Compiling Spreadsheet-Defined Functions

Peter Sestoft
IT University of Copenhagen

E-mail: sestoft@itu.dk

Abstract

Spreadsheets are ubiquitous end-user programming tools, but lack even the simplest abstraction mechanism: The ability to encapsulate a computation as a function.

This paper presents a solution in the form of sheet-defined functions, *which are built from well-known spreadsheet cells, formulas and references. They should be understandable to most spreadsheet users, yet offer far more programming power than standard spreadsheet programs.*

We present a prototype implementation of sheet-defined functions and several example applications. We show that they can perform as well as built-in functions and better than external languages such as VBA.

1 Introduction

Spreadsheet programs such as Microsoft Excel, OpenOffice Calc and Gnumeric provide a simple, powerful and easily mastered end-user programming platform for mostly-numeric computation. Yet as observed by several authors [9, 11], spreadsheets lack even the most basic abstraction mechanism: The creation of a named function directly from spreadsheet formulas.

Namely, although most spreadsheet programs allow function definitions in external languages such as VBA, Java or Python, those languages present a completely different programming model that many competent spreadsheet users find impossible to use. Moreover, the external language bindings are sometimes surprisingly inefficient.

Here we present an implementation of *sheet-defined functions* that (1) uses only standard spreadsheet concepts and notations, no external languages, so it should be understandable to competent spreadsheet users, and (2) is very efficient, so that user-defined functions can be as fast as built-in ones. Furthermore, the ability to define functions directly from spreadsheet formulas should (3) permit gradual untangling of data and algorithms in spreadsheet models and (4) encourage the development of shared function libraries; both of these in turn should (5) improve reuse, reliability and upgradability of spreadsheet models.

Our implementation is written in C# and achieves high performance thanks to portable runtime code generation on the Common Language Infrastructure (CLI) [4], as implemented by Microsoft .NET and the Mono project.

Our motivation is pragmatic. A sizable minority of spreadsheet users, including biologists, physicists and financial analysts, build very complex spreadsheet models. This is because it is convenient to experiment with both models and data, and because the models are easy to share and distribute. We believe that one can advance the state of the art by giving spreadsheet users better tools, rather than telling them that they should have used Matlab, Java, Python or Haskell instead.

We do *not* think that spreadsheets will make programming languages redundant, but that they provide a computation platform many of whose positive features seem to have been overlooked, and we believe that this platform can be considerably improved by fairly simple technical means.

We have encountered spreadsheets used for protein structure modeling, financial and actuarial modeling, game character development, pesticide kinetics and much more, with up to 50 MB workbook files, 600,000 active cells, and 750 million inter-cell dependencies. Some of these take hours to load and recalculate. It seems worthwhile to improve the tools to build and run such models.

2 Sheet-defined functions

2.1 A small example

Consider the problem of calculating the area of each of a large number of triangles whose side lengths a, b and c are given in colums E, F and G of a spreadsheet, as in figure 1. The area is given by the formula $\sqrt{s(s-a)(s-b)(s-c)}$ where $s = (a+b+c)/2$ is half the perimeter. Now, either one must allocate column H to hold the value s and compute the area in column I, or one must inline s four times in

	D	E	F	G	H	I
1		a	b	c	s	area
2		3	4	5	6	6
3		30	40	50	60	600
4		100	100	100	150	4330.127019
5		6	8	10	12	24

Figure 1. Excel table of triangle side lengths and their areas, with intermediate results in column H.

A6	A	B	C	D	E
1	'Area ...				
2	'a	'b	'c	's	'area
3	3	4	5	=(A3+B3+C3)/2	=SQRT(D3*(D3-A3)*(D3-B3)*(D3-C3))
4					=MAKEFUN("triarea", E3, A3, B3, C3)

H2		E	F	G	H
1		a	b	c	area
2		3	4	5	=TRIAREA(E2, F2, G2)
3		30	40	50	600
4		100	100	100	4330.12701892219
5		6	8	10	24

Figure 2. Left: Function sheet, where `MAKEFUN` in E4 creates function `TRIAREA` with input A3, B3 and C3, output E3, and intermediate cell D3. Right: Ordinary sheet calling `TRIAREA` in H2:H5.

the area formula. The former pollutes the spreadsheet with intermediate results, whereas the latter would create a long expression that is nearly impossible to enter without mistakes. It is clear that many realistic problems require even more space for intermediate results and even more unwieldy formulas.

