JFP **17** (1): 131–143, 2007.
doi:10.1017/S0956796806006186 Printed in the United Kingdom

Spreadsheet functional programming

DAVID WAKELING

The Business School, University of Gloucestershire,
The Park, Cheltenham, Gloucestershire GL50 2RH, UK
(*e-mail:* dwakeling@glos.ac.uk)

Abstract

The functional programming community has shown some interest in spreadsheets, but surprisingly no one seems to have considered making a standard spreadsheet, such as Excel, work with a standard functional programming language, such as Haskell. In this paper, we show one way that this can be done. Our hope is that by doing so, we might get spreadsheet programmers to give functional programming a try.

1 Introduction

The programming language community in general has had little time for spreadsheet programming, having concluded, it seems, that spreadsheets are intrinsically uninteresting (Casimir, 1992). The functional programming community in particular, however, has shown more interest, having realised that spreadsheet calculation is a form of functional computation. Indeed, two approaches have been tried. The first approach is one in which the spreadsheet programming language *is* a functional programming language (Wray & Fairbairn, 1989; de Hoon *et al.*, 1995; Burnett *et al.*, 2001; Malmström, 2004). This approach requires the user to abandon their investment in their favourite spreadsheet, and asks them to learn a new one. The second approach is one in which the spreadsheet programming language *incorporates ideas* from functional programming languages (Ahmad *et al.*, 2003; Peyton Jones *et al.*, 2003; Abraham & Erwig, 2004; Antoniu *et al.*, 2004). This approach permits the user to retain their investment in their favourite spreadsheet, but asks them to learn additional rules and techniques. In this paper, we consider a third approach in which the spreadsheet programming language *may be* a functional programming language. This approach permits the user to retain their investment in their favourite spreadsheet, and asks them to learn additional rules and techniques only at their own pace.

This paper is organised as follows. Section 2 introduces spreadsheets. Section 3 describes our design for spreadsheet functional programming. Section 4 outlines how this design is implemented. Section 5 presents an example. Section 6 discusses the strengths and weaknesses of our approach. Section 7 concludes.

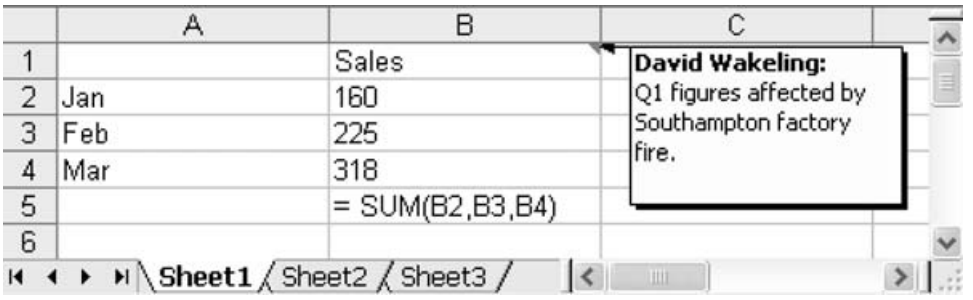

Fig. 1. A spreadsheet.

2 Spreadsheets

A *spreadsheet* (Kay, 1984) is an interactive program for performing arithmetic on columns and rows of data. It displays a *workbook* consisting of several *worksheets*[1]. See Figure 1. Each worksheet is divided into *cells*, which are arranged in a grid that may be indexed by a *reference* formed from a column letter and a row number. A cell may contain a *value* that is either a *number* or a *label*, and a *formula* that can be used to calculate the number. A formula is introduced with '=' and consists of constants, functions, and references to other cells. If the value of a cell is changed, then so are those of any cells whose formulae reference it. In large, and especially shared spreadsheets, it is often helpful to attach a *comment* to a cell to explain its value or formula. Most spreadsheets come with a *macro language* that allows them to be controlled by program code, as well as by keystrokes and mouse clicks.

3 Design

Our design provides the Excel user with a way to define Haskell functions, to use these functions in formulae, and to evaluate these formulae in spreadsheets.

