Sheet-defined functions: implementation and initial evaluation
Version 1.1 of 2013-01-16

Peter Sestoft and Jens Zeilund

IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark

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 was observed by Peyton-Jones and others [14], who proposed a mechanism to define such functions using only standard spreadsheet cells, formulas and references.
This paper extends their work by increasing expressiveness and emphasizing execution speed of the functions thus defined. First, we support recursive and higher-order functions, while still using only standard spreadsheet notation. Secondly, we obtain fast execution by a careful choice of data representation and compiler technology.
The result is a concept of *sheet-defined functions* that should be understandable to most spreadsheet users, yet offer sufficient programming power and performance to make end-user development of function libraries practical and attractive.
We outline a prototype implementation Funcalc of sheet-defined functions, and provide a case study with some evidence that it can express many important functions while maintaining good performance.

1 Introduction

Spreadsheet programs such as Microsoft Excel, OpenOffice Calc, Gnumeric and Google Docs provide a simple, powerful and easily mastered end-user programming platform for mostly-numeric computation. Yet as observed by several authors [12, 14], spreadsheets lack even the most basic abstraction mechanism: The creation of a named function directly from spreadsheet formulas.

Many spreadsheet programs allow function definitions in external languages such as VBA, Java or Python, but those languages present a completely different programming model that many competent spreadsheet users struggle to use.

Here we present a prototype implementation, called Funcalc, of *sheet-defined functions* that (1) uses only standard spreadsheet concepts and notations, as proposed by Peyton-Jones et al. [14], 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) [6], as implemented by Microsoft .NET and the Mono project.

Our ultimate motivation is pragmatic. A sizable minority of spreadsheet users, including biologists, physicists and financial analysts, build very complex spreadsheet models. This is because spreadsheets make it convenient to experiment with both computations and data, and because the resulting 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 we do believe that they provide a computation platform with many useful features that can be considerably improved by fairly simple technical means.

	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

Fig. 1. Triangle side lengths and computed areas, with intermediate results in column H.

A6	A	B	C	D	E	F
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					=DEFINE("triarea", E3, A3, B3, C3)	

Fig. 2. Function sheet, where `DEFINE` in E4 creates function `TRIAREA` with input cells A3, B3 and C3, output cell E3, and intermediate cell D3.

H2	E	F	G	H	I
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	

Fig. 3. Ordinary sheet calling `TRIAREA`, defined in Fig. 2, from cells H2:H5.

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 columns E, F and G of a spreadsheet, as in Fig. 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 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 the sheet containing the triangle data.

Fig. 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.

Fig. 3 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; these 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 `DEFINE` built-in (see Sect. 2.3); the function is immediately ready to use from ordinary sheets and from other functions.

Within a project, company or scientific 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 sometimes caused by the strong intermixing of model and data found in many spreadsheet models.

2.3 New Built-In Functions

Our prototype implementation uses the standard notions of sheet, cell, formula and built-in function. It adds just three new built-in functions to support the definition and use of sheet-defined functions. As illustrated by cell E4 in Fig. 2, there is a function to create a new function:

- `DEFINE("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 create a function value (closure) and to apply it, respectively:

- `CLOSURE("name", e1..eM)` evaluates `e1..eM` to values `a1..aM` and returns a closure for the sheet-defined function `"name"`. An argument `ai` that is an ordinary value gets stored in the closure, whereas an argument that is `NA()` signifies that this argument will be provided later when calling the closure.
- `APPLY(fv, e1..eN)` evaluates `fv` to a closure, evaluates `e1..eN` to values `b1..bN`, and applies the closure by using the `bj` values for those arguments in the closure that were `NA()` at closure creation.

The `NA()` mechanism provides a very flexible way to create closures (partially applied functions), which is rather unusual from a programming language perspective, but fits well with the standard spreadsheet usage of `NA()` to signify a value that is not (yet) available.

3 Interpretive Implementation

Our prototype implementation is written in C# and consists of a rather straightforward *interpretive implementation* combined with a novel *compiled implementation* of sheet-defined functions, described in Sects. 4 and 5.

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. A worksheet is represented as a "lumpy" sparse array data structure that is space-efficient, highly scalable, and performs very well on modern CPUs.

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 (C# type `double`) as Number objects, and testing and unwrapping of Number objects as IEEE floating-point numbers. One goal of the compiled implementation presented in Sect. 4 is to avoid this overhead.