Here we propose instead to define a function, `TRIAREA` say, using standard spreadsheet cells and formulas, but on a separate *function sheet*, and then call this function as needed from column D on the sheet containing the triangle data.

The left-hand side of figure 2 shows a function sheet containing a definition of function `TRIAREA`, with inputs a, b and c in cells A3, B3 and C3, the intermediate result s in cell D3, and the output in cell E3.

The right-hand side shows an ordinary sheet with triangle side lengths in columns E, F and G, function calls `=TRIAREA(E2,F2,G2)` in column H to compute the triangles' areas, and no intermediate results; they exist only on the function sheet. As usual in spreadsheets, it suffices to enter the function call once in cell H2 and then copy it down column H with automatic adjustment of cell references.

2.2 Expected mode of use

A user may develop formulas on a function sheet and interactively experiment with input values and formulas until satisfied that the results are correct. Subsequently the user may turn these formulas into a sheet-defined function by calling the `MAKEFUN` built-in (see section 2.3); the function is immediately ready to use from ordinary sheets and from other functions.

Within a project, company or discipline, groups of frequently used functions can be turned into function libraries, distributed on function sheets. This makes for a smooth transition from experiments and *ad hoc* models to more stable and reliable libraries of functions, without barring users from adapting library functions to new scientific or business requirements, as may be the case with VBA libraries.

Moreover, improving the separation between "mostly data" ordinary sheets and "mostly model" function sheets provides a way to mitigate the upgrade and consistency problems caused by the strong intermixing of model and data found in most current spreadsheet models.

2.3 New built-in functions

Our prototype implementation uses the standard notions of sheet, cell, formula and built-in function, adding just three new built-in functions. As shown in cell E4 to the left in figure 2, there is a function to create a new function:

- `MAKEFUN("name", out, in1..inN)` creates a function with the given name, result cell `out`, and input cells `in1..inN`, where `N >= 0`.

Two other functions are used to compute a function value and to apply it, respectively:

- `GETFUN("name", e1..eM)` evaluates `e1..eM` to values `a1..aM` and returns a function value `(sdf, a1..aM)` which is a partial application of the sheet-defined function `"name"`.
- `APPLY(fv, e1..eN)` evaluates `fv` to a function value `(sdf, a1..aM)`, evaluates `e1..eN` to values `b1..bN`, and applies the function to these values by calling `sdf(a1..aM,b1..bN)`.

3 Interpretive implementation

Our prototype is written in C# and consists of a rather straightforward *interpretive implementation*, described in this section, combined with a novel *compiled implementation* of sheet-defined functions, described in section 4.

As in most spreadsheet programs, a workbook contains worksheets, each worksheet contains a grid of cells, and each cell may contain a constant or a formula (or nothing). A formula contains an expression and a value cache.

Since spreadsheet formulas are dynamically typed, runtime values are represented by subclasses of abstract class Value, namely Number, Text, Error, Array, and Function.

A formula expression `e` in a given cell on a given worksheet is evaluated interpretively by calling `e.Eval(sheet,col,row)`, which returns a Value object. Such interpretive evaluation involves repeated wrapping and unwrapping of values, where the most costly in terms of runtime overhead is the wrapping of IEEE 64-bit floating-point numbers (type double) as Number objects, and testing and unwrapping of Number objects as IEEE floating-point numbers. The goal of the compiled implementation presented in section 4 is to avoid this overhead.

A technical report [14] gives more details about the interpretive implementation and its design tradeoffs.

