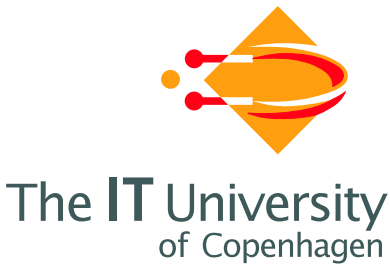

A Spreadsheet Core Implementation in C#

Version 1.0 of 2006-09-28

Peter Sestoft

IT University Technical Report Series **TR-2006-91**

ISSN 1600–6100 **September 2006**

ISSN 1600–6100

ISBN 87-7949-135-9

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaardsvej 7
DK-2300 Copenhagen S
Denmark

Telephone: **+45 72 18 50 00**
Telefax: **+45 72 18 50 01**
Web `www.itu.dk`

Preface

Spreadsheet programs are used daily by millions of people for tasks ranging from neatly organizing a list of addresses to complex economical simulations or analysis of biological data sets. Spreadsheet programs are easy to learn and convenient to use because they have a clear visual data model (tabular) and a simple efficient computation model (functional and side effect free).

Spreadsheet programs are usually not held in high regard by professional software developers [15]. However, their implementation involves a large number of non-trivial design considerations and time-space tradeoffs. Moreover, the basic spreadsheet model can be extended, improved or otherwise experimented with in many ways, both to test new technology and to provide new functionality in a context that could make a difference to a large number of users.

Yet there does not seem to be a coherently designed, reasonably efficient open source spreadsheet implementation that is a suitable platform for experiments. Existing open source spreadsheet implementations such as Gnumeric and OpenOffice are rather complex, written in unmanaged languages such as C and C++, and the documentation of their internals is sparse. Commercial spreadsheet implementations such as Microsoft Excel neither expose their internals through their source code nor through adequate documentation of data representations and functions.

Goals of this report The purpose of this report is to enable others to make experiments with innovative spreadsheet functionality and with new ways to implement it. Therefore we have attempted to collect in one place a considerable body of knowledge about spreadsheet implementation.

To our knowledge neither the challenges of efficient spreadsheet implementation nor possible solutions to them are systematically presented in the existing scientific literature. There are many patents on spreadsheet implementation, but they offer a very fragmented picture, since patents traditionally do not describe the prior art on which they build.

This report is a first attempt to provide a more coherent picture by gleaning information from experience with existing spreadsheet implementations and with our own implementation CoreCalc, from the scientific literature, and from patents and patent applications. For commercial software, this necessarily involves some guesswork, but we have not resorted to any form of reverse engineering.

Contents of this report The report comprises the following parts:

- A summary of the spreadsheet computation model and the most important challenges for efficient recalculation, in chapter 1.
- A survey of scholarly works, spreadsheet implementations and related patents, in sections 1.10 through 1.12.
- A description of a core implementation of essential spreadsheet functionality for making practical experiments, in chapter 2. The implementation itself, called CoreCalc, is available in source form under a liberal license, and is written in C#, using only managed code.
- A discussion of alternatives to some of the design decisions made in CoreCalc, in chapter 3.
- A thorough investigation of one particular design alternative, presenting a possibly novel way to represent the support graph, an important device for achieving minimal recalculation, in chapter 4.
- A description of an experiment with using runtime bytecode generation on the .NET platform for efficient formula evaluation, in chapter 5.
- A description of experiments with permitting users to define functions in terms of special function sheets, rather than in an extraneous programming language such as VBA, in chapter 6.
- A list of possible extensions and future projects, in chapter 7.
- An appendix listing all known US patents and patent applications related to spreadsheet implementation, in appendix A.

Goals of the CoreCalc implementation The purpose of the CoreCalc implementation described in chapter 2 of this report is to provide a source code platform for experiments with spreadsheet implementation. The CoreCalc implementation is written in C# and provides all essential spreadsheet functionality. The implementation is small and simple enough to allow experiments with design decisions and extensions, yet complete and efficient enough to benchmark against real spreadsheet programs such as Microsoft Excel, Gnumeric and OpenOffice Calc, as shown in section 5.2.

The hope is that this core implementation can be used in further experiments with implementation design decisions and extended functionality.

Availability and license The complete implementation, including documentation, is available in binary and source form from the IT University of Copenhagen:

`http://www.itu.dk/people/sestoft/corecalc/`

The CoreCalc implementation is copyrighted by the authors and distributed under a BSD-style license:

This means that you can use and modify the CoreCalc software for any purpose, including commerce, without a license fee, but the copyright notice must remain in place, and you cannot blame us for any consequences of using or abusing the software. In particular, we accept no responsibility if the commercial exploitation of an idea presented in this report is construed to violate one or more patents. This license does not apply to the ideas discussed in chapter 4.

Also, all trademarks belong to their owners.

Acknowledgements The original impetus to look at spreadsheets came from Simon Peyton Jones and Margaret Burnett during a visit to Microsoft Research, Cambridge UK, in 2001, and from their 2003 paper with Alan Blackwell [70].

Thomas S. Iversen investigated the use of runtime code generation for speeding up spreadsheet calculations in his 2006 MSc thesis project [45], jointly supervised with Torben Mogensen (DIKU, University of Copenhagen). Parts of this work are summarized in chapter 5. Thomas also restructured the core code base and added functionality to read XMLSS files exported from Microsoft Excel.

Daniel S. Cortes and Morten W. Hansen investigated how to design and implement so-called function sheets, thus allowing spreadsheet users to define their own functions using well-known spreadsheet concepts. This work was done in their 2006 MSc thesis project [20]. Parts of this work are summarized in section 6.1.

Quan Vi Tran and Phong Ha investigated an alternative implementation of function sheets, using the infrastructure provided by Microsoft Excel. This work was done in their 2006 MSc thesis project [40] and is briefly summarized in section 6.2.

Other IT University students, including Jacob Atzen, Claus Skoubølling Jørgensen and Jens Lind, investigated other corners of the spreadsheet design space.

Source code naming conventions

Name	Meaning	Type	Page
`ae`	adjusted expression	Adjusted<Expr>	45
`c`	column index variable	`int`	
`ca`	cell address, absolute	CellAddr	33
`ccar`	cell or cell area reference	CellRef, CellArea	69
`cell`	cell	Cell	26
`col`	column number, zero-based	`int`	
`cols`	column count	`int`	
`deltaCol`	column increment	`int`	
`deltaRow`	row increment	`int`	
`e`	expression in formula	Expr	27
`es`	expression array	Expr[]	
`lr`	lower right corner of area	RARef	32
`matrix`	matrix value	MatrixValue	27
`r`	row index variable	`int`	
`raref`	relative/absolute reference	RARef	32
`row`	row number, zero-based	`int`	
`rows`	row count	`int`	
`sheet`	sheet	Sheet	25
`ul`	upper left corner of area	RARef	32
`v`	value	Value	31
`vs`	value array	Value[]	
`workbook`	workbook	Workbook	25

Contents

Chapter 1

What is a spreadsheet

1.1 History

The first spreadsheet program was VisiCalc, developed by Dan Bricklin and Bob Frankston in 1979 for the Apple II computer [10, 94]. A version for MS-DOS on the IBM PC was released in 1981; the size of the executable was a modest 27 KB.

Many different spreadsheet programs followed, including SuperCalc, Lotus 1-2-3, PlanPerfect, QuattroPro, and many more. By now the dominating spreadsheet program is Microsoft Excel [57], whose executable weighs in at 9838 KB. Several open source spreadsheet programs exist, including Gnumeric [37] and OpenOffice Calc [67]. See also Wikipedia's entry on spreadsheets [93].

1.2 Basic concepts

All spreadsheet programs have the same visual model: a two-dimensional grid of cells. Columns are labelled with letters A, B, ..., Z, AA, ..., rows are labelled with numbers 1, 2, ..., cells are addressed by row and column: A1, A2, ..., B1, B2, ..., and rectangular cell areas by their corner coordinates, such as B2:C4. A cell can contain a number, a text, or a formula. A formula can involve constants, arithmetic operators such as `(*)`, functions such as `SUM(...)`, and most importantly, references to other cells such as `C2` or to cell areas such as `D2:D4`. Also, spreadsheet programs perform automatic recalculation: whenever the contents of a cell has been edited, all cells that directly or transitively dependent on that cell are recalculated.

Figure 1.1 shows an example spreadsheet, concerning three kinds of tools. For each tool we know the unit count (column B) and the unit weight (column C). We compute the total weight for each kind of tool (column D), the total number of tools (cell B5), the total weight of all tools (cell D5) and the average unit weight (cell C7). Moreover, in cells E2:E4 we compute the percentage the count for each kind of tool makes up of the total number of tools. Figure 1.2 shows the formulas used in these computations.

	A	B	C	D	E	F
1		Count	Weight	Total weight	Count %	
2	Crowbars	5	7	35	22.73	
3	Screwdrivers	11	2	22	50	
4	Hammers	6	3	18	27.27	
5	Sum	22		75		
6						
7		Average	3.41			
8						

Figure 1.1: Spreadsheet window showing computed results.