4 Compiled Implementation

Our prototype is written in C# and compiles a sheet-defined function to CLI bytecode [6] at runtime, so that functions can be created and edited interactively, as any spreadsheet user would expect.

This section outlines the compilation process and some of the steps taken to ensure good performance.

4.1 Compilation Process Outline

1. Build a dependency graph whose nodes are the 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. This saves a local variable at no cost in code size.
4. Using the dependency graph, determine the evaluation condition (see Sect. 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`, generate code corresponding to this assignment:

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

4.2 No Value Wrapping

The simplest compilation scheme generates code to emulate interpretive evaluation. The code for an expression `e` leaves the value of `e` on the (CLI virtual machine) stack top as a Value object.

However, wrapping every intermediate result as an object of a subclass of Value 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 useless Number object is slow: it requires allocation in the heap and causes work for the garbage collector.

Therefore, when the result of an expression `e` will definitely be used as a number, we use a second compilation method. It generates code that, when executed, leaves the value of `e` on the stack as a 64-bit floating-point value, avoiding costly allocation in the heap. If the result of `e` is an error (or a non-number such as a text or array), the resulting number will be a NaN [8].

4.3 Efficient 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 [8, section 6.2], which is supported by all modern hardware. 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 considerably slower.

4.4 Compilation of Comparisons

According to spreadsheet principles, a comparison such as `B8>37` must propagate errors. If B8 evaluates to an error, then the entire 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, never undefined, in CLI [6, section III.3].

Therefore we introduce a third compilation method. It takes an expression `e` and two code generators `ifProper` and `ifBad`. It generates code that evaluates `e`; and if the value is a non-NaN number, leaves that value on the stack top as a 64-bit floating-point value and continues with the code generated by `ifProper`; otherwise, continues with the code generated by `ifBad`.

The code generators `ifProper` and `ifBad` generate the success continuation and the failure continuation [20] for the evaluation of `e`.

4.5 Compilation of Conditions

Like other expressions, a conditional `IF(e0,e1,e2)` must propagate errors from the condition `e0`, so if `e0` gives an error value, then the entire conditional expression must give the same error value.

To achieve this we introduce a fourth compilation method, for expressions that are used as conditions. The method takes an expression `e0` and three code generators `ifT`, `ifF` and `ifBad`, and generates code that evaluates `e0`; 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`; otherwise, continues with the code generated by `ifBad`.

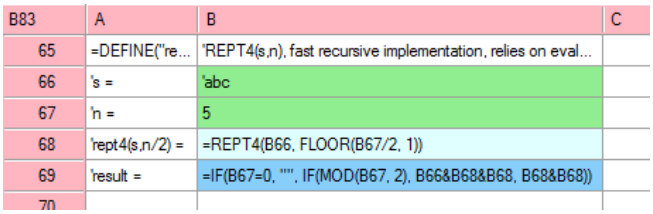
For instance, to compile `IF(e0,e1,e2)`, we compile `e0` as a condition whose `ifT` and `ifF` continuations generate code for `e1` and `e2`.

5 Evaluation Conditions

Whereas most of the compilation machinery described in Sect. 4 would be applicable to any dynamically typed language in which numerical computations and error propagation play a prominent role, this section addresses a problem that seems unique to recursive 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)` in Fig. 4 is optimal, using $O(\log n)$ string concatenation operations (written `&`) for a total running time of $O(n \cdot |s|)$, where $|s|$ is the length of s.



B83	A	B	C
65	=DEFINE("re...	'REPT4(s,n), fast recursive implementation, relies on eval...	
66	's =	'abc	
67	'n =	5	
68	'rept4(s,n/2) =	=REPT4(B66, FLOOR(B67/2, 1))	
69	'result =	=IF(B67=0, "", IF(MOD(B67, 2), B66&B68&B68, B68&B68))	
70			

Fig. 4. Recursive function `REPT4(s,n)` illustrates the need for evaluation conditions.

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 $O(n \cdot |s|)$, thwarting the programmer's intentions.

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 Sect. 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 Sect. 5.2.
4.2 While building the evaluation conditions, perform logical simplifications; see Sect. 5.3.
4.3 If the cell's formula is *trivial*, for instance a constant or a cell reference, then set its evaluation condition to constant true, indicating unconditional evaluation.
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 Sect. 4.1, modified as follows: If the cell's evaluation condition is not constant true, generate code to evaluate and cache (Sect. 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.