4 Compiled implementation

Our prototype compiles a sheet-defined function to CLI bytecode [4] at runtime, so that functions can be created and edited interactively, as any spreadsheet user would expect.

4.1 Compilation process outline

1. Build dependency graph whose nodes are cells transitively reachable by cell references from the output cell.
2. Perform a topological sort of the dependency graph, so a cell is preceded by all cells that it references. It is illegal for a sheet-defined function to have static cyclic dependencies.
3. If a cell in the graph is referred only once (statically), inline its formula at its unique occurrence.
4. Using the dependency graph, determine the evaluation condition (see section 5) for each remaining cell; build a new dependency graph that takes evaluation conditions into account; and redo the topological sort.
5. Generate CLI bytecode for the cells in forward topological order. For each cell `c` with associated variable `v_c`, we generate:

```
v_c = <code for c's formula>;
```

4.2 No value wrapping

The simplest compilation scheme, implemented by method `Compile` on expressions, generates code to emulate interpretive evaluation. The call `e.Compile()` must generate code that, when executed, leaves the value of `e` on the stack top as a Value object.

However, using `Compile` would wrap every intermediate result as an object of a subclass of Value, which would be inefficient, in particular for numeric operations. In an expression such as `A1*B1+C1`, the intermediate result `A1*B1` would be wrapped as a Number, only to be immediately unwrapped. The creation of that Number object would dominate all other costs, because it requires allocation in the heap and causes work for the garbage collector.

To avoid runtime wrapping of results that will definitely be used as numbers, we introduce another compilation method. The call `e.CompileToDoubleOrNan()` must generate code that, when executed, leaves the value of `e` on the stack as a 64-bit floating-point value. If the result is an error value, then the number will be a NaN.

4.3 Error propagation

When computing with naked 64-bit floating-point values, we represent an error value as a NaN and use the 51 bit "payload" of the NaN to distinguish error values, as per the IEEE standard [6, section 6.2]. Since arithmetic operations and mathematical functions preserve NaN operands, we get error propagation for free. For instance, if `d` is a NaN, then `Math.Sqrt(6.1*d+7.5)` will be a NaN with the same payload, thus representing the same error. As an alternative to error propagation via NaNs, one could use exceptions, but that is several orders of magnitude slower.

4.4 Compilation of comparisons

According to spreadsheet principles, comparisons such as `B8>37` must propagate errors, so that if B8 evaluates to an error, then the comparison evaluates to the same error. When compiling a comparison we cannot rely on NaN propagation; a comparison involving one or more NaNs is either true or false, not undefined, in CLI [4, section III.3].

Hence we introduce yet another compilation method on expressions. The call `e.CompileToDoubleProper(ifProper,ifBad)` must generate code that evaluates `e`; and if the value is a non-NaN number, leaves it on the stack top as a 64-bit floating-point value and continues with the code generated by `ifProper`; else, it leaves the value in a special variable and continues with the code generated by `ifBad`.

Here `ifProper` and `ifBad` are themselves code generators, which generate the success continuation and the failure continuation [15] of the evaluation of `e`.

4.5 Compilation of conditions

Like other expressions, a conditional `IF(e0,e1,e2)` must propagate errors from `e0`, so if `e0` gives an error value, then the entire conditional expression gives the same error value. For this purpose we introduce a fourth and final compilation method on expressions.

The call `e.CompileCondition(ifT,ifF,ifBad)` must generate code that evaluates `e`; and if the value is a non-NaN number different from zero, it continues with the code generated by `ifT`; if it is non-NaN and equal to zero, continues with the code generated by `ifF`; else, leaves the value in a special variable and continues with the code generated by `ifBad`.

For instance, to compile `IF(e0,e1,e2)`, we compile `e0` as a condition whose `ifT` and `ifF` continuations generate code for `e1` and `e2`.

4.6 On-the-fly optimizations

The four compilation methods give some opportunities for making local optimizations on the fly. For instance, in the comparison `B8>37` we must test whether B8 is an error, but clearly the constant 37 need not be tested at runtime. Indeed, method `CompileToDoubleProper` on a NumberConst object performs the error test at code generation time instead.