Modern spreadsheet programs have one further essential feature in common. A reference in a formula can be *relative* such as `C2`, or *absolute* such as `$B$5`, or a mixture such as `B$5` which is row-absolute but column-relative.

This distinction matters when the reference occurs in a formula that is copied from one cell to another. In that case, an absolute reference remains unchanged, whereas a relative reference gets adjusted by the distance (number of columns and rows) from the original cell to the cell receiving the copy. A row-absolute and column-relative reference will keep referring to the same row, but will have its column adjusted. The adjustment of relative references works also when copying a formula from one cell to an entire cell area: each copy of the formula gets adjusted according to its goal cell.

Figure 1.2 shows the formulas behind the sheet from figure 1.1. The formulas in D3:D4 are copies of that in D2, with the row numbers automatically adjusted from 2 to 3 and 4. The formula in D5 is a copy of that in B5, with the column automatically adjusted from B to D in the cell area reference. Finally, the formulas in E3:E4 are copies of the formula `=B2/$B$5*100` in E2; note how the relative row number in `B2` gets adjusted whereas the absolute row number in `$B$5` does not.

	A	B	C	D	E	F
1		Count	Weight	Total weight	Count %	
2	Crowbars	5	7	=B2*C2	=B2/$B$5*100	
3	Screwdrivers	11	2	=B3*C3	=B3/$B$5*100	
4	Hammers	6	3	=B4*C4	=B4/$B$5*100	
5	Sum	=SUM(B2:B4)		=SUM(D2:D4)		
6						
7		Average	=D5/B5			
8						

Figure 1.2: The formulas behind the spreadsheet in figure 1.1.

So far, we have viewed a spreadsheet as a rectangular grid of cells. An equally

valid view is that a spreadsheet is a graph whose nodes are cells, and whose edges (arrows) are the dependencies between cells; see figure 1.3. The two views correspond roughly to what is called the physical and logical views by Isakowitz [43].

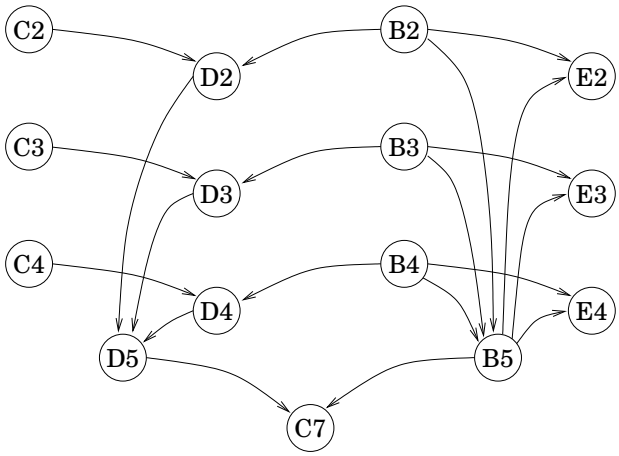

Figure 1.3: A graph-oriented view of the spreadsheet in figures 1.1 and 1.2.

1.3 Cell reference formats

Usually, cell references and cell area references are entered and displayed in the *A1 format* shown above, consisting of a column and a row indication. References are relative by default, and an absolute column or row is indicated by the dollar ($) prefix.

In Piersol's 1986 spreadsheet implementation [71] and in Excel there is an alternative format called the *R1C1 format*, in which the row number is shown followed by the column number (so the opposite of the A1 format). References are numeric for both rows and columns, and absolute by default, with relative references indicated by an offset in square brackets. When the offset is zero it is left out, so RC means "this cell".

The R1C1 format is interesting because it is the essentially the internal format of out implementation CoreCalc. The R1C1 is used in Excel's XML export format XMLSS, and Excel and Gnumeric (but apparently not OpenOffice) can optionally display formulas in R1C1 format.

The main virtue of R1C1 format is that it is invariant under the adjustment of relative cell references implied by copying of a formula. Figure 1.4 compares the two reference formats.

A1 format	R1C1 format	Meaning
A1	R[-1]C[-1]	Relative; previous row, previous column
A2	RC[-1]	Relative; this row, previous column
B1	R[-1]C	Relative; previous row, this column
B2	RC	Relative; this cell
C3	R[+1]C[+1]	Relative; next row, next column
$A$1	R1C1	Absolute; row 1, column 1 (A)
$A$2	R2C1	Absolute; row 2, column 1 (A)
$B$1	R1C2	Absolute; row 1, column 2 (B)
$B$2	R2C2	Absolute; row 2, column 2 (B)
$C$3	R3C3	Relative; row 3, column 3 (C)
$A1	R[-1]C1	Relative row (previous); absolute column 1 (A)

Figure 1.4: References from cell B2 shown in A1 format and in R1C1 format.

1.4 Formulas, functions and matrices

As already shown, a formula in a cell is an expression that may contain references to other cells, standard arithmetic operators such as (+), and calls to functions such as SUM. Most spreadsheet programs implement standard mathematical functions such as EXP, LOG and SIN, statistical functions such as MEDIAN and probability distributions, functions to generate pseudo-random number such as RAND, functions to manipulate times and dates such as NOW and TODAY, financial functions such as "present value", a conditional function IF, matrix functions (see below), and much more.

Some functions take arguments that may be a cell area reference, or range, such as D2:D4, which denotes the three cells D2, D3 and D4. In general an area reference consists of two cell references, here D2 and D4, giving two corners of a rectangular area of a sheet. The cell references giving the two corners may be any combination of relative, absolute, or mixed relative/absolute. For instance, one may enter the formula =SUM(A$1:A1) in cell B1 and copy it to cell B2 where it becomes =SUM(A$1:A2), to cell B3 where it becomes =SUM(A$1:A3), and so on, as shown in figure 1.5. The effect is that column B computes the partial sums of the numbers in column A. Moreover, since the corner references were column relative, copying column B's formulas to column C would make column C compute the partial sums of column B.

	A	B
1	0.5	=SUM(A$1:A1)
2	=A1*1.00001	=SUM(A$1:A2)
3	=A2*1.00001	=SUM(A$1:A3)
...	...	...
12288	=A12287*1.00001	=SUM(A$1:A12288)

Figure 1.5: Adjustment of cell area references when copying a formula.

Some built-in functions, called matrix functions, return an entire matrix of values rather than a number or a text string. Such functions include `TRANSPOSE`, which transposes a cell area, and `MMULT`, which computes matrix multiplication. The matrix result must then be expanded over a rectangular cell area of the same shape, so that each cell in the area receives one component (one atomic value). In Excel, Gnumeric and OpenOffice this is achieved by entering the formula as a so-called *matrix formula*. First one marks the cell area that should receive the values, then one enters the formula, and finally one types Ctrl+Shift+Enter instead of just Enter to complete the formula entry. The resulting formula is shown in curly braces, like `{=TRANSPOSE(A1:B3)}`, in every cell of the result area, although each cell contains only one component of the result. See figure 1.6 for an example.

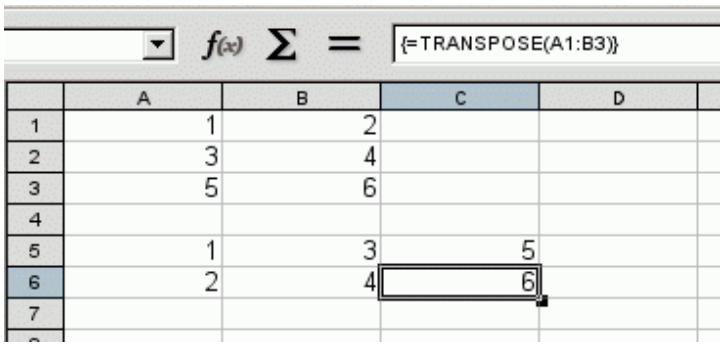

Figure 1.6: The matrix formula `{=TRANSPOSE(A1:B3)}` in result area A5:C6.

Finally, modern spreadsheet programs allow the user to define multiple related sheets, bundled in a so-called *workbook*. A cell reference can optionally refer to a cell on another sheets in the same workbook using the notation `Sheet2!A$1` in Excel and Gnumeric, and `Sheet2.A$1` in OpenOffice. Similarly, cell area references can be qualified with the sheet, as in `Sheet2!A$1:A1`. Naturally, the two corners of a cell area must lie within the same sheet.

The CoreCalc spreadsheet implementation described in chapter 2 of this report supports all the functionality described above, including built-in functions and matrix formulas.

1.5 Other spreadsheet features

Most modern spreadsheet programs furthermore provide business graphics (bar charts, pie charts, scatterplots), pivot tables, database access, spell checkers, and a large number of other useful and impressive features. Microsoft Excel'97 even contained a flight simulator, which was activated as follows: Open a new workbook; press F5; enter X97:L97 and press Enter; press Tab; press Ctrl+Shift; click the Chart Wizard button. Such features shall not concern us here.

