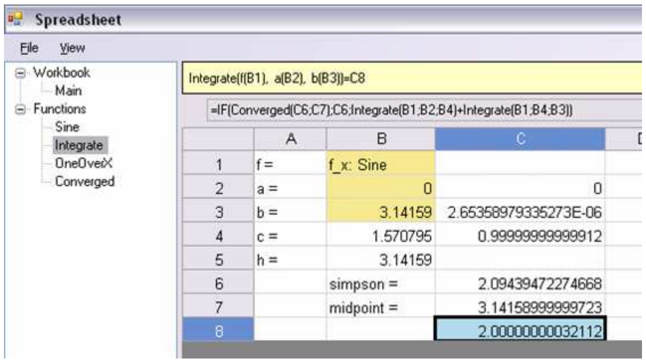
$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$$

The beauty of recursion here is that the algorithm automatically uses a coarse subdivision of the x axis where f varies little, and a fine subdivision where f varies much.

2.2 Higher-order function sheets

It is natural to pass the function f being integrated as an argument, so function `Integrate` in figure 2 is higher-order. A recent paper on integration of spreadsheets and a functional language gives another example of using higher-order functions [10].

A further reason to support higher-order functions is that they go well together with array calculations, where an arbitrary number of columns or rows are to be processed in the same way.



Integrate(f(B1), a(B2), b(B3))=C8

=IF(Converged(C6;C7);C6;Integrate(B1;B2;B4)+Integrate(B1;B4;B3))

	A	B	C
1	f =	f_x: Sine	
2	a =	0	0
3	b =	3.14159	2.65358979335273E-06
4	c =	1.570795	0.99999999999912
5	h =	3.14159	
6		simpson =	2.09439472274668
7		midpoint =	3.14158999999723
8			2.00000000032112

Figure 2: Recursive and higher-order function sheet `Integrate` to calculate the integral of f over $[a, b]$. Cells B1:B3 are input cells, B4 holds the midpoint c, C2:C4 contain the function values, C6 and C7 contain the two initial approximations to the integral, and output cell C8 holds the final approximation. The C8 formula is shown in the formula bar; it contains two recursive calls to `Integrate`.

3. COMPILED IMPLEMENTATION

To address the shortcomings of the interpretive implementation of function sheets mentioned in section 2, we have recently embarked on a compiled implementation. This implementation has a user interface for editing and evaluating ordinary worksheets, and for converting ordinary worksheets into function sheets. The implementation provides high performance, thanks to runtime bytecode generation and elimination of boxing and unboxing of intermediate values. However, this implementation does not yet support array calculations or recursive or higher-order function sheets. This is joint work with Morten Poulsen and Poul Serek.

There are some initial indications that numerical computations implemented as function sheets on this platform are just as fast as Excel's built-in functions. Figure 3 show a somewhat non-trivial function sheet, defining the cumulative distribution function of the Gaussian (normal) distribution. It is more accurate than Excel's corresponding NORMDIST built-in function. Also, although ours is a function defined by an end user, it is no slower than Excel's built-in function, when evaluated with our new prototype implementation.

The challenges when compiling function sheets to bytecode include:

- Scheduling calculations. This is done by a topological sort in dependency order, provided there are no static dependency cycles in the function sheet, which is a reasonable assumption.
- Avoiding boxing and unboxing of numerical intermediate results. This requires a type analysis of the function sheet's cells, and an careful representation of unboxed values to accommodate error values.

- Lazy evaluation of non-strict functions, such as `IF()`. This requires (1) scheduling the evaluation of the function sheet's cells according to an augmented dependency graph labeled with conditions, and (2) making evaluation of a cell `c` dependent on the disjunction, over all paths from `c` to output cells, of the conjunction of augmented dependency graph labels along that path.