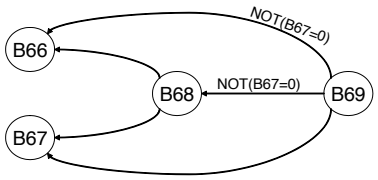


Fig. 5. Evaluation dependencies in `REPT4`. Output cell B69 depends on B66 and on B68 if `NOT(B67=0)`, and unconditionally on B67.

Fig. 5 shows the conditional dependency graph for function `REPT4` from Sect. 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 (must evaluate). Also, if there is no path from the output to c, then the evaluation condition is false (need not evaluate).

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 order-preserving simplification rules, rather than reduction to disjunctive or conjunctive normal form.

The approach outlined above finds the evaluation condition `NOT(B67=0)` for `B68` in `REPT4` from Fig. 4, which is exactly as desired.

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, because it may involve a call to a volatile function such as `RAND()` that would produce different results on each evaluation.

Hence each occurrence of an atom is compiled to a *cache* that tests whether the atom has already been evaluated, and if so just returns the cached value; and if not, evaluates the atom and saves the value. In the cache, an evaluated atom is represented by its value, and an unevaluated one is represented by a special NaN.

5.5 Reflection on Evaluation Conditions

Why don't we simply use the caching mechanism for all cell values (instead of bothering with evaluation conditions), as in lazy functional languages [13]? One reason is that unlike atom caching, general expression 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. Moreover, this exponential code size blowup is likely to happen in practice.

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			0.088388347648318 4	=C21

Fig. 6. Sheet-defined function `NORMDISTCDF(z)`, with input cell B6 and output cell B7, computes the cumulative distribution function of the normal distribution $N(0, 1)$.

6 Some 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 [7] can be implemented as shown in Fig. 6. Depending on z, it either computes a quotient between two polynomials (in A14:B20 and C14:D21) or a continued fraction (in B11). 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 makes `COUNTIF` more powerful and avoids 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, CLOSURE("NORMALBMI",NA()))` 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.

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 formula cells; it cannot be implemented using only standard spreadsheet functions or VBA.

Correct and comprehensive calendar functions The calendar functions in many spreadsheet programs do not handle ISO week numbers, calculation of holidays (such as Easter), finding the first Monday of a given month, and so on. Such computations are easily and efficiently implementable as sheet-defined functions, starting from a source such as [5].

7 Case Study: Financial Functions

The second author [21] evaluated the feasibility of using sheet-defined functions (instead of built-in ones) by implementing many of the financial functions that are built into Microsoft Excel 2010. This case study was chosen because (1) finance is an important application domain for spreadsheets, and (2) a faithful implementation of Excel financial functions is available in the functional language F#, complete with source code and thousands of test cases [2].

This evaluation was carried out by a software development student, and we do not claim that it says much about the ease of programming with sheet-defined functions. However, we do claim that it demonstrates that sheet-defined functions can be expressive and fast enough to replace built-in ones.

7.1 Performance of Sheet-Defined Financial Functions

Fig. 7 lists some of the implemented financial functions. In most cases the sheet-defined functions are faster than the corresponding Excel built-ins, or comparable to them. Two notable exceptions are functions `RATE` and `IRR`, marked by an asterisk (*) in the figure. The reason for their poor performance probably is that they use a naive general binary search procedure, instead of a Newton-Raphson root-finding algorithm, for instance. This is a question of choice of algorithm, not a problem of the sheet-defined function implementation itself.

Function	Excel	Funcalc	Note
PV	1461	804	
FV	1445	1138	
NPER	1055	472	
RATE	2297	44864	*
PMT	1523	664	
FVSCHEDULE	2960	928	
IMPT	1593	1732	
PPMT	1805	1292	
CUMIPMT	3117	3400	
CUMPRINC	2742	4072	
ISPMT	468	170	
IRR	4750	79804	*
NPV	2156	2060	
MIRR	3515	8328	
SLN	125	158	
SYD	453	212	
AMORLINC	14921	2054	
AMORDEGRC	16343	4444	

Fig. 7. Execution time for Excel 2010 built-in functions and Funcalc sheet-defined functions (ns/call). For the *-marked cases, see text.