3.1 Functions

The Excel macro language is a variant of Visual Basic known as Visual Basic for Applications (VBA) (Bullen *et al*., 2003). Unfortunately, the Excel VBA editor cannot sensibly be used to enter Haskell code. The problem is that it "knows" many of the rules of VBA, and its colourful and noisy interventions when those rules are broken by Haskell are too distracting. Looking for somewhere else to enter Haskell code, one soon comes upon cell comments. Once the standard box has been enlarged and the standard font changed to one of fixed-width, these are eminently suitable. See Figure 2. Of course, not all comments will be code, and to indicate those that are,

[1] Spreadsheets have a long history (Ceruzzi, 2000), but Microsoft's Excel (Microsoft Corporation, 2006) has come to dominate the market. Throughout this paper, therefore, we use Excel terminology.

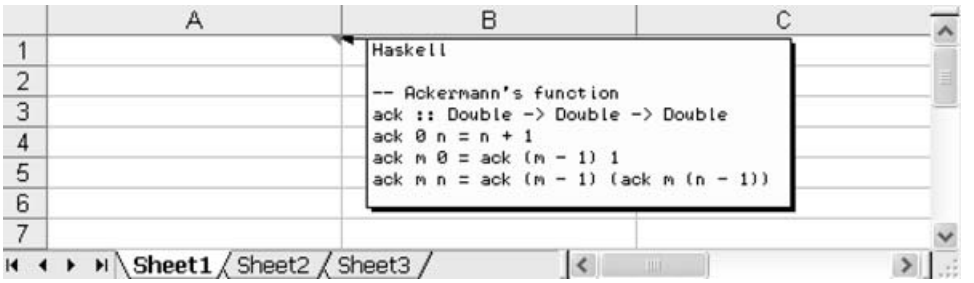

Fig. 2. Entering functions.

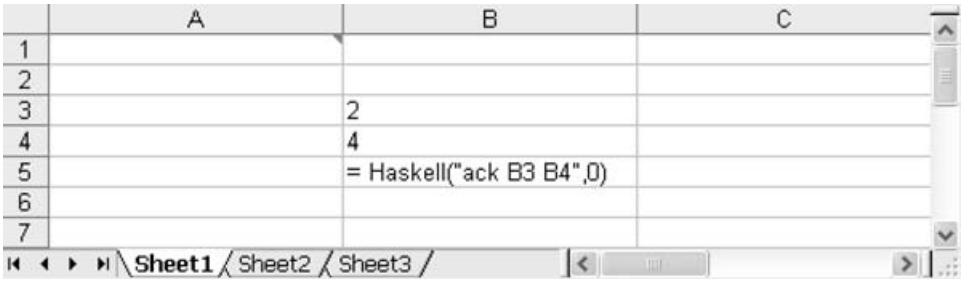

Fig. 3. Entering formulae.

we have adopted the convention that if the first line is `Haskell`, then the remaining lines are Haskell code.

3.2 Formulae

Similarly, the Excel formula bar cannot sensibly be used to enter Haskell expressions. As before, the problem is that it, and the Office Assistant, "know" the rules of Excel expressions, and their meddling and animated interventions when those rules are broken by Haskell expressions are again too distracting. A way around this is to quote expressions, so that they are treated as strings. See Figure 3. Thus, we have adopted the convention that if the outermost macro is `Haskell`, then its first argument is a quoted Haskell expression and its second argument is a number – the role of this second argument will become clear later. Of course, numbers that have a common syntax in Excel and Haskell do not need to be quoted.

3.3 Evaluation

One way to evaluate Haskell formulae would be to develop a Haskell interpreter in VBA. This would require a lot of work, and would probably be rather inefficient. Another way would be to use an existing Haskell interpreter. This would require far less work, almost certainly be more efficient, and is what we have chosen to do. Excel sends calculations to a Haskell interpreter (for example, Hugs (Jones, 1994)), which carries them out and returns the results. See Figure 4.