As another optimization, when compiling the unary logical operator `NOT(e0)` as a condition, we simply swap its `ifT` and `ifF` generators. This is useful because evaluation conditions (section 5) may contain such "silly" negations.

5 Evaluation conditions

Whereas most of the compilation machinery described in the previous section would be applicable to any dynamically typed language in which numerical computations play a prominent role, this section addresses a problem that seems unique to sheet-defined functions.

5.1 Motivation and outline

Consider computing s^n, the string consisting of $n \geq 0$ concatenated copies of string s, corresponding to Excel's built-in `REPT(`s`,`n`)`. The sheet-defined function `REPT4(s,n)` shown below is optimal, using $O(\log n)$ string concatenation operations (denoted &) for a total running time of $O(n \cdot |s|)$, where $|s|$ is the length of s:

```
B66 = <input s>
B67 = <input n>
B68 = REPT4(B66, FLOOR(B67/2,1))
B69 = <output>
    = IF(B67=0, "", IF(MOD(B67,2), B66&B68&B68,
                                   B68&B68))
```

If $n = 0$, that is `B67=0`, then the result is the empty string and there is no need to evaluate cell B68. In fact, it would be horribly wrong to unconditionally evaluate B68 because it performs a recursive call to the function itself, so this would cause an infinite loop. It would be equally wrong to inline B68's formula in the B69 formula, since that would duplicate the recursive call and make the total execution time $O(n^2 \cdot |s|)$ rather than $O(n \cdot |s|)$.

A cell such as B68 must be evaluated only when actually needed by further computations. That is the reason for step 4 in the compilation process outline in section 4.1, which we flesh out as follows:

4.1 For each cell in the sheet-defined function, compute its *evaluation condition*, a logical expression that says when the cell must be evaluated; see section 5.2.

4.2 While building the evaluation conditions, perform logical simplifications; see section 5.3.

4.3 If the cell's formula is *trivial*, then set its evaluation condition to constant true.

4.4 Rebuild the cell dependency graph and redo the topological sort of cells, taking also the cell references in the cell's evaluation condition into account.

4.5 Generate code in topological order, as in step 5 of section 4.1, modified as follows: If the cell's evaluation condition is not constant true, generate code to evaluate and cache (section 5.4) and test the evaluation condition, and to evaluate the cell's formula only if true:

```
if (<evaluation condition for c>)
  v_c = <code for c's formula>;
```

5.2 Finding the evaluation conditions

A cell needs to be evaluated if the output cell depends on the cell, given the actual values of the input cells. Hence evaluation conditions can be computed from a *conditional dependency graph*, which is a labelled multigraph.

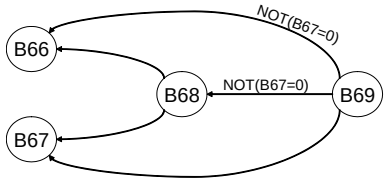

Figure 3. Evaluation dependencies in `REPT4`. Output cell B69 depends on B66 and on B68 if `NOT(B67=0)`, and unconditionally on B67.

Figure 3 shows the conditional dependency graph for function REPT4 from section 5.1. A node represents a cell, and an edge represents a dependency of one cell on another, arising from a particular cell-to-cell reference. An edge label is the condition under which the cell reference will be evaluated.

Now the *evaluation condition* of a non-input cell c is the disjunction, over all paths π from the output cell to c, of the conjunction of all labels ℓ_p along path π. More precisely, if P_c is the set of labelled paths from the output cell to c, then the evaluation condition b_c of c is

$$b_c = \bigvee_{\pi \in P_c} \bigwedge_{p \in \pi} \ell_p$$

Note that when c is the output cell itself, there is a single empty path in $P_c = \{\langle\rangle\}$, so the evaluation condition is true, as it should be. Also, if there is no path from the output to c, then the evaluation condition is false, as it should be.