4. THE CORECALC IMPLEMENTATION

The function sheet implementations described in sections 2 and 3 are based on Corecalc, a spreadsheet core implementation that we developed as a platform for experiments with spreadsheet technology [8]. Notable features of CoreCalc include:

- The formula language and the calculation engine support absolute and relative references, array formulas, arrays as cell values, non-strict and volatile built-in functions, sharing-preserving copying of formulas, and an easily extensible set of built-in functions.
- A rudimentary graphical user interface allows editing, moving and copying of cell contents, including formulas and array formulas.
- Corecalc is written in C# using Windows Forms for the graphical user interface. The C#/.NET platform is safe (unlike C and C++), efficient and widely available. Also, C# supports high-level functional and object-oriented abstractions, so the implementation is concise (4,100 lines of code) and easily modifiable.
- Despite the managed platform and the simplicity of the implementation, it achieves very good recalculation speed. Full recalculations are typically no more than 10 times slower than in Microsoft Excel, and generally faster than Gnumeric and OpenOffice Calc, two open source spreadsheet implementations.

In addition to functions sheets, a few other experiments have been based on Corecalc, including:

- Runtime bytecode generation from spreadsheet formulas for faster recalculation [4].
- A new technique to reduce recalculation in spreadsheet models, based on a so-called support graph. This achieves performance comparable to industrially used spreadsheet programs on large real-life spreadsheet models from finance, insurance, protein modeling, nuclear physics, role-playing games, and probability theory, with between 1,000 and 594,000 active cells [7, 9].

5. FUNCTION SHEETS IN EXCEL?

Using Microsoft Visual Studio tools for MS Office, it may be possible to create a C# component that works with MS Excel, such that:

- The C# component uses Excel events to intercept creation and editing of Excel worksheets with special names, such as `@MySheet`, and where named ranges within the sheet designates input and output cells. For each such sheet, the C# component parses the Excel formulas in the sheet and generates a .NET method corresponding to the function defined by that sheet.
- The C# component and a little VBA glue code installs the generated method as a WorksheetFunction callable from Excel formulas.

Then an Excel user can define function sheets as ordinary, but specially named, sheets within an Excel workbook, and call the defined functions just like ordinary Excel built-in functions. We have not yet pursued this line of work because early experiments indicated it would not work. Also, the interface from C# to Excel's object model is somewhat unattractive and imposes seemingly arbitrary limitations. For instance, some methods in the API require 30 arguments, most of which are ignored.

6. RELATED WORK

Nuñez [5] describes an extended spreadsheet with many innovations, including function sheets, and Peyton Jones, Blackwell and Burnett [6, 4.1] investigated the implications of this idea in much more depth. However, none of these works provides an efficient compiled implementation of function sheets.

There are several, mostly commercial, tools available for converting spreadsheet models into imperative program code, for instance to create web services. Presumably such tools address some of the problems involved in code generation from function sheets, but most documentation does not throw much light on this, except for Waldau's patent application (US2003226105) and Francoeur's compiled-mode implementation [2]. However, these works do not address efficient compilcation of non-strict functions such as `IF`.

Our technical report [8] contains many more references to the literature and to spreadsheet-related patents.

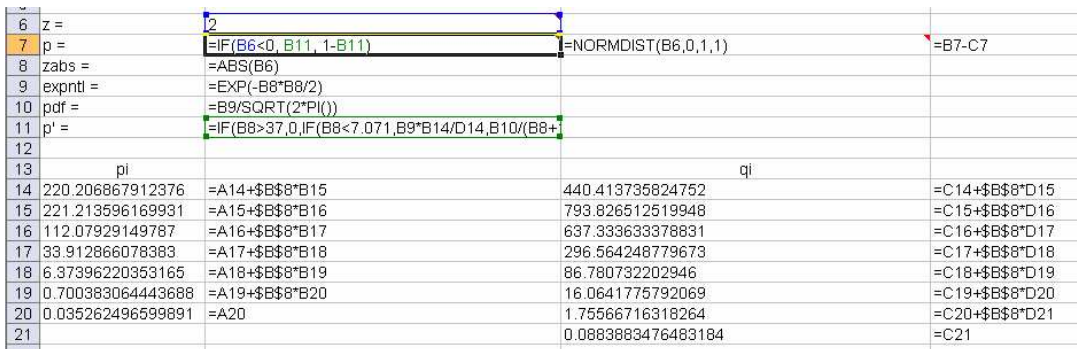
7. FUTURE WORK

Many issues have not been addressed by our work yet, including:

- Completion, quality assurance and performance testing of the compiled implementation of function sheets in section 3.
- Construction of a more robust and usable user interface, so that realistic experiments involving end-users can be conducted. Although it seems exceedingly plausible, we do not have empirical data to show that function sheets are easier to use and cause fewer reliability problems than VBA.
- Visualization of function sheet evaluation, so that end-users can debug and test function sheets the same as ordinary work sheets.