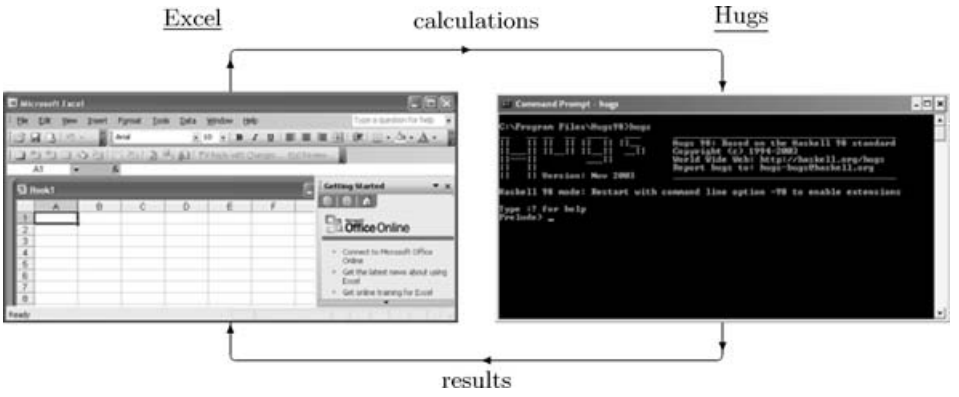

Fig. 4. Evaluating formulae.

3.4 Programming

This arrangement allows the spreadsheet programmer to write:

- functions as `Haskell` comments that may use any of the features of the language, including higher-order functions, polymorphic types and lazy evaluation;
- formulae as `Haskell` macros that may call these functions with numbers, cell references or functions (only) as arguments, and numbers (only) as results – cell calculation forces strict evaluation of these formulae.

So, for example, a financial modeller struck by the elegance of the combinator-based approach to contracts in Peyton Jones *et al.* (2000) could enter their functions directly, but could then only use them to calculate numerical results.

4 Implementation

Whenever a worksheet changes, our implementation executes some VBA code that writes a calculations file, runs a Haskell interpreter, and reads a results file.

4.1 Writing a calculations file

A calculations file contains a Haskell program. See Figure 5. This calculations file corresponds to the worksheet of Figures 2 and 3. A calculations file consists of a *prologue*, a *body* and an *epilogue*. The prologue is the same for all worksheets. It names the module and imports two standard ones. The body is made up of function and value definitions from the worksheet. The function definitions come from any `Haskell` comments, and the value definitions from any `Haskell` formulae and any numeric constants. Cell references are not valid as variable names, and so they are prefixed with an underscore. The epilogue defines a `main` function that evaluates those definitions which come from any `Haskell` formulae and writes the results to a file whose name was passed as an argument.

```
module Main (main) where
import IO
import System

-- Ackermann's function
ack :: Double -> Double -> Double
ack 0 n = n + 1
ack m 0 = ack (m - 1) 1
ack m n = ack (m - 1) (ack m (n - 1))

_B3 :: Double
_B3 = 2

_B4 :: Double
_B4 = 4

_B5 :: Double
_B5 = ack _B3 _B4

main :: IO ()
main
  = do { argv <- getArgs
       ; ifd  <- openFile (argv !! 0) WriteMode
       ; hPutStrLn ifd (show ([
             ("B5", _B5)
           ] :: [ (String, Double) ]))
       ; hClose ifd
       }
```

Fig. 5. A calculations file.

```
[("B5",11.0)]
```

Fig. 6. A results file.

4.2 Running a Haskell interpreter

A Haskell interpreter can be run by creating a separate process for it. Unfortunately, the obvious VBA `Shell` process is asynchronous, and so one must wait for it to complete by polling its status regularly.