The labels, or cell-cell reference conditions, on the conditional dependency graph arise from non-strict functions such as `IF(`p`,e1,e2)` and `CHOOSE(`n`,e1..en)`. For instance:

- If a cell contains the formula `IF(`q`,A1,A2+A3)`, then it has an edge to A1 with label q, and edges to A2 and A3 both with label $\neg q$. Also, if q is e.g. `B8>37`, then the cell has an edge to B8 with label true.
- If a cell contains `CHOOSE(`n`,A1,A2,A3)`, then it has an edge to A1 with label `n=1`, an edge to A2 with label `n=2`, and an edge to A3 with label `n=3`.
- If a cell contains the formula `IF(`q, e_1, e_2`)`, then edges arising from e_1 will have labels of form $q \wedge r$, and edges arising from e_2 will have labels of form $\neg q \wedge r$.

We can compute the evaluation conditions of all cells in backwards topological order. We start with the output cell, whose evaluation condition is constant true, and initially set the evaluation condition of all other non-input cells to false. To process a cell whose evaluation condition p has already been found, we traverse the abstract syntax tree of the cell's formula and accumulate (conjoin) conditions q when we process the operands of non-strict functions. Whenever we encounter a reference to cell c, we update that cell's evaluation condition b_c with $b_c := b_c \vee (p \wedge q)$.

5.3 Simplification of evaluation conditions

Since an evaluation condition must be evaluated to control the evaluation of a formula, efficiency could suffer dramatically unless the evaluation condition is reduced to the simplest logically equivalent form.

A subexpression of an evaluation condition itself may involve a recursive call or effectful external call, and therefore should be evaluated only if needed, so any logical simplifications must preserve order of evaluation. Hence we use *ad hoc* simplification rules like these, rather than reduction to disjunctive or conjunctive normal form:

$\neg\neg p$	$\Longrightarrow$	p
$p \wedge false$	$\Longrightarrow$	$false$
$p \wedge true$	$\Longrightarrow$	p
$p \vee false$	$\Longrightarrow$	p
$p \vee true$	$\Longrightarrow$	$true$
$p \wedge \neg p$	$\Longrightarrow$	$false$
$p \vee \neg p$	$\Longrightarrow$	$true$
$p \wedge q \vee p \wedge r$	$\Longrightarrow$	$p \wedge (q \vee r)$
$p \vee (p \wedge q)$	$\Longrightarrow$	p

The second last and third last rules together give the reduction $(p \wedge q) \vee (p \wedge \neg q) \Longrightarrow p$ which is useful when a cell with evaluation condition p refers to another cell A2 in both branches of a conditional `IF(`q`,A2,A2)` whose condition is q; in this case the evaluation condition of A2 is just p.

The last rule handles the case where a cell with evaluation condition p has both an unconditional and a conditional dependence q on a cell B2, as in `B2+IF(`q`,B2,...)`; in this case the evaluation condition of B2 should be just p.

In practice, these simplification rules reduce many evaluation conditions to the constant true.

5.4 Caching atomic conditions

An evaluation condition is built from logical connectives and from the conditions in non-strict functions such as `IF(B67=0,...)`; we call such a condition an *atom*. An atom may appear in the evaluation condition of multiple cells, but for correctness it must be evaluated at most once.

Namely, the atom may involve a volatile or effectful function, as in this example where cell B180 should evaluate to 1 with probability 20%, and evaluate to 10 with probability 80%:

```
B179 = ... non-trivial expression giving 1 ...
B180 = IF(RAND()<0.2, B179*B179, 10)
```

The evaluation condition of B179 is the atom `RAND()<0.2`. Since `RAND` is volatile, each evaluation may produce a different result, so we should compute the atom once, cache it, and reuse the value.

Our compiler creates a copy of the code for the atom and its caching at every use. It might be desirable to create a kind of subroutine for evaluation of the atom, in the style of the "local subroutines" of the Java Virtual Machine [8, section 7.13]. However, this is not possible in the CLI, which requires that at each bytecode location the stack must have

the same depth and types, regardless of how that point is reached [4, section I.12.3.2.1].