We intend to:

- Complete the compiled function sheet implementation discussed in section 3, including array formulas and higher-order and recursive function sheets;
- Further develop and maintain the spreadsheet core implementation described in section 4 and to continue to make it available to others for use or experimentation;
- Investigate whether the technology developed in section 3 can be implemented in an Excel plugin or similar, as hypothesized in section 5. That would make adoption of this technology much more likely.



6	z =	2		
7	p =	=IF(B6<0, B11, 1-B11)	=NORMDIST(B6,0,1,1)	=B7-C7
8	zabs =	=ABS(B6)		
9	expntl =	=EXP(-B8*B8/2)		
10	pdf =	=B9/SQRT(2*PI())		
11	p' =	=IF(B8>37,0,IF(B8<7.071,B9*B14/D14,B10/(B8+		
12				
13	pi		qi	
14	220.206867912376	=A14+$B$8*B15	440.413735824752	=C14+$B$8*D15
15	221.213596169931	=A15+$B$8*B16	793.826512519948	=C15+$B$8*D16
16	112.07929149787	=A16+$B$8*B17	637.333633378831	=C16+$B$8*D17
17	33.912866078383	=A17+$B$8*B18	296.564248779673	=C17+$B$8*D18
18	6.37396220353165	=A18+$B$8*B19	86.780732202946	=C18+$B$8*D19
19	0.700383064443688	=A19+$B$8*B20	16.0641775792069	=C19+$B$8*D20
20	0.035262496599891	=A20	1.75566716318264	=C20+$B$8*D21
21			0.0883883476483184	=C21

Figure 3: Function sheet to calculate the cumulative distribution function of the Gaussian distribution. Cell B6 is the input cell and cell B7 is the output cell. The sheet involves constants in A14:A20 and C14:C21, and arithmetic operations, built-in functions and conditional expressions in other cells.

8. REFERENCES

[1] Daniel S. Cortes and Morten Hansen User-defined functions in spreadsheets. Master's thesis, IT University of Copenhagen, September 2006. At http://www.itu.dk/people/sestoft/corecalc/CortesHansen2006.pdf.

[2] Joe Francoeur. Algorithms using Java for spreadsheet dependent cell recomputation. Technical Report cs.DS/0301036v2, arXiv, June 2003. At http://arxiv.org/abs/cs.DS/0301036.

[3] SAS Institute. Homepage. At http://www.sas.com.

[4] Thomas S. Iversen Runtime code generation to speed up spreadsheet computations. Master's thesis, DIKU, University of Copenhagen, August 2006. At http://www.itu.dk/people/sestoft/corecalc/Iversen.pdf.

[5] Fabian Nuñez. An extended spreadsheet paradigm for data visualisation systems, and its implementation. Master's thesis, University of Cape Town, November 2000. At http://citeseer.ist.psu.edu/543469.html.

[6] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in Excel. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 165–176, New York, NY, USA, 2003. ACM Press.

[7] M. Poulsen and P. Serek. Optimized recalculation for spreadsheets with the use of support graph. Master's thesis, IT University of Copenhagen, Denmark, 2007.

[8] P. Sestoft. A Spreadsheet Core Implementation in C#. Technical Report ITU-TR-2006-91, IT University of Copenhagen, September 2006. 135 pages.

[9] Peter Sestoft, Morten Poulsen, and Poul Serek. Minimizing recalculation in spreadsheet implementations. Draft paper, IT University of Copenhagen, Denmark, 2007.

[10] David Wakeling. Spreadsheet functional programming. *Journal of Functional Programming*, 17(1):131–143, 2007.