1.6 Dependency, support, and cycles

Clearly, a central concept is the dependence of one cell on the value of another. When cell D2 contains the formula `=B2*C2` as in figure 1.2, then we say that D2 *directly depends on* cells B2 and C2, and cells B2 and C2 *directly support* cell D2. Some spreadsheet programs, notably Excel and OpenOffice, can display the dependencies using a feature called *formula audit*, as shown in figure 1.7. The arrows from cells B5 and D5 to cell C7 show that both of those cells directly support C7, or equivalently, that C7 directly depends on those two cells. In turn D5 depends on D2:D4, and so on. In fact, the formula audit in figure 1.7 simply combines the graphical view in figure 1.3 with the usual spreadsheet grid view.

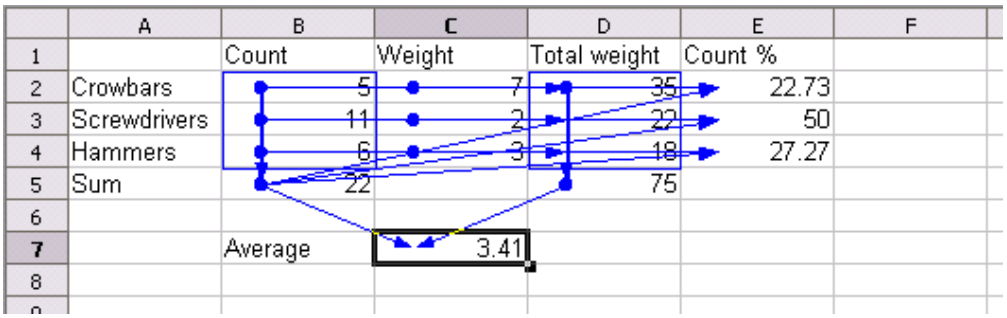

Figure 1.7: The dependencies in the sheet from figures 1.1 and 1.2.

A cell may directly depend on any number of other cells. For instance, cell B5 in figures 1.2 and 1.7 directly depends B2, B3 and B4. Similarly, a cell may directly support any number of other cells: cell B5 directly supports E2, E3 and E4.

More precisely, B5 both *statically* and *dynamically* depends on B2, B3 and B4. By static dependence we mean that the formula text in B5 refers to the cells in B2:B5, and by dynamic dependence, we mean that calculating the value of B5 requires calculating the values of those three cells.

A static dependence may or may not cause a dynamic dependence; it is an approximation of dynamic dependence. For instance, a cell containing the formula `=IF(G1<>0; G2; G3)` statically depends G1, G2 and G3, but in any given recalculation dynamically depends only on G1 and G2 or G1 and G3, according as G1 is non-zero or zero. This is because `IF` is a non-strict function; see section 1.7.4.

A cell *transitively depends on* another cell (possibly itself) if there is a non-empty chain of direct dependencies from the former to the latter. For instance, cell D5 indirectly depends on the nine cells in B2:D4. The notion of *transitive support* is defined similarly. For instance, cell B4 transitively supports B5, D4, D5, C7 and F2, F3, F4 — the latter three because they depend on B5.

If a cell statically transitively depends on itself, then there is a *static cycle* in the workbook; and if a cell dynamically transitively depends on itself, then there is a *dynamic cycle*. Sections 1.7.6 and 4.7 have more to say about cycles.

1.7 Recalculation

When the contents of a cell is changed by editing it, all cells supported by that cell, whether in the same sheet or another sheet in the workbook, must be recalculated. This happens relatively frequently, although hardly more than once every 2 seconds when a human edits the sheet. Recalculations may happen far more frequently when the cell is edited by a numerical zero-finding routine such as `GOAL.SEEK` or a numerical optimization routine such as `SOLVER`.

1.7.1 Recalculation order

Recalculation should be *completed* in dependency order: If cell B2 depends on cell A1, then the evaluation of A1 should be completed before the evaluation of B2 is completed. However, recalculation can be *initiated* in bottom-up order or top-down order.

In *bottom-up* order, recalculation starts with cells that do not depend on any other cells, and always proceeds with cells that depend only on cells already computed.

In *top-down* order, recalculation may start with any cell. When the value of an as yet uncomputed cell is needed, then that cell is computed, and when that computation is completed, the computation of the original cell is resumed. The subcomputation may recursively lead to further subcomputations, but will terminate unless there is a dynamic cyclic dependency. The current CoreCalc implementation uses top-down recalculation; see chapter 2.

1.7.2 Requirements on recalculation

The design of the recalculation mechanism is central to the efficiency and reliability of a spreadsheet implementation, and the design space turns out to be large. First let us consider the requirements on a recalculation after one cell has been edited, which is the most frequent scenario:

- Recalculation should be correct. After a recalculation the contents of all cells should be consistent with each other (in the absence of dynamic cycles).
- Recalculation should be efficient in time and space. The time required for a recalculation should be at most linear in the total size of formulas in the sheet, and ideally it should be linear in the size of formulas in those cells supported by the cells that have changed, which is potentially a much smaller number. Also, supporting data structures should require space that is at most linear in the total size of formulas in the workbook. See section 1.7.3.
- Recalculation should accurately detect dynamic cycles; see section 1.7.6.
- Recalculation should avoid evaluating unused arguments of non-strict functions such as `IF(e1; e2; e3)` and should evaluate volatile functions such as `NOW()` and `RAND()`; see sections 1.7.4 and 1.7.5.

1.7.3 Efficient recalculation

One way to ensure that recalculation takes time at most linear in the total size of formulas, is to make sure that each formula and each matrix formula is evaluated at most once in every recalculation. This is rather easy to ensure: visit every active cell and evaluate its formula if not already evaluated, recursively evaluating any supporting cells. This is the approach taken in CoreCalc, which evaluates every formula exactly once in each recalculation, using extra space (for the recursion stack) that is at most linear in the total size of formulas.

It is possible but surprisingly complicated to do better than this, as discussed in section 3.3 and chapter 4.

1.7.4 Non-strict functions

Most built-in functions in spreadsheet programs are strict: They require all their arguments to be evaluated before they are called. But the function `IF(e1; e2; e3)` is *non-strict*, as it evaluates at most one of `e2` and `e2`. For instance, the function call `IF(A2<>0; 1/A2; 1)` evaluates its second operand `1/A2` only if `A2` is non-zero.

It is straightforward to implement non-strict functions: simply postpone argument evaluation until it is clear that the argument is needed. However, the existence of non-strict functions means that a static cyclic dependency may turn out to be harmless, and it complicates the use of topological sorting to determine a safe recalculation order. See section 3.3.3.

1.7.5 Volatile functions

Furthermore, some functions are *volatile*: Although they take no arguments, different calls typically produce different values. Typical volatile functions are `NOW()` which returns the current time, and `RAND()` which returns a random number. Both are easy to implement, but complicate the use of explicit dependency information of to control recalculation order. See sections 3.3.1 and 3.3.2.

1.7.6 Dependency cycles

The existence of non-strict functions has implications for the presence or absence of cycles. Assume that cell `A1` contains the formula `IF(A2<>0; A1; 1)`. Then it would seem that there is a cyclic dependence of `A1` on `A1`, but that is the case only if `A2` is non-zero — only those arguments of an `IF`-function that actually get evaluated can introduce a cycle.

This is how Excel and OpenOffice work. They report a cyclic dependency involving the argument of a non-strict functions only if the argument actually needs to be evaluated. Strangely, Gnumeric does not appear to detect and report cycles at all, whether involving non-strict functions or not.

1.8 Spreadsheets are dynamically typed

Spreadsheet programs distinguish between several types of data, such as numbers, text strings, logical values (Booleans) and matrices. However, this distinction is made dynamically, in the style of Scheme [49], rather than statically, in the style of Haskell [41] or Standard ML [58].

For instance, the formula `=TRANSPOSE(IF(A1>0; B1:C2; 17))` is perfectly OK so long as `A1>0` is true, so that the argument to `TRANSPOSE` is a matrix-shaped cell area, but evaluates to a matrix of error values `#ARGTYPE` if `A1>0` is false.

Similarly, it is fine for cell D1 to contain the formula `=IF(A1>0; 42; D1)` so long as `A1>0` is true, but if `A1>0` is false, then there is a cyclic dependency in the sheet evaluation.

1.9 Spreadsheets are functional programs

The recalculation mechanism of a spreadsheet program is in a sense dual to that of lazy functional languages such as Haskell [41]. In a lazy functional language, an intermediate expression is evaluated only when there is a demand for it, and its value is then cached so that subsequent demands will use that value.

In a spreadsheet, a formula in a cell is (re)calculated only when some cell on which it depends has been recalculated, and its value is then cached so that all cells dependent on it will use that value.

So calculation in a lazy functional language is driven by *demand for output*, or backwards, whereas (re)calculation in a spreadsheet is driven by *availability of input*, or forwards.

The absence of assignment, destructive update and proper recursive definitions implies that there are no data structure cycles in spreadsheets. All cyclic dependencies are computational and are detected by the recalculation mechanism.