In practice, the code duplication caused by atom caching seems harmless. First, many formulas are trivial or have constant true evaluation conditions, in which case no atoms need to be cached at all. Second, the number of atom duplicates is usually low. Third, there is no recursive duplication, because although an atom may be nested inside another, as in `IF(IF(e00,e01,e02),e1,e2)` where both `e00` and `IF(e00,e01,e02)` are atoms, this does not lead to recursive duplication of the atom caching code.

5.5 Reflection on evaluation conditions

The approach outlined above finds the evaluation condition `NOT(B67=0)` for `B68` in `REPT4` from section 5.1, which is exactly as desired.

But why don't we simply use the caching mechanism for all cell values (instead of bothering with evaluation conditions), as in lazy functional languages [10]? The reason is that unlike atom caching, general caching may lead to an exponential code size increase: one lazily evaluated cell may contain multiple references to another lazily evaluated cell, and the code for that cell's formula will be duplicated at each possible use.

6 Example functions

Distribution function of the normal distribution. Sheet-defined functions may be used to define statistical functions, such Excel's `NORMSDIST(z)`, the cumulative distribution function $F(z)$ of the normal distribution. A widely used approximation due to Hart [5] can be implemented as shown in figure 4. Depending on z, it either computes a quotient between two polynomials (in A14:B20 and C14:D21) or a continued fraction (in B11). As shown in section 7.3, our implementation compiles this sheet-defined function to fast CLI bytecode.

Sheet-defined functions as predicates. The ability to create a (sheet-defined) function and treat it as a value gives much expressive power as is known from functional programming, with operations such as a map, fold/reduce, filter and tabulate. Here we focus on the added value for more common spreadsheet operations.

For instance, Excel's `COUNTIF` function takes as argument a cell area and a criterion, which may be a string that encodes a comparison such as `">= 18.5"`. However, one cannot express composite criteria such as `"18.5 <= x < 25"`. Passing the criterion as a string imposes arbitrary restrictions and also raises questions about the meaning of cell references in the criterion.

Passing the criterion as a sheet-defined function would make `COUNTIF` more powerful and avoid these unclarities. We can create a sheet-defined function `NORMALBMI` with input cell A1 and output cell containing `=AND(18.5<=A1, A1<25)`, and then use `COUNTIF(C1:C100, GETFUN("NORMALBMI"))` to count the number of people in range C1:C100 whose body mass index (BMI) is between 18.5 and 25, that is, "normal".

Numerical equation solving. Perhaps more surprisingly, we can implement Excel's Goal Seek feature as a sheet-defined function. Goal Seek is a dialog-based mechanism for numerical equation solving, such as "set cell C1 to 100 by changing cell B1", which really means to find a solution B1 to the equation $f(B1) = 100$ where f expresses the contents of cell C1 as a function of B1. Clearly, this f can be expressed as a sheet-defined function.

A sheet-defined function `GOALSEEK(f,r,a)` that returns an x so that $f(x) = r$, if one exists, can be defined as follows. The input is a function f, a target value r, and an initial guess a at the value of x. Function `GOALSEEK` first calls an auxiliary function to find a value b so that $f(a)$ and $f(b)$ have different signs, if possible. Then it uses a finite number of explicit bisection steps, expressed in the usual spreadsheet style of copying a row of formulas. Just 30 such steps seems to give better precision than Excel's Goal Seek wizard.

Once `GOALSEEK` has been encapsulated as a function, we can numerically solve multiple equations by ordinary copying of formulas, whereas Excel's dialog-based Goal Seek would have to be manually invoked for each equation.