7.2 Ideas for Improvement Arising from Case Study

The process of implementing the financial functions generated several ideas for improving our prototype Funcalc (none of which have yet been implemented), including these:

- Proposal: Add a simple scope mechanism.
  Problem: Funcalc, like other spreadsheet programs, has a single scope, so all names are visible anywhere in a workbook. This pollutes the global namespace with auxiliary functions and may lead to name clashes.
  Possible solutions: (1) Name-based scope. A function `_FOO` whose name begins with a single underscore is a *global auxiliary* and can be called only from function sheets, not from an ordinary sheet; a function `__FOO` whose name begins with two underscores is *sheet-local* and can be called only from the function sheet in which it is defined; a function `_BAR_FOO` is *function-local* and can be called only from public function `BAR` and from other function-local auxiliaries such as `_BAR_BAZ`. (2) Visual scope. A global function and all its auxiliaries are surrounded by a graphical "fence", restricting the scope of the auxiliaries.
- Proposal: Avoid infinite recursion, especially when loading workbooks.
  Problem: A recursive function may fail to terminate (go into an infinite loop), a mistake that is especially nasty during the loading of a workbook from file.
  Possible solutions: (1) Allow manual interruption of computations, for instance by pressing ESC or Ctrl-C. Such interruption may leave a computation (a recursive call) unfinished; in this case its result might be a special

kind of error such as `#BREAK` or `#LOOP`, which would propagate as usual to any cell depending on it. (2) Set a function call limit for each recalculation, and make it low when recalculating a workbook upon reloading. The same error mechanism could be used as for manual interruption. It would be more useful to limit the call depth rather than the total number of calls, but the latter may be simpler to implement, and faster.

– Proposal: Error messages should be made more informative.
  Problem: According to spreadsheet semantics, an error value propagates from operand to result. In an ordinary spreadsheet where all cells are manifest, it is fairly easy to trace an error back to the cell containing the original offending formula. With sheet-defined functions, the error may have originated in a deeply nested auxiliary function, and tracing this can be very cumbersome.
  Possible solutions: (1) Make error values carry the address of the cell containing the original offending computation, for instance, as `#NUM!#Sheet1!A1`, instead of just `#NUM!`. (2) For errors originating from within a sheet-defined function, make the error value carry the entire argument vector of the (innermost) function call that caused the function to return an error value. This would enable "replaying" that call and hence enable debugging.

8 Evaluation

8.1 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 `DEFINE`, `CLOSURE` and `APPLY`, described in Sect. 2.3.

The prototype implementation is relatively compact, comprising less than 13,000 lines of C# code.

8.2 Expressiveness

Sects. 6 and 7 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 much more powerful and have a less obscure (less text-based) semantics.

Although not illustrated here, sheet-defined functions can take array (range) values as arguments and return them as results. Since the "language" of sheet-defined functions supports recursive and higher-order functions, and is dynamically typed, it is conceptually similar to a pure (side-effect free) version of Lisp [10] or Scheme, albeit with a very unusual syntax.

Some computations are difficult or impossible to express as sheet-defined functions, chiefly because we have ruled out side-effects and destructive array update. Yet we do not want to support side-effects, because that would ruin the simplicity of the model and the compiler's freedom to rearrange computations. In particular, it would complicate parallelization; see Sect. 10.

8.3 Performance

According to micro-benchmarks (not shown here) a non-trivial numerical sheet-defined function such as that in Fig. 6 can be considerably faster than a corresponding user-defined function in VBA (the macro language of MS Excel), and only 2–3 times slower than a function written in a "proper" programming language such as C, Java or C#. This is quite satisfying, given that our sheet-defined functions are dynamically typed and that the compiler is quite compact.

Moreover, benchmarking results from the case study in Sect. 7 show that financial functions built in to Excel can be implemented as sheet-defined functions without loss of efficiency. This is important because it shows that such libraries of functions need not be built-in, but could be developed and maintained by the relevant user communities, without resorting to external programming languages.

9 Related Work

Peyton-Jones, Blackwell and Burnett proposed [14] 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 [12, section 5.2.2]. What we have implemented is strongly inspired by Peyton-Jones et al., but extends expressiveness by permitting recursive and higher-order functions.

Cortes and Hansen in their 2006 MSc thesis [4] elaborated the concept of sheet-defined function and created an interpretive implementation. However, being based on the interpretive CoreCalc implementation [17], it cannot achieve the performance goals we have set in the present work.

Resolver One [15] 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 coarser 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.