4.3 Reading a results file

A results file contains of a list of pairs of cell references and results. See Figure 6. Again, this results file corresponds to the worksheet of Figures 2 and 3. In principle, all one need do after reading a results file is to change the values of the cells to the results. In practice, one must cope with the way that the value and formula of a cell are tied together in Excel: change the value, and the formula automatically changes to that value; change the formula, and the value automatically changes to

its result. The roundabout way that we change the value without losing the formula is therefore as follows. Suppose that before calculation, the formula is

```
= Haskell( e, x )
```

where e is a Haskell expression and x is any number. After calculation, we change the formula to

```
= Haskell( e, y )
```

where y is the result of evaluating e. The macro `Haskell` simply returns its second argument, thus changing the value from x to y, but the expression e is not lost.

5 An example

The application that we have in mind for this work is in the field of *bioinformatics*, in the area of *biological simulation*. Spreadsheet calculation is a good metaphor for biological simulation, and has been employed successfully, for example, in the Computational Cell Biology Laboratory at Virginia Polytechnic Institute (Vass *et al.*, 2002). Here, we show how a classic biological model can be simulated.

5.1 A classic model

Hodgkin and Huxley modelled the current flow through the surface membrane of the nerve fibre of a squid as the sum of the *capacitative*, *sodium*, *potassium* and *leakage* currents (Hodgkin & Huxley, 1990).

5.1.1 The capacitative current

The capacitative current, I_{cap}, is given by

$$I_{\text{cap}} = C_m \times \frac{dV_m}{dt}$$

where C_m is the membrane capacitance and V_m is the transmembrane potential.

5.1.2 The sodium current

The sodium current, I_{Na}, is given by

$$I_{\text{Na}} = g_{\text{Na,max}} \times m^3 \times h \times (V_m - E_{\text{Na}})$$

where $g_{\text{Na,max}}$ is the maximum conductance of sodium, and m and h satisfy the equations

$$\frac{dm}{dt} = \alpha_m \times (1 - m) - \beta_m \times m$$

$$\frac{dh}{dt} = \alpha_h \times (1 - h) - \beta_h \times h$$

The rate coefficients, α_m, β_m, α_h and β_h, were found by curve-fitting to be

$$\alpha_m = \frac{0.1 \times (V_m + 25.0)}{\exp(0.1 \times (V_m + 25.0)) - 1.0}$$

$$\beta_m = 4 \times \exp(V_m/18.0)$$

$$\alpha_h = 0.07 \times \exp(V_m/20.0)$$

$$\beta_h = \frac{0.1}{\exp(0.1 \times (V_m + 30.0)) + 1.0}$$

5.1.3 The potassium current

The potassium current, I_K, is given by

$$I_K = g_{K,max} \times n^4 \times (V_m - E_K)$$

where $g_{K,max}$ is the maximum conductance of potassium, and n satisfies the equation

$$\frac{dn}{dt} = \alpha_n \times (1 - n) - \beta_n \times n$$

The rate coefficients, α_n and β_n, were again found by curve-fitting to be

$$\alpha_n = \frac{0.01 \times (V_m + 10.0)}{\exp(0.1 \times (V_m + 10.0)) - 1.0}$$

$$\beta_n = 0.125 \times \exp(V_m/80.0)$$

5.1.4 The leakage current

The leakage current, I_L, is given by

$$I_L = g_L \times (V_m - E_L)$$

where E_L is the reversal potential for the leakage current.

5.1.5 The total current

Combining the equations, the total current across a cell membrane, I, is given by

$$I = I_{cap} + I_{Na} + I_K + I_L$$

That is,

$$I = C_m \times \frac{dV_m}{dt} + I_{Na} + I_K + I_L$$

This can be rearranged to

$$\frac{dV_m}{dt} = \frac{I - I_{Na} - I_K - I_L}{C_m}$$

	A	B	C	D	E
1	0.006	step			
2					
3	t	m	h	n	Vm
4	0	0.05	0.6	0.325	-75
5	= A4 + A1				
6	= A5 + A1				
7	= A6 + A1				
8	= A7 + A1				
9	= A8 + A1				
10	= A9 + A1				
11	= A10 + A1				
12	= A11 + A1				

Sheet1 Sheet2 Sheet3

Fig. 7. Entering the initial values and formulae.