Adaptive integration. To compute the integral of a function $f(x)$ on an interval $[a, b]$, we can use a combination of higher-order functions and recursion. Compute $m = (a + b)/2$ and two approximations to the integral, for instance by Simpson's rule $(b-a)(f(a)+4f(m)+f(b))/6$ and the midpoint formula $(b - a)f(m)$. If the approximations are nearly equal, return one of them; otherwise recursively compute the integral on $[a, m]$ and the integral on $[b, m]$ and add the results. Such higher-order adaptive integration can be implemented by a user-defined function using just seven formulas; it cannot be implemented using only standard spreadsheet functions or VBA.

7 Evaluation

7.1 Simplicity

We believe we have obtained a dramatic extension of the expressiveness and user-programmability of spreadsheet models, despite using no new syntax, only two new concepts, namely *sheet-defined function* and *function value*, and only three new built-in functions `MAKEFUN`, `GETFUN` and `APPLY`, described in section 2.3.

6	'z =	-5		
7	'p =	=IF(B6<0, B11, 1-B11)		
8	'zabs =	=ABS(B6)		
9	'expntl =	=EXP(-1*B8*B8/2)		
10	'pdf =	=B9/SQRT(2*PI())		
11	'p' =	=IF(B8>37, 0, IF(B8<7.071, B9*B14/D14, B10/(B8+1/(B8+2/(B8+3/(B8+4/(B8+0.65))))))		
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 4. Sheet-defined function `NORMDISTCDF(`z`)`, with input cell B6 and output cell B7, computes the cumulative distribution function of the normal distribution $N(0, 1)$.

7.2 Expressiveness

The examples in section 6 show that many useful functions can be implemented efficiently as sheet-defined functions, including functions that must be built-in black boxes in Excel and other spreadsheet programs. Also, by writing predicates as higher-order functions, Excel built-ins such as `COUNTIF` and `SUMIF` can be both more powerful and have a less obscure (less text-based) semantics.

Nevertheless, some computations are difficult or impossible to express as sheet-defined functions because we have ruled out destructive array update. Allowing destructive update would ruin the simplicity of the model and preclude parallelization; see section 9.

7.3 Performance

Figure 5 compares several (equally accurate) implementations of a statistical function. It shows that a sheet-defined function in our prototype implementation may be less than twice as slow as a function written in a "real" programming language, and considerably faster than one written in VBA or built into Excel. (Experimental platform: Pentium Mobile 1.6 GHz, Windows XP SP3, .NET 3.5 and Excel 2003).

Implementation	ns/call
Sheet-defined function `NORMDISTCDF`	255
C#	153
C (gcc 3.3.5)	204
Excel VBA (Office 2003)	3,400
Excel 2003 built-in `NORMSDIST`	9,600

Figure 5. Running time for cumulative distribution function for the normal distribution.

8 Related work

Peyton-Jones, Blackwell and Burnett proposed [11] that user-defined functions should be definable as so-called *function sheets* using ordinary spreadsheet formulas. Similar ideas are found in Nuñez's spreadsheet system ViSSh [9, section 5.2.2].

Cortes and Hansen in their 2006 MSc thesis [3] elaborated the concept of sheet-defined function and created an interpretive implementation. However, being based on the interpretive CoreCalc implementation [14], it did not achieve the performance goals of the present work.

Resolver One [12] is a commercial Python-based spreadsheet program with a feature called `RUNWORKBOOK` that allows a workbook to be invoked as a function, similar to a sheet-defined function at a coarse granularity. Invocation of a workbook is implemented by loading it from file, setting the values of some cells in it, and recalculating it, which is slow. It does not appear to support recursive invocation, nor higher-order functions. Hence it does not achieve the efficiency and expressiveness goals of the present work.

Microsoft Excel has a feature called Data Table that tabulates the values of a formula for a number of different inputs. In effect, this makes the formula a sheet-defined function of one or two arguments. However, change propagation from Data Tables is unreliable; the implementation seems half-baked. Conversely, our sheet-defined functions can easily, and reliably, emulate the Data Table mechanism: Simply use mixed absolute and relative references in a function call `=MYFUNCTION($A8,B$7)` and copy that call into all cells in the table.