Spreadsheet programs have been proposed that are lazy also in the above sense of evaluation being driven by demand for output; see Nuñez's [64], and Du and Wadge [27], who call this *eductive evaluation*.

1.10 Related work

Despite some non-trivial implementation design issues, the technical literature on spreadsheet implementation is relatively sparse, as opposed to the trade literature consisting of spreadsheet manuals, handbooks and guidelines. There is also a considerable scholarly literature on ergonomic and cognitive aspects of spreadsheet use [43], on risks and mistakes in spreadsheet use [32, 68] and on techniques to avoid them [74].

However, our interest here is spreadsheet implementation, and variations and extensions on the spreadsheet concept. Literature in that area includes Piersol's

1986 paper [71] on implementing a spreadsheet in Smalltalk. On the topic of recalculation, the paper hints that at first, an idea similar to update event listeners (section 3.3.1) was attempted, but was given up in favor of another mechanism that more resembles that implemented by Corecalc and described in section 2.11.

De Hoon's 1995 MSc thesis [22] and related papers [23] describe a rather comprehensive spreadsheet implementation in the lazy functional language Clean. The resulting spreadsheet is somewhat non-standard, as it uses the Clean language for cell formulas, allows the user to define further functions in that language, and supports symbolic computation on formulas. Other papers on extended spreadsheet paradigms in functional languages include Davie and Hammond's Functional Hypersheets [21] and Lisper and Malmström's Haxcel interface to Haskell [53].

Nuñez's remarkable 2000 MSc thesis [64] presents ViSSh (Visualization Spreadsheet), an extended spreadsheet system. The system is based on three ideas. First, as in Piersol's system, there is a rich variety of types of cell contents, such as graphical components; second, the functional language Scheme is used for writing formulas, and there is no distinction between values and functions; and third, the system uses lazy evaluation so recalculation is performed only when it has an impact on observable output. Among other things, these generalizations enable a spreadsheet formula to "call" another sheet as a function. The implementation seems to maintain both an explicit dependency graph and an explicit support graph. This can be very space-consuming in the presence of copies of formulas with cell area arguments, as discussed in section 3.3.2.

Wang and Ambler developed an experimental spreadsheet program called Formulate [91]. Region arguments are used instead of the usual relative/absolute cell references, and functions are applied based on the shape of their region arguments. The Formulate implementation does not appear to be publicly available.

Burnett et al. developed Forms/3 [13], which contains several generalizations of the spreadsheet paradigm. New abstraction mechanisms are added, and the evaluation mechanism is extended to react not only to user edits, but also to external events such as time passing, or new data arriving asynchronously on a stream. Forms/3 is implemented in Liquid Common Lisp and is a available (for non-commercial use) in binary form for the Sun Solaris and HP-UX operating systems, but does not appear to be available in source form.

A MITRE technical report [35] by Francoeur presents a recalculation engine ExcelComp in Java for Excel spreadsheets. The engine has an interpreted mode and a compiled mode. The approach requires that the spreadsheet does not contain any static cyclic dependencies, and it is not clear that it handles volatile functions. There is no discussion of the size of the dependency graph or of techniques for representing it compactly. The ExcelComp implementation is not available to the public [36].

Yoder and Cohn have written a whole series of papers on spreadsheets, data-flow computation, and parallel execution. Topics include the relation between spreadsheet computation, demand-driven (eductive, lazy) and data-driven (eager) evaluation, parallel evaluation, and generalized indexing notations [97]; the design of a spreadsheet language Mini-SP with matrix values and recursion (not unlike CoreCalc) and a case study solving several non-trivial computation problems [98]; and a

Generalized Spreadsheet Model in which cell formulas can be Scheme expressions, including functions, and an explicit "dependency graph" (actually a support graph as defined in section 3.3.2) is used to perform minimal recalculation and to schedule parallel execution [96, 99].

Clack and Braine present a spreadsheet paradigm modified to include features from functional programming, such as higher-order functions, as well as features from object-oriented programming, such as virtual methods and dynamic dispatch [16].

None of the investigated implementations appear to use the sharing-preserving formula representation of CoreCalc.

In addition to Yoden and Cohn's papers mentioned above, there are a few other papers on parallelization of spreadsheet computations. For instance, in his thesis [90], Wack investigates how the dependency graph can be used to schedule parallel computation.

Field-programmable custom hardware for spreadsheet evaluation has been proposed by Lew and Halverson [51]. Custom circuitry realizing a particular spreadsheet's formula is generated at runtime by configuring an FPGA (field-programmable gate array) chip attached to a desktop computer. This can be thought of as an extreme form of runtime code generation. As an added benefit it ought to be possible to perform computations in parallel; spreadsheets lends themselves well to parallelization because of a fairly static dependency structure.

A paper [85] by Stadelmann describes a spreadsheet paradigm that uses equational constraints (as in constraint logic programming) instead of unidirectional formulas. Some patents and patent applications (numbers 168 and 220) propose a similar idea. This seriously changes the recalculation machinery needed; Stadelmann used Wolfram's Mathematica [95] tool to compute solutions.

A spreadsheet paradigm that computes with intervals, or even interval constraints, is proposed by Hyvönen and de Pascale in a couple of papers [24, 1, 42].

The interval computation approach was used in the PhD thesis [7] of Ayalew as a tool for testing spreadsheets: Users can create a "shadow" sheet with interval formulas that specify the expected values of the real sheet's formulas.

Burnett and her group have developed several methods for spreadsheet testing, in particular the Wysiwyt or "What You See Is What You Test" approach [14, 75, 76, 77, 33], within the EUSES consortium [82]. This work is the subject also of patents 144 and 145, listed in appendix A.

Several researchers have recently proposed various forms of type systems for spreadsheets, usually to support units of measurements so that one can prevent accidental addition of dollars and yen, or of inches and kilograms. Some notable contributions: Erwig and Burnett [30]; Ahmad and others [5]; Antoniu and others [6]; Coblenz [17]; and Abraham and Erwig [2, 4].

1.11 Online resources and implementations

The company Decision Models sells advice on how to improve recalculation times for Excel spreadsheets, and in that connection provides useful technical information on Excel's implementation on their website [26]; see section 3.3.5.

There are quite a few open source spreadsheet implementations in addition to the modern comprehensive implementations Gnumeric [37] and OpenOffice Calc [67], already mentioned. A Unix classic is `sc`, originally written by James Gosling and now maintained by Chuck Martin [54], and the several descendants of `sc` such as `xspread`, `slsc` and `ss`. The user interface of `sc` is text-based, reminiscent of VisiCalc, SuperCalc and other DOS era spreadsheet programs.

A comprehensive and free spreadsheet program is Abykus [81] by Brad Smith. This program is not open source, and presents a number of generalizations and deviations relative to the mainstream (Excel, OpenOffice and Gnumeric).

One managed code open source spreadsheet program is Vincent Granet's XXL [38], written in STk, a version of Tk based on the Scheme programming language. Another one, currently less developed, is Einar Pehrson's CleanSheets [69], which is written in Java. More spreadsheet programs — historical, commercial or open source — are listed on Chris Browne's spreadsheet website [11], with historical notes connecting them. Another source of useful information is the list of frequently asked questions [79] from the Usenet newsgroup `comp.apps.spreadsheets`, although the last update was in June 2002. The newsgroup itself [88] seems to be devoted mainly to spreadsheet application and does not appear to receive much traffic.

A number of commercial closed source managed code implementations of Excel-compatible spreadsheet recalculation engines, graphical components and report generators exist. Two such implementation are Formula One for Java [72] and SpreadsheetGear for .NET [84]; the lead developer for both is (or was) Joe Erickson. Such implementations are typically used to implement spreadsheet logic on servers without the need to reimplement formulas and so on in Java, C# or other programming languages.

Spreadsheet implementation is frequently used to illustrate the use of a programming language or software engineering techniques; for instance, that was the original goal of the above-mentioned XXL spreadsheet program. A very early example is the MicroCalc example distributed in source form with Borland Turbo Pascal 1.0 (November 1983), still available at Borland's "Antique Software" site [9]. A more recent example is the spreadsheet chapter in John English's Ada95 book [29, chapter 18]; however, this is clearly not designed with efficiency in mind.

1.12 Spreadsheet implementation patents

The dearth of technical and scientific literature on spreadsheet implementation is made up for by the great number of patents and patent applications. Searches for such documents can be performed at the European Patent Office's Espacenet [66] and the US Patents and Trademarks Office [87]. A search for US patents or

patent applications in which the word "spreadsheet" appears in the title or abstract currently gives 581 results. Appendix A lists several hundred of these that appear to be concerned with the *implementation* rather than the *use* of spreadsheets.