5.2 Simulation

Like many other simulations, this one involves a system of *ordinary differential equations*, which in general take the form

$$\frac{dx}{dt} = f(t, x)$$

These say how a *dependent variable*, x, representing some value of interest, changes with an *independent variable*, t, usually representing time. A simulation involves solving systems of such equations numerically. A common way to do this is to use the fourth-order Runge-Kutta method summarised by the equations

$$\begin{aligned} k_1 &= sf(t_n, x_n) \\ k_2 &= sf\left(t_n + \frac{s}{2}, x_n + \frac{k_1}{2}\right) \\ k_3 &= sf\left(t_n + \frac{s}{2}, x_n + \frac{k_2}{2}\right) \\ k_4 &= sf(t_n + s, x_n + k_3) \end{aligned}$$

$$\begin{aligned} x_{n+1} &= x_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k4}{6} \\ t_{n+1} &= t + s \end{aligned}$$

where the constant s is the time step size (Gear, 1971).

In our simulation spreadsheet, then, there will be columns for the independent variable, t, and for the dependent variables, m, h, n, V_m, and a row for each time step. See Figure 7. Entering the formulae for the rows is not as tedious as it might appear because Excel provides a smart block copy-and-paste facility which adjusts copied references appropriately. Any that should not be adjusted are written with '$' characters. A function for solving ordinary differential equations numerically using the Runge-Kutta method, `rk`, and those for calculating the dependent variables,

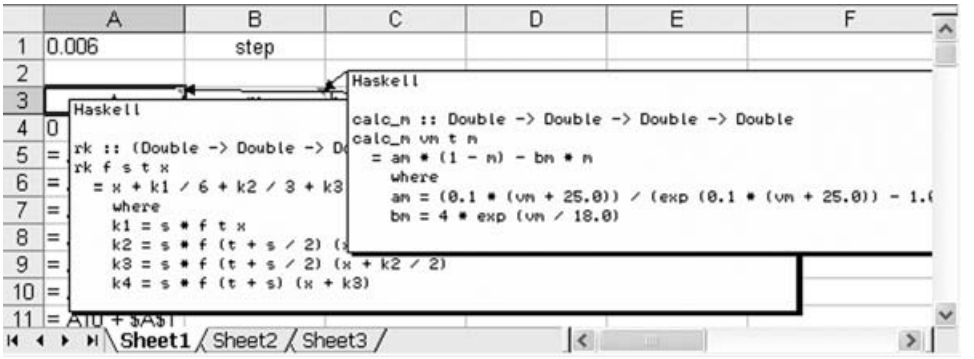

Fig. 8. Entering the calculation functions.

	A	B	C
1	0.006	step	
2			
3	t	m	h
4	0	0.05	0.6
5	= A4 + A1	= rk((calc_m E4) A1 A4 B4, 0)	
6	= A5 + A1	= rk((calc_m E5) A1 A5 B5, 0)	
7	= A6 + A1	= rk((calc_m E6) A1 A6 B6, 0)	
8	= A7 + A1	= rk((calc_m E7) A1 A7 B7, 0)	
9	= A8 + A1	= rk((calc_m E8) A1 A8 B8, 0)	
10	= A9 + A1	= rk((calc_m E9) A1 A9 B9, 0)	
11	= A10 + A1	= rk((calc_m E10) A1 A10 B10, 0)	
12	= A11 + A1	= rk((calc_m E11) A1 A11 B11, 0)	

Sheet1 Sheet2 Sheet3

Fig. 9. Entering the calculation formulae, step 1.

`calc_m`, `calc_h`, `calc_n` and `calc_vm`, are then entered as comments. See Figure 8. It then remains only to enter the formulae for calculating the dependent variables at each time step. Unfortunately, because the smart block copy-and-paste facility does not adjust references within quoted expressions, this must be done in two stages. In the first stage, the formulae are entered without the `Haskell` macro, and copied with the smart block copy-and-paste facility. See Figure 9. In the second stage, pressing a function key runs a conventional macro that wraps these formulae with the `Haskell` macro. It prompts for the top-left and then for the bottom-right cell reference where it should begin and end its work. See Figure 10. The calculation formulae have then been entered as required. See Figure 11.