Whereas we believe that the concept of evaluation condition (section 5) is novel, the compilation techniques presented in section 4 are similar to those used to compile other dynamically typed languages to efficient code [13].

9 Future work

Currently, our prototype implementation passes arguments and results of sheet-defined functions as wrapped Value objects. A global unboxing analysis or type-based unboxing [7] could further improve performance by avoiding such wrapping, especially for simple numerical functions.

While Peyton-Jones, Blackwell and Burnett verified that sheet-defined functions are understandable to spreadsheet users [11], our design deviates from theirs in several ways, so our design needs to be validated by empirical studies.

Spreadsheets exhibit quite explicit parallelism, in contrast to Fortran, Java and C# where it is only implicit and where alias analyses are required to deal with shared data and destructive update. Chandy proposed already in 1984 to exploit spreadsheet parallelism [2], and now multicore processors and graphics processors provide the required technological platform. Sheet-defined functions may play an interesting role here: since a function may be called thousands of times in each recalculation, it is a more interesting target for optimization and parallelization than an ordinary spreadsheet formula, which is evaluated at most once in each recalculation. Indeed, if parallelization is near automatic and performance is adequate, spreadsheets would become an even better framework for scientific and financial simulation [1]. We are investigating this.

Our current prototype implementation provides little of the ancillary functionality—graphics, formatting, auditing, pivot tables, data import—expected of a spreadsheet program, so it would be interesting to embed it in one that does.

10 Conclusion

We have presented a portable spreadsheet implementation in which many previously built-in functions can be user-defined, with no loss of efficiency. The main technical contribution of this work is probably the concept of evaluation conditions (section 5).

More importantly, we have demonstrated that sheet-defined functions considerably increase the expressiveness of spreadsheets while preserving their dynamic interactive behavior, and requiring only a few new concepts.

By allowing more functions to be user-defined, we soften the separation between users and developers, and empower end-users. This may lead to the development of user-created function libraries and more useful and reliable as well as faster spreadsheet models.

Acknowledgments Thanks to Bob Muller for valuable comments, and to IT University MSc students Iversen, Cortes, Hansen, Serek, Poulsen, Ha, Tran, Xu, Liton and Brønnum, who investigated many aspects of spreadsheet technology.

References

[1] D. Abramson, P. Roe, L. Kotler, and D. Mather. Activesheets: Super-computing with spreadsheets. In *2001 High Performance Computing Symposium (HPC'01), Seattle, USA*, pages 110–115, 2001.

[2] M. Chandy. Concurrent programming for the masses. (PODC 1984 invited address). In *Principles of Distributed Computing 1985*, pages 1–12. ACM, 1985.

[3] D. S. Cortes and M. Hansen User-defined functions in spreadsheets. Master's thesis, IT University of Copenhagen, September 2006.

[4] Ecma TC39 TG3. *Common Language Infrastructure (CLI). Standard ECMA-335, 3rd edition*. Ecma International, June 2005.

[5] J. Hart et al. *Computer Approximations*. Wiley, 1968.

[6] IEEE. IEEE standard for floating-point arithmetics. IEEE Std 754-2008, 2008.

[7] X. Leroy. The effectiveness of type-based unboxing. In *Types in Compilation workshop, Amsterdam*, 1997.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[9] F. Nuñez. An extended spreadsheet paradigm for data visualisation systems, and its implementation. Master's thesis, University of Cape Town, November 2000.

[10] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[11] S. Peyton Jones, A. Blackwell, and M. 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. ACM, 2003.

[12] Resolver Systems. Resolver one. Homepage. At http://www.resolversystems.com/.

[13] B. Serpette and M. Serrano. Compiling scheme to JVM bytecode: a performance study. In *International Conference on Functional Programming (ICFP) 2002*, pages 259–270. ACM, 2002.

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

[15] C. Strachey and C. Wadsworth. Continuations: a mathematical semantics for handling full jumps. *Higher Order and Symbolic Computation*, 13:135–152, 1974. Reprint of Oxford PRG-11, January 1974.