Some patents of interest are:

- Harris and Bastian at WordPerfect Corporation have a patent, number 223 in appendix A, on a method for "optimal recalculation", further discussed in section 3.3.7.
- Roger Schlafly has two patents, numbers 194 and 213 in appendix A, that describe runtime compilation of spreadsheet formulas to x86 code. A distinguishing feature is clever use of the math coprocessor and the IEEE754 double-precision floating-point number representation, and especially NaN values, to achieve very fast formula evaluation.
- Bruce Cordel and others at Microsoft have submitted a patent application, number 24 in appendix A, on multiprocessor recalculation of spreadsheet formulas. It includes a description of the uniprocessor recalculation model that agrees with that given by La Penna [50], summarized in section 3.3.5.

In fact, in one of the first software patent controversies, several major spreadsheet implementors were sued (not by the inventors) in 1989 for infringing on US Patent No. 4,398,249, filed by Rene K. Pardo and Remy Landau in 1970 and granted in 1983 [48]. The patent in question appears to contain no useful contents at all. The United States Court of Appeals for the Federal Circuit in 1996 upheld the District Court's ruling that the patent is unenforceable [86].

A surprising number of patents and patent applications claim to have invented compilation of a spreadsheet formula to more traditional kinds of code, similar to the compiled-mode version of Francoeur's implementation [35] mentioned above:

- Schlafly's patents (numbers 194 and 213 in appendix A) describe compilation to x86 machine code.
- Khosrowshahi and Woloshin's patent (number 141) describes compilation to a procedural programming language.
- Chovin and Chatenay's patent (number 133) describes compilation of a graph of formulas to code for embedded systems.
- Rank and Pampuch's patent application (number 132) describes compilation for PDAs.
- Rubin and Smialek's patent application (number 101) describes compilation to Java source code.
- Waldau's patent application (number 82) describes compilation to a different platform, such as a mobile phone.

- Jager and Rosenau's patent application (number 45) describes compilation to Java source code with special application to database queries.
- Tanenbaum's patent applications (number 16 and 46) describe compilation to C source code.

Chapter 2

CoreCalc implementation

This chapter describes the CoreCalc spreadsheet core implementation, focusing on concepts and details that may be useful to somebody who wants to modify it.

2.1 Definitions

Here we define the main CoreCalc concepts in the style of Landin and Burge. A UML-style summary is given in figure 2.1.

- A *workbook* of class Workbook (section 2.3) consists of a collection of sheets.
- A *sheet* of class Sheet (section 2.4) is a rectangular array, each of whose elements may contain null or a cell.
- A non-null *cell* of abstract class Cell (section 2.5) may be
  - a constant floating-point number of class NumberCell
  - or a constant text string of class TextCell
  - or a formula of class Formula
  - or a matrix formula of class MatrixFormula

  A cell could also specify the formatting of contents, data validation criteria, background colour, and other attributes, but currently does not.
- A *formula* of class Formula (section 2.5) consists of
  - a non-null expression of class Expr to produce the cell's value
  - and a cached value of class Value
  - and a workbook reference of class Workbook
  - and an `uptodate` field and a `visited` field, both of type `bool`.

- A *matrix formula* of class MatrixFormula (section 2.5) consists of
  - a non-null cached matrix formula of class CachedMatrixFormula
  - and a cell address of struct type CellAddr
- A *cached matrix formula* of class CachedMatrixFormula (section 2.5) consists of
  - a formula of class Formula
  - and the address, as a pair (c,r), at which that formula was entered
  - and the corners $(\texttt{ulCa},\texttt{lrCa})$ of the rectangle of cells sharing the formula
- An *expression* of abstract class Expr (section 2.6) may be
  - a floating-point constant of class NumberConst
  - or a constant text string of class TextConst
  - or a static error of class Error
  - or a cell reference of class CellRef (an optional sheet and a relative/absolute reference)
  - or an area reference of class CellArea (an optional sheet and two relative/absolute references)
  - or an application (call) of an operator or function, of class FunCall.
- A *value* of abstract class Value (section 2.7) is produced by evaluation of an expression. A value may be
  - a floating-point number of class NumberValue
  - or a text string of class TextValue
  - or an error value of class ErrorValue
  - or a matrix value of class MatrixValue.
- An *atomic value* is a NumberValue or a TextValue.
- A *matrix value* of class MatrixValue consists of a rectangular array of values of class Value, some of which may be null.
- A *raref* or relative/absolute reference of class RARef (section 2.8) is a four-tuple (`colAbs`, `col`, `rowAbs`, `row`) used to represent cell references `A1`, `$A$1`, `$A1`, `A$1`, and area references `A1:$B2` and so on in formulas. If the `colAbs` field is true, then the column reference `col` is absolute (`$`), otherwise relative (non-`$`); and similarly for rows.
- A *cell address* of struct type CellAddr (section 2.10) is the absolute, zero-based location (col,row) of a cell in a sheet.
- A *function* of class Function (section 2.13) represents a built-in function such as `SIN` or a built-in operator such as `(+)`.

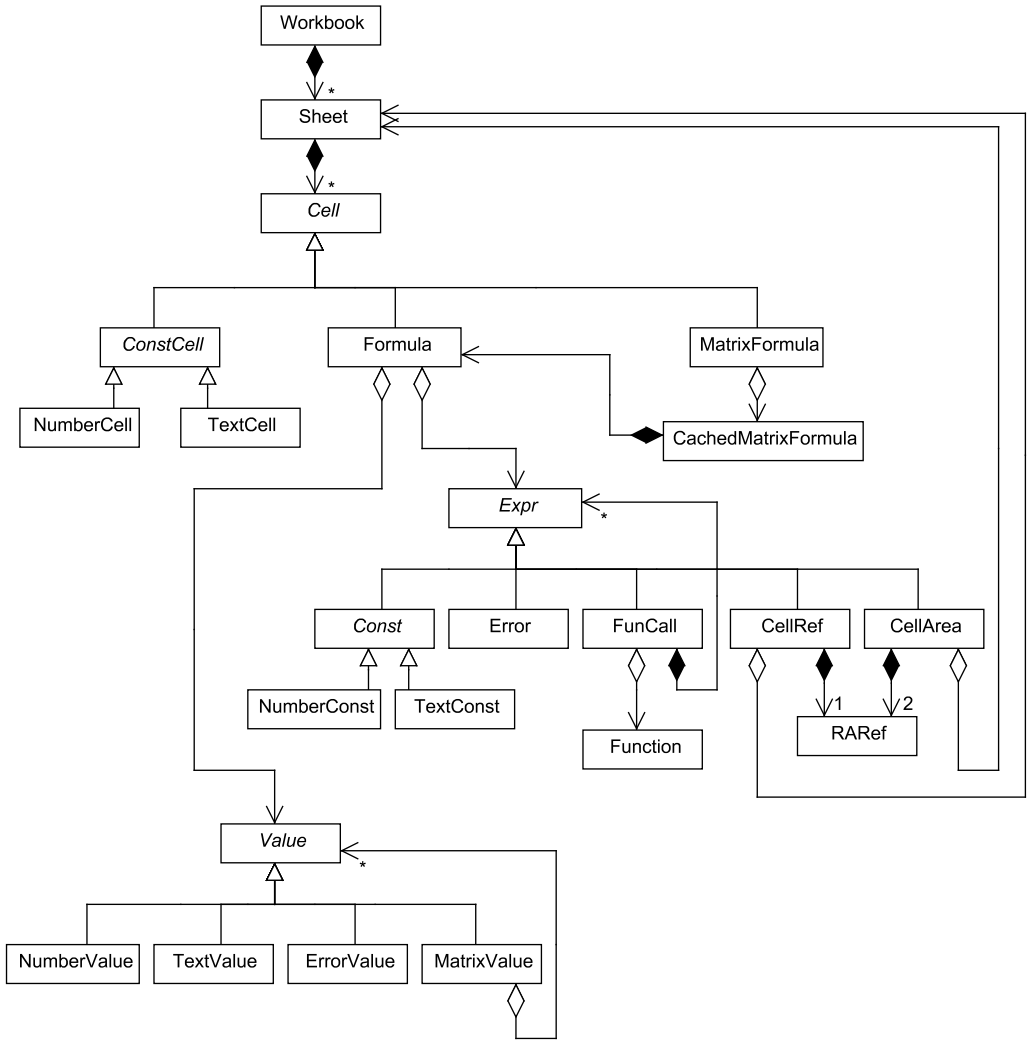

Figure 2.1: Class diagram for CoreCalc. A triangular arrow denotes *inheritance*, with the arrow pointing at the base class, as seen in the three class hierarchies deriving from abstract classes Cell, Expr and Value. An arrow originating in an open rhombus denotes *aggregation*: the instance at the rhombus end has zero or more references to instances at the other end, though possibly shared with other instances. An arrow originating in a solid rhombus denotes *composition*: the instance at the rhombus end has zero or more references to instances at the other end, accessible only from the instance at the rhombus end.