Of course, the results of the simulation can be studied using any of the usual Excel built-in or third-party tools for data analysis and visualisation. See Figure 12. This simple chart may be compared with those in the original paper (Hodgkin & Huxley, 1990).

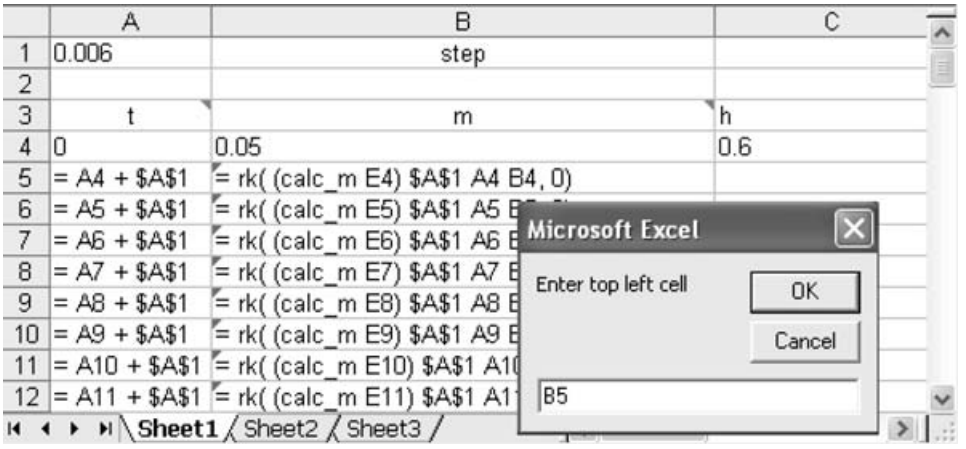

Fig. 10. Entering the calculation formulae, stage 2.

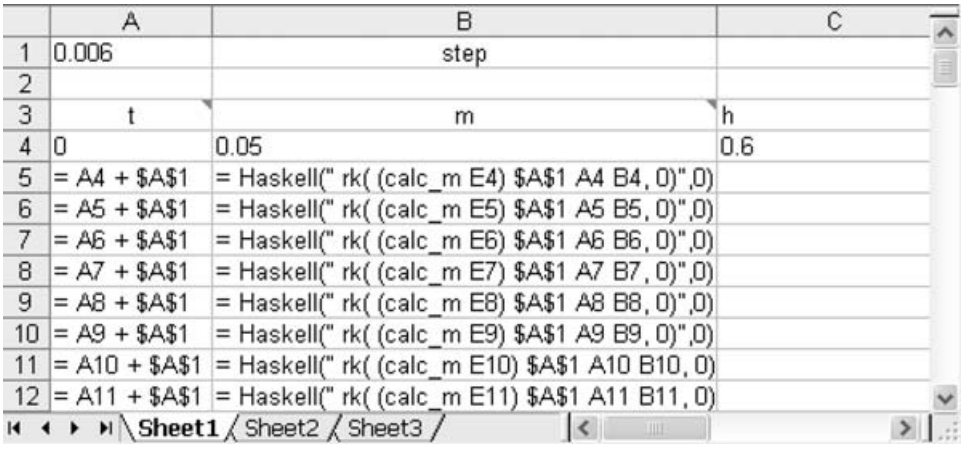

Fig. 11. Entering the calculation formulae, after stages 1 and 2.

6 Strengths and weaknesses

Below we briefly discuss the strengths and weaknesses of our approach to spreadsheet programming.

6.1 Strengths

The strengths that we claim for our approach to spreadsheet programming are as follows:

(a) *It uses an ordinary spreadsheet*. The functionality of, experience with, and support for a familiar product all carry over.
(b) *It uses an ordinary functional language*. The benefits of such languages also carry over (Backus, 1978; Turner, 1982; Hughes, 1989).