We believe that the concept of evaluation condition (Sect. 5) is original with this work. The other compilation techniques presented in Sect. 4 are similar to those used by other dynamically typed languages [16].

Preliminary reports of this work includes an oral presentation [18] and a rough draft of a book-length manuscript [19]. None of these includes the case study reported in Sect. 7.

10 Perspectives and Future Work

Currently, our prototype implementation passes arguments and results of sheet-defined functions as wrapped objects. A global unboxing analysis or type-based

unboxing [9] 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 [14], our design deviates from theirs in several ways, so our design needs to be revalidated empirically.

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 [3], and today 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. If parallelization is near automatic and performance is adequate, spreadsheets could become an even better framework for scientific and financial simulation [1]; a framework for "end-user high-performance computing". In fact, spreadsheets with sheet-defined functions constitute a dataflow language in the style of Sisal [11], so it may be possible to leverage the 1990es work on automatic parallelization of such languages.

Our prototype is a standalone spreadsheet implementation with a simplistic user interface. It provides very little of the ancillary functionality—graphics, formatting, auditing, pivot tables, data import—expected of a spreadsheet program, so it would be interesting to turn it into a plugin for one that does, such as Excel.

11 Conclusion

We have shown that a spreadsheet implementation can accommodate user-defined functions with sufficient convenience and performance that previously built-in functions can be user-defined instead.

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 expressive, more reliable and faster spreadsheet models.

The main *technical* innovation required to achieve this is probably the concept of evaluation conditions (Sect. 5).

Moreover, we have demonstrated that sheet-defined functions considerably increase the expressiveness of spreadsheets while preserving their dynamic interactive behavior, and with conceptual parsimony, requiring only a few new concepts and built-in functions, and no new notation.

Acknowledgments Thanks to Bob Muller for valuable comments, and to IT University MSc students Iversen, Cortes, Hansen, Serek, Poulsen, Ha, Tran, Xu, Liton, Brønnum, Hamann, Patapavicius, Salas and Nielsen 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. L. Bolognese. Excel financial functions for .NET. MSDN webpage, 2009. At http://archive.msdn.microsoft.com/FinancialFunctions.
3. M. Chandy. Concurrent programming for the masses. (PODC 1984 invited address). In *Principles of Distributed Computing 1985*, pages 1–12. ACM, 1985.
4. D. S. Cortes and M. Hansen User-defined functions in spreadsheets. Master's thesis, IT University of Copenhagen, September 2006.
5. N. Dershowitz and E. M. Reingold *Calendrical calculations*. Cambridge University Press, third edition edition, 2008.
6. Ecma TC39 TG3. *Common Language Infrastructure (CLI). Standard ECMA-335, 3rd edition*. Ecma International, June 2005.
7. J. Hart et al. *Computer Approximations*. Wiley, 1968.
8. IEEE. IEEE standard for floating-point arithmetics. IEEE Std 754-2008, 2008.
9. X. Leroy The effectiveness of type-based unboxing. In *Types in Compilation workshop, Amsterdam*, 1997.
10. J. McCarthy et al. *Lisp 1.5 Programmer's Manual*. MIT Press, 1962.
11. J. McGraw et al. Sisal. Streams and iteration in a single assignment language. Language reference manual, version 1.2. Technical report, Lawrence Livermore National Labs, March 1985.
12. F. Nuñez. An extended spreadsheet paradigm for data visualisation systems, and its implementation. Master's thesis, University of Cape Town, November 2000.
13. S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
14. 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.
15. Resolver Systems. Resolver one. Homepage. At http://www.resolversystems.com/.
16. 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.
17. P. Sestoft. A Spreadsheet Core Implementation in C#. Technical Report ITU-TR-2006-91, IT University of Copenhagen, September 2006. 135 pages.
18. P. Sestoft. Implementing function spreadsheets. Oral presentation, Fourth Workshop on End-User Software Engineering (WEUSE IV), Leipzig, Germany, May 2008. At http://www.itu.dk/people/sestoft/papers/weuse-sestoft.pdf.
19. P. Sestoft. Spreadsheet Technology. version 0.12 of 2012-01-31. Technical Report ITU-TR-2011-142, IT University of Copenhagen, January 2012.
20. 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.
21. J. Z. Sørensen. An evaluation of sheet-defined financial functions in Funcalc. Master's thesis, IT University of Copenhagen, March 2012.