2.2 Syntax and parsing

2.2.1 CoreCalc cell contents syntax

The syntax of CoreCalc cell contents is very similar to that of Excel, Gnumeric and OpenOffice:

```
Expr ::=
   Expr == Expr
 | Expr <> Expr
 | Expr < Expr
 | Expr <= Expr
 | Expr > Expr
 | Expr >= Expr
 | Expr & Expr
 | Expr + Expr
 | Expr - Expr
 | Expr * Expr
 | Expr / Expr
 | Expr ^ Expr
 | Raref
 | Raref : Raref
 | Sheetref
 | Number
 | " String "
 | ( Expr )
 | Call

Sheetref ::=
   Name ! Raref
 | Name ! Raref : Raref

Raref ::=
   Column Row
 | $ Column Row
 | Column $ Row
 | $ Column $ Row
 | R Offset C Offset

Offset ::=
   <empty>
 | Uint
 | [ Int ]

Call ::=
   Name ( Exprs )

Exprs ::=
   Expr
 | Expr ; Exprs

CellContents ::=
   Number
 | ' String
 | " String "
 | = Expr
```

Above, `Number` is a floating-point constant; `String` is a sequence of characters; `Name` is a legal function or sheet name; `Column` is a column name A, B, ...; `Row` is a row number 1, 2, ...; `Uint` is a non-negative integer; and `Int` is an integer.

There is no special syntax for matrix formulas. As in Excel and OpenOffice, such formulas are written as ordinary formulas, and then completed with the special incantation Ctrl+Shift+Enter.

2.2.2 Formula parsing

The above grammar has been rewritten to produce a scanner and parser specification for the CoCo/R generator of recursive descent parsers [61]. Mostly the rewrite has been necessary to give operators the correct associativity and precedence, while avoiding left recursive grammar productions. All operators are left associative, even the exponentiation operator (^), just as in Excel and OpenOffice. The resulting parser builds and returns the abstract syntax tree as a Cell object. This is pretty straightforward, but the following things must be considered:

- When parsing a formula we must know the workbook that contains it, and the cell address at which it was entered. Otherwise relative cell references and area references, and sheet-absolute ditto, cannot be resolved to the abstract syntax that we use.
- The CoCo/R scanner apparently does not support the definition of overlapping token classes, such as column (`[a-zA-Z]+`) and identifier (`[a-zA-Z][a-zA-Z0-9]*`).

  This complicates the notation for calls to functions, such as `LOG10`, whose name looks like a cell reference. This is not a problem in Excel 2003, Gnumeric and OpenOffice 2, in which the last column name is `IV`, corresponding to column number 256.

2.3 Workbooks and sheets

A workbook of class Workbook contains zero or more sheets, organized as a list of non-null Sheet references, where no two references refer to the same Sheet object.

Notable methods on class Workbook include:

- `void` **`AddSheet`**`(Sheet sheet)` adds `sheet` at the end of the workbook.
- `Sheet` **`this`**`[String name]` returns the named sheet.
- `void Recalculate()` initiates a recalculation of all active cells in all sheets of the workbook.

2.4 Sheets

A Sheet contains a rectangular array of cells (type Cell[,]) each element of which may be null, representing an inactive cell, or non-null, representing an active cell. No two cell references from the same sheet or from different sheets can refer to the same Cell object.

Notable methods on class Sheet include:

- `Cell` **`InsertCell`**`(String text, CellAddr ca)` parses `text` to a cell, stores it at position `ca` in the sheet, and returns the cell.
- `void` **`InsertMatrixFormula`**`(Cell cell, int col, int row, CellAddr ulCa, CellAddr lrCa)` creates as CachedMatrixFormula from `cell`, which must be a Formula, and stores MatrixFormula objects in the cells in the area with corners `ulCa` and `lrCa`, all sharing the same CachedMatrixFormula.
- `void` **`InsertRowCols`**`(int R, int N, bool doRows)` inserts `N` new rows (or columns) before row (or column) `R >= 0` in this sheet, and adjusts all referring formulas in this sheet and other sheets by calling `InsertRowCols` on active cells. Performs row insertion if `doRows` is true; otherwise performs column insertion. See section 2.16.

- `void` **`MoveCell`**`(int fromCol, int fromRow, int col, int row)` move the cell contents in cell (`fromCol`,`fromRow`) to cell (`col`, `row`).
- `void` **`PasteCell`**`(Cell cell, CellAddr ca, int cols, int rows)` pastes or copies `cell`, which must be a formula or constant, to the cell area that has upper left-hand corner (`ca.col`, `ca.row`), and `cols` columns and `rows` rows. If `cell` is a formula, all the resulting Formula objects will be distinct but will share the same underlying Expr object.
- `void` **`PasteCell`**`(Cell cell, CellAddr ca)` pastes or copies `cell`, which must be a formula or constant, to the cell address `ca`. If `cell` is a formula, then the new cell has its own Formula object, but shares `cell`'s underlying Expr object.
- `void` **`Recalculate`**`()` initiates a recalculation of all active cells in the sheet.
- `void` **`Reset`**`()` calls `Reset()` on every active cell in the sheet.
- `void` **`ShowAll`**`(Shower show)` calls `show(col, row, val)` for every active cell in the sheet, passing its column, row and value.
- `String` **`Show`**`(int col, int row)` returns a string representing the Cell contents at position (`col`,`row`).
- `String` **`ShowValue`**`(int col, int row)` returns a string representing the value (if any) in the cell at position (`col`,`row`).
- `Cell` **`this`**`[int col, int row]` gets or sets the cell at position (`col`,`row`) in the sheet.
- `Cell` **`this`**`[CellAddr ca]` gets or sets the cell at position in the sheet.

2.5 Cells, formulas and matrix formulas

A cell in a sheet may contain an object of abstract type Cell, which has concrete subclasses NumberCell, TextCell, Formula and MatrixFormula; see figure 2.1.

Abstract class Cell has the following significant methods:

- `Value` **`Eval`**`(Sheet sheet, int col, int row)` evaluates the cell's contents, and all cells that it depends on, and marks the cell up to date, unless already up to date; then returns the cell's value.
- `void` **`InsertRowCols`**`(Dictionary<Expr,Adjusted<Expr>> adjusted, Sheet modSheet, bool thisSheet, int R, int N, int r, bool doRows)` adjusts the formula in this cell, originally in row (or column) `r`, after insertion of `N` new rows (or columns) before row (or column) `R >= 0`. Performs row insertion if `doRows` is true; otherwise performs column insertion. See section 2.16.
- `Cell` **`MoveContents`**`(int deltaCol, int deltaRow)` returns a new cell object, resulting from moving the given cell by (`deltaCol`, `deltaRow`).

- `static Cell` **`Parse`**`(String text, Workbook workbook, int col, int row)` parses `text` to a cell within the given workbook and assuming the cell's position is (`col`, `row`).
- `void` **`Reset`**`()` resets the cell's `visited` and `uptodate` flags, if any; see section 2.11.
- `String` **`Show`**`(int col, int row, Format fo)` shows the cell's contents (nothing, constant, formula, matrix formula).
- `String` **`ShowValue`**`(Sheet sheet, int col, int row)` returns a string displaying the cell's value, if necessary computing it first.

A floating-point constant is represented by a NumberCell object, and a text constant is represented a TextCell object.

An ordinary number-valued or text-valued formula is represented by a Formula object and is basically an expression together with machinery for caching its value, once computed. Thus a formula contains a non-null expression of class Expr, a cached value of class Value, flags `uptodate` and `visited` to control recalculation (see section 2.11), and a reference to the containing workbook. The latter serves to resolve absolute sheet references within the expression.

Whereas a given Formula object shall not be reachable from multiple distinct Cell[,] elements, an Expr object may well be reachable from many distinct Formula objects. In fact, it is a design objective of CoreCalc to achieve such sharing of Expr objects; see section 2.8.

A matrix formula computes a matrix value, that is, a rectangular array of values. This result must be expanded over a rectangular cell area of the same shape as the matrix value, so that each cell in the area receives one component (one ordinary value) from the matrix value, just as in Excel and OpenOffice. A MatrixValue is a cell entry that represents one cell's component of the matrix. Hence a MatrixValue object in a sheet cell contains two things: a non-null reference to a CachedMatrixFormula object shared among all cells in the cell area, and that sheet cell's (col, row) location within the cell area. The shared CachedMatrixFormula contains a Formula, whose expression must evaluate to a MatrixValue, as well as an indication of the cell area's location within the sheet.

The evaluation of one cell in the matrix formula's cell area will evaluate the underlying shared Formula once and cache its value (of type MatrixValue) for use by all cells in the cell area.

2.6 Expressions

The abstract class Expr has concrete subclasses NumberConst, TextConst, CellRef, CellArea, FunCall; see figure 2.1. Expressions are used to recursively construct composite expressions, and ultimately, formulas, but whereas a formula caches its value, an expression itself does not.

Class Expr has the following abstract methods:

- `void` **`Apply`**`(Sheet sheet, int col, int row, Act<Value> act)` applies `act` to the result of evaluating this expression at cell address `sheet[col, row]`, where `sheet` must be non-null. For a cell area expression, it applies `act` recursively to the result of evaluating each cell in that area. For any other matrix valued expression, it applies `act` to each value in the matrix. This is used to evaluate functions whose arguments are cell areas; see sections 2.13.4 and 2.13.5.