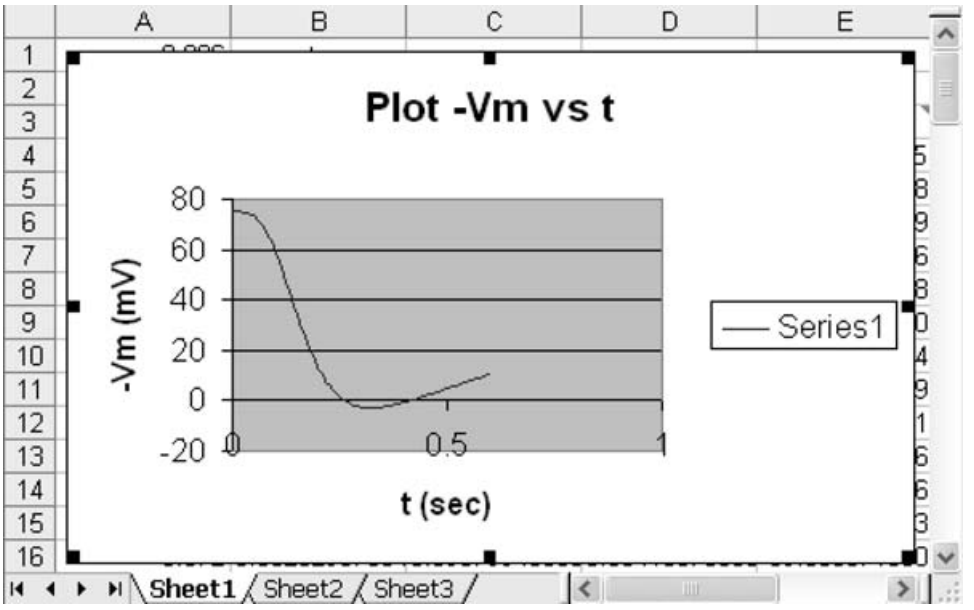

Fig. 12. A simple chart.

(c) *It can accommodate any pace of change*. The old way of doing things can coexist with the new one.

Existing approaches all seem to offer either: (a) (Abraham & Erwig, 2004; Ahmad *et al*., 2003; Antoniu *et al*., 2004); (b) (Wray & Fairbairn, 1989; de Hoon *et al*., 1995; Burnett *et al*., 2001; Malmström, 2004); or (a) and (c) (Peyton Jones *et al*., 2003). However, we are convinced that only by offering (a), (b) and (c) might we get spreadsheet programmers to give functional programming a try.

6.2 Weaknesses

The weaknesses that we admit for our approach to spreadsheet programming are as follows.

(a) *It is somewhat clumsy*. The `Haskell` macro must be remembered, and then it takes time to enter it, and space to display it.
(b) *It is somewhat inefficient*. Every calculation involves writing a file, running an interpreter, and reading a file.
(c) *It handles errors poorly*. The implementation that we have described fails silently if functions or formulae contain errors; the one that we actually use does a bit better, displaying the interpreter error messages in a dialogue box.

Of these, (a) can take some getting used to, (b) can sometimes be irritating when the spreadsheet is large or the computer is slow, and (c) can occasionally confuse novice functional programmers.

Overall, our experience has been mixed. On the positive side, many everyday programming errors are detected that would otherwise lead to incorrect results, including dangling cell references, undefined functions, and functions with the wrong type of arguments. Notice that these programming errors are different from the unit errors detected elsewhere (Ahmad *et al*., 2003; Abraham & Erwig, 2004; Antoniu *et al*., 2004), and that our work could usefully be combined with theirs. On the negative side, getting spreadsheet programmers to take up functional programming means tackling the *training problem* that is one of the reasons why no one uses functional languages (Wadler, 1998). This is an important problem – something that we were reminded of while carrying out a small experimental study with some novice functional programmers (Wakeling, 2004) – but one that is beyond the scope of this paper to solve.