- `Value` **`Eval`**`(Sheet sheet, int col, int row)` returns the result of evaluating this expression at cell address `sheet[col, row]`, where `sheet` must be non-null.

- `Expr` **`Move`**`(int deltaCol, int deltaRow)` returns a new Expr in which relative cell references have been updated as if the containing cell were moved, not copied, by (`deltaCol`, `deltaRow`); see section 2.15.

- `Adjusted<Expr>` **`InsertRowCols`**`(Sheet modSheet, bool thisSheet, int R, int N, int r, bool doRows)` returns an expression, originally in row (or column) `r`, adjusting its references after insertion of `N` new rows (or columns) before row (or column) `R >= 0`. Performs row insertion if `doRows` is true; otherwise performs column insertion. See section 2.16.

- `String` **`Show`**`(int col, int row, int ctxpre, Format fo)` returns a string resulting from prettyprinting the expression in a fixity context `ctxpre` and with formatting options `fo`; see section 2.18.

2.6.1 Number constant expressions

A NumberConst represents a floating-point constant such as `3.14` in a formula. A NumberConst object encapsulates the number, represented as a NumberValue (section 2.7). Its `Eval` method returns that value; its `Apply` method applies `act` to it:

```
class NumberConst : Const {
  private readonly NumberValue value;
  public NumberConst(double d) {
    value = new NumberValue(d);
  }
  public override Value Eval(Sheet sheet, int col, int row) {
    return value;
  }
  public override void Apply(Sheet sheet, int col, int row, Act<Value> act) {
    act(value);
  }
  public override String Show(int col, int row, int ctxpre, Format fo) {
    return value.ToString();
  }
}
```

2.6.2 Text constant expressions

A TextConst represents a text constant such as "foo" in a formula and is very similar to a NumberConst, except in the way the constant is displayed:

```
class TextConst : Const {
  private readonly TextValue value;
  public TextConst(String s) {
    value = new TextValue(s);
  }
  public override Value Eval(Sheet sheet, int col, int row) {
    return value;
  }
  public override void Apply(Sheet sheet, int col, int row, Act<Value> act) {
    act(value);
  }
  public override String Show(int col, int row, int ctxpre, Format fo) {
    return "\"" + value.ToString() + "\"";
  }
}
```

2.6.3 Cell reference expressions

A CellRef represents a cell reference such as $B7; it consists of a raref (section 2.8) and, if the cell reference is sheet-absolute, a sheet reference. A cell reference is evaluated relative to a given sheet, column and row. Its evaluation involves computing the referred-to cell address ca and evaluating the formula in that cell. Its Apply method applies delegate act to the value of the referred-to cell.

```
class CellRef : Expr {
  private readonly RARef raref;
  private readonly Sheet sheet;   // non-null if sheet-absolute
  public override Value Eval(Sheet sheet, int col, int row) {
    if (this.sheet != null)
      sheet = this.sheet;
    CellAddr ca = raref.Addr(col, row);
    Cell cell = sheet[ca];
    return cell==null ? null : cell.Eval(sheet, ca.col, ca.row);
  }
  public override void Apply(Sheet sheet, int col, int row, Act<Value> act) {
    ... Just like Eval, excep that the last statement is:
    if (cell != null)
      cell.Apply(sheet, ca.col, ca.row, act);
  }
  public override String Show(int col, int row, int ctxpre, Format fo) {
    String s = raref.Show(col, row, fo);
    return sheet==null ? s : sheet.Name + "!" + s;
  }
}
```

2.6.4 Cell area reference expressions

A CellArea represents a cell area reference such as `$B7:B52` in a formula. It consists of two rarefs (section 2.8) giving the area's corner cells and, if the cell area reference is sheet-absolute, a sheet reference. A cell area is evaluated, by `Eval`, relative to a given sheet, column and row by finding the cell addresses of the upper left corner `ulCa` and lower right corner `lrCa` of the referred-to cell area, evaluating all cells in that area, and returning a MatrixValue (section 2.7) of the results. The `Apply` method instead calls itself recursively on each cell in the area.

```
class CellArea : Expr {
  private readonly RARef ul, lr;  // upper left, lower right
  private readonly Sheet sheet;   // non-null if sheet-absolute
  public override Value Eval(Sheet sheet, int col, int row) {
    CellAddr ulCa = ul.Addr(col, row), lrCa = lr.Addr(col, row);
    CellAddr.NormalizeArea(ulCa, lrCa, out ulCa, out lrCa);
    int cols = lrCa.col-ulCa.col+1, rows = lrCa.row-ulCa.row+1;
    int col0 = ulCa.col, row0 = ulCa.row;
    Value[,] values = new Value[cols,rows];
    for (int c=0; c<cols; c++)
      for (int r=0; r<rows; r++) {
        Cell cell = sheet[col0+c, row0+r];
        if (cell != null)
          values[c,r] = cell.Eval(sheet, col0+c, row0+r);
      }
    return new MatrixValue(values);
  }
  public override void Apply(Sheet sheet, int col, int row, Act<Value> act) {
          ... Just like CellArea.Eval, but the inner loop contains:
          cell.Apply(sheet, col0 + c, row0 + r, act);
  }
  public override String Show(int col, int row, int ctxpre, Format fo) {
    String s = ul.Show(col, row, fo) + ":" + lr.Show(col, row, fo);
    return sheet==null ? s : sheet.Name + "!" + s;
  }
  ...
}
```

2.6.5 Function call and operator expressions

A FunCall represents a function call such as `SIN(B7)`, or an infix operator application such as `A1+B6`, in a formula. It consists of a Function object representing the function to call, and a non-null array of argument expressions. A function call is evaluated relative to a given sheet, column and row by invoking the function's `applier` (section 2.13) on the argument expressions and sheet, column and row. The argument expressions are passed unevaluated to cater for non-strict functions such as `IF`. The `Apply` function evaluates the function call and applies `act` to its value.

The Show function displays the function call in prefix or infix notation as appropriate; see section 2.18. Section 2.13 describes the function call machinery in more detail.

```
class FunCall : Expr {
  private readonly Function function;   // Non-null
  private readonly Expr[] es;           // Non-null, elements non-null
  public override Value Eval(Sheet sheet, int col, int row) {
    return function.applier(sheet, es, col, row);
  }
  public override void Apply(Sheet sheet, int col, int row, Act<Value> act) {
    Value v = function.applier(sheet, es, col, row);
    if (v is NumberValue || v is TextValue)
      act(v);
    else if (v is MatrixValue)
      (v as MatrixValue).Apply(act);
    else if (v != null)
      throw new ArgumentException();   // Non-null but wrong type
  }
  public override String Show(int col, int row, int ctxpre, Format fo) {
    StringBuilder sb = new StringBuilder();
    int pre = function.fixity;
    if (pre == 0) { // Not operator
      ... show as F(arg1; ...; argN) ...
    } else { // Operator.  Assume es.Length is 1 or 2
      ... show as arg1+arg2 or similar ...
    }
    return sb.ToString();
  }
  ...
}
```

2.7 Runtime values

The abstract class Value has concrete subclasses NumberValue, TextValue, MatrixValue and ErrorValue as shown in figure 2.1.

A NumberValue represents a floating-point number, and has a public readonly field value containing that number. A NumberValue can also represent a logical value with 0.0 meaning false and all other numbers true. A TextValue represents a text string, and has a public readonly field value containing that string. A MatrixValue represents the value of a cell area expression or the result of a matrix formula, and has a public readonly field values of type Value[,] holding the elements of the matrix. A matrix of size 1x1 is distinct from an atomic value. An ErrorValue represents the result of an illegal operation (there are no exceptions in spreadsheets), and has a public readonly field msg of type String holding a description of the error.

2.8 Representation of cell references

Cell references should be represented so that they, and the expressions in which they appear, can be copied without change. Namely, it is common for a formula to be entered in one cell and then copied to many (even thousands) of other cells. Sharing the same expression object between all those cells would give considerable space savings. In particular, when using runtime code generation (RTCG) on expressions to speed up spreadsheet calculations, there should be as few expression instances as possible.

Hence in CoreCalc cell references and cell area references, we store absolute ($) references as absolute zero-based cell addresses, and relative (non-$) references as positive, zero or negative offsets relative to the address of the cell containing the formula. Concretely, CoreCalc uses a class RARef, short for relative/absolute reference, to represent references in formulas:

```
public sealed class RARef {
  public readonly bool colAbs, rowAbs;  // True=absolute, False=relative
  public readonly int colRef, rowRef;
  ...
  public CellAddr Addr(int col, int row) {
    return new CellAddr(this, col, row);
  }
  public String Show(int col, int row, Format fo) {
    if (fo.RcFormat)
      return "R" + RelAbsFormat(rowAbs, rowRef)
           + "C" + RelAbsFormat(colAbs, colRef);
    else {
      CellAddr ca = new CellAddr(this, col, row);
      return (colAbs ? "$" : "") + CellAddr.ColumnName(ca.col)
           + (rowAbs ? "$" : "") + (ca.row+1);
    }
  }
}
```

A raref is somewhat similar to the R1C1 reference format (section 1.3) but since we put the column number first (as in the A1 format) and use zero-based numbering, our format could be called the *C0R0 format*. Figure 2.2 shows the four basic forms of a C0R0 format reference. As a consequence of this representation, an expression must be interpreted relative to the address of the containing cell when evaluating or displaying the expression. This adds a little extra runtime cost.

We shall use the term *virtual copy* to denote a reference from a formula cell to a shared expression instance in this representation.

When an expression is moved (not copied) from one cell to another, its relative references must be updated and hence the abstract syntax tree must be duplicated; see section 2.15. But moving a formula does not increase the number of formulas, whereas copying may enormously increase the number of formulas, so it is more

C0R0 format	Meaning
CcRr	Absolute reference to cell (c,r) where $0 \leq c,r$
CcR[r]	Absolute column c, relative row offset r
C[c]Rr	Relative column offset c, absolute row r
C[c]R[r]	Relative column offset c, relative row offset r

Figure 2.2: The four basic forms of C0R0 references.

important to preserve the the formula representation when copying the formula than when moving it.

Also, when rows or columns are inserted or deleted, both relative and absolute references may have to be adjusted in a way that preserves as much sharing of virtual copies as possible; see section 2.16.

2.9 Sheet-absolute and sheet-relative references

A cell reference `Sheet1!B7` or an area reference `Sheet1!B7:D9` may refer to another sheet than the one containing the enclosing formula. This is implemented by adding a `sheet` field to CellRef and CellArea. If the field is non-null, then the reference is sheet-absolute and refers to a cell in that sheet. If the field is null, then the reference is sheet-relative and refers to a cell in the current sheet (the one containing the enclosing formula), that is, the sheet argument passed to the `Eval` method.

The `sheet` reference (or the absence of it) is preserved when copying or moving the CellRef or AreaRef from one sheet to another. Sheet-absolute references remain sheet-absolute, and sheet-relative references become references to the new sheet to which the enclosing formula gets copied.

The adjustment of column and row references is the same regardless of whether the reference is sheet-absolute or sheet-relative. Namely, a column-relative or row-relative but sheet-absolute reference presumably refers to a sheet that has a similar structure to the present one. Note that OpenOffice makes another distinction between sheet-relative and sheet-absolute references: A reference of the form `Sheet17.A1` is adjusted to `Sheet18.A1` if the formula is copied from Sheet1 to Sheet2. Excel does not support this kind of adjustment.

2.10 Cell addresses

A CellAddr represents an absolute cell address in a sheet as a pair of a zero-based column number and a zero-based row number. This is in contrast to a RARef (section 2.8) which represents cell references and cell area references in formulas. Given the column and row number of a RARef occurrence, the CellAddr constructor computes the absolute cell address that the RARef refers to:

```
public struct CellAddr {
```

```
  public readonly int col, row;
  public CellAddr(RARef cr, int col, int row) {
    this.col = cr.colAbs ? cr.colRef : cr.colRef + col;
    this.row = cr.rowAbs ? cr.rowRef : cr.rowRef + row;
  }
  public override String ToString() {
    return ColumnName(col) + (row+1);
  }
  ...
}
```

2.11 Recalculation

The value of a cell may depend on the values of other cells. Whenever any cell changes, the value of all dependent cells must be recalculated, exactly once, in some order that respects the dependencies (unless a cyclic dependency makes this impossible).

A workbook is recomputed by recomputing all its sheets in some order; a sheet is recomputed by recomputing all its cells in some order.

A formula cell caches its value to make the runtime complexity linear in the number of active cells. A matrix formula caches the value of the underlying matrix-valued expression, which is shared between all the cells that must receive some part of that matrix value.

To support recalculation and caching, each formula has two boolean flags: `visited` and `uptodate`. At the beginning of a recalculation both are set to false.

A formula is evaluated as follows:

1. If `uptodate` is true, then return the cached value.

2. Else, if `visited` is true, then the cell depends on itself; stop and report a cyclic dependency.

3. Else, set `visited` to true and evaluate the cell's expression. This will cause referred-to cells to be recomputed and may ultimately reveal circularities.

4. If the evaluation succeeds, set `uptodate` to true, cache the result value, and return it.

Hence all formulas are eventually recomputed, and when necessary they are recomputed in the order imposed by dependencies, by simple recursive calls. This may cause deep recursion if there are long dependency chains and an unfortunate order of visits is chosen. (This could be fixed as follows: If the recalculation depth exceeds some threshold, an approximate topological sort in dependency order might be performed and cells may be recomputed in that order. But that would lose the simplicity of the above scheme).

One could implement the `visited` and `uptodate` field management by explicitly resetting the `visited` and `uptodate` fields to false for every formula before a recalculation. To avoid the expense of this traversal, we introduce a field called `set`, global to a workbook, and make sure to maintain the following invariant:

> State invariant: Between recalculations, the `visited` and `uptodate` fields of every formula equals the `set` field of the workbook.

Resetting the `visited` and `uptodate` fields of all formulas to false is a simple matter of inverting the global `set` field. The possible states of a formula, and their interpretation, are:

State	Meaning
`visited != set && uptodate != set`	Awaits recalculation
`visited == set && uptodate != set`	Is being recomputed
`visited == set && uptodate == set`	Is up to date
`visited != set && uptodate == set`	(Impossible)

Since only three of the four possible states are actually used, it might be better to represent the state with an enum type having three values: AwaitsComputation, BeingComputed, UpToDate. However, that would require resetting the state of every cell from UpToDate to AwaitsComputation at the beginning of a recalculation.

Another possibility is to replace `visited` and `uptodate` fields by two (hash-based) sets during recalculation, one containing at any time the formulas for which `visited` would be true, and another containing at any time the formulas for which `uptodate` would be true. This makes the "resetting" very fast: simply discard the current sets and replace them by empty sets.

2.12 Cyclic references

The value of a cell may depend on the value of other cells, and may directly or indirectly depend on itself. The purpose of a formula's `visited` field is to discover such dependencies, stop recalculation, and report the discovery of a cycle. To maintain the state invariant after the discovery of a cycle, all formulas should have their `visited` and `uptodate` fields reset to the value of the workbook's global `set` field. Every formula is reachable from one or more cells, so it is sufficient to perform an explicit traversal of every cell of every sheet in the workbook. This cost is acceptable, because cyclic references should be rare, whereas ordinary recalculation must happen every time any cell has been edited, which happens frequently.

2.13 Built-in functions

CoreCalc built-in functions include mathematical functions (`SIN`), cell area functions (`SUM`), matrix-valued functions (`TRANSPOSE`), the conditional function (`IF`), which is

non-strict, and volatile functions (RAND). Built-in operators include the usual arithmetic operators, such as +, -, * and /.

Built-in functions and built-in operators are represented internally by objects of class Function:

```
class Function {
  public readonly String name;
  public readonly Applier applier;
  public readonly int fixity; // Non-zero: precedence of operator
  private static readonly IDictionary<String,Function> table;
  private static readonly Random rnd = new Random();
  private static readonly long basedate = new DateTime(1899, 12, 30).Ticks;
  private Function(String name, Applier applier) {
    this.name = name;
    this.applier = applier;
    table.Add(name, this);
  }
  ...
}
```

The Function class uses a hash dictionary table to map a function name such as "SIN" or an operator name such as "+" to a Function object. The rnd and basedate static fields are used in the implementation of spreadsheet functions RAND() and NOW(); see section 2.13.3.

The most important component of a Function object is a delegate applier of type Applier. This delegate takes as argument a sheet reference, an array of argument expressions, and column and row numbers:

```
public delegate Value Applier(Sheet sheet, Expr[] es, int col, int row);
```

Evaluation of a function call or operator application simply passes the argument expressions to the function's Applier delegate as shown in section 2.6.5. A family of auxiliary methods called MakeFunction in class Function can be used to create the Applier delegate for a strict function from a delegate representing the function. The following generic delegate types are used to represent non-void functions:

```
public delegate R Fun<R>();
public delegate R Fun<A1, R>(A1 x1);
public delegate R Fun<A1, A2, R>(A1 x1, A2 x2);
```

2.13.1 Strict one-argument functions

Most functions are *strict*, that is, their arguments are fully evaluated before the function is called. The applier for a strict function evaluates the argument expressions as if at cell sheet[col,row] and applies the function to the resulting argument values, each of type Value.

An applier for a strict unary function from double to double, such as SIN(), can be manufactured like this:

```
private static Applier MakeFunction(Fun<double,double> dlg) {
  return
    delegate(Sheet sheet, Expr[] es, int col, int row) {
      if (es.Length == 1) {
        NumberValue v0 = es[0].Eval(sheet, col, row) as NumberValue;
        if (v0 != null)
          return new NumberValue(dlg(v0.value));
        else
          return new ErrorValue("