7 Conclusions

In this paper, we have shown how a standard spreadsheet, Excel, and a standard functional programming language, Haskell, can be made to work together. This, we hope, might get spreadsheet programmers to give functional programming a try. Although we have concentrated on Haskell, something similar would clearly work with other programming languages for which an interpreter is available, such as Scheme, SML and Prolog, and we encourage supporters of those languages to make them work with Excel too.

Acknowledgements

Our thanks to the referees for some useful comments and suggestions.

References

Abraham, R. & Erwig, M. (2004) Header and unit interfence for spreadsheets through spatial analyses. *Pages 165–172 of: IEEE Symposium on Visual Languages and Human-centric Computing*.

Ahmad, Y., Antoniu, T., Goldwater, S. & Krishnamurthi, S. (2003) A type sysetm for statically detecting spreadsheet errors. *Pages 174–183 of: IEEE International Conference on Automated Software Engineering*. IEEE Press.

Antoniu, T., Steckler, P. A., Krishnamurthi, S., Neuwirth, E. & Felleisen, M. (2004) Validating the unit correctness of spreadsheet programs. *Pages 439–448 of: Proceedings of the International Conference on Software Engineering*. ACM Press.

Backus, J. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, **21**(8), 613–641.

Bullen, S., Green, J., Bovey, R. & Rosenberg, R. (2003) *Excel 2002 VBA programmer's reference*. Wrox Press.

Burnett, M., Atwood, J., Djang, R. W., Gottfried, H., Reichwein, J. & Yang, S. (2001) Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Func. Program.* **11**(2), 155–206.

Casimir, R. J. (1992) Real programmers don't use spreadsheets. *ACM SIGPLAN Notices*, **27**(6), 10–16.

Ceruzzi, P. E. (2000) *A History of Modern Computing*. MIT Press.

de Hoon, W. A. C. A. J., Rutten, L. M. W. J. & van Eekelen, M. C. J. D. (1995). Implementing a functional spreadsheet in Clean. *J. Func. Program.* **3**(5), 383–414.

Gear, C. (1971) *Numerical Initial Value Problems in Ordinary Differential Equations*. Longman Higher Education.

Hodgkin, A. L. & Huxley, A. F. (1990) A quantitative description of membrane current and its application to conduction and excitation in nerve. *Bull. Math. Biol.* **52**(1/2), 25–71.

Hughes, J. (1989) Why functional programming matters. *Comput. J.* **32**(2), 98–107.

Jones, M. P. (1994) *The implementation of the Gofer functional programming system*. Tech. rept. YALEU/DCS/RR-1030. Yale University, Department of Computer Science.

Kay, A. (1984) Computer software. *Scientific American*, **3**(251), 53–59.

Malmström, J. (2004) *Haxcel: A spreadsheet interface to Haskell written in Java*. M.Phil. thesis, Department of Computer Science, Mälardalen University.

Microsoft Corporation (2006) *Excel*. `http://www.microsoft.com/excel`.

Peyton Jones, S. L., Eber, J.-M. & Seward, J. (2000) Composing contracts: An adventure in financial engineering (functional pearl). *Pages 280–292 of: Proceedings of the International Conference on Functional Programming*. ACM Press.

Peyton Jones, S. L., Blackwell, A. & Burnett, M. (2003) A user-centred approach to functions in Excel. *Pages 165–176 of: Proceedings of the International Conference on Functional Programming*. ACM Press.

Turner, D. A. (1982) Recursion equations as a programming language. In: *Pages 1–28 of:* Darlington, J., Henderson, P. and Turner, D. A. (eds.), *Functional Programming and its Applications*. Cambridge University Press.

Vass, M., Shaffer, C. A., Tyson, J. J., Ramakrishnan, N. & Watson, L. T. (2002) *The JigCell model builder: A tool for modeling intra-cellular regulatory networks*.

Wadler, P. (1998) Why no one uses functional languages. *SIGPLAN Notices*, **33**(8), 23–27.

Wakeling, D. (2004) *Dealing with life in the cells: An experimental study*.

Wray, S. C. & Fairbairn, J. (1989) Non-strict languages – programming and implementation. *The Comput. J.* **32**(2), 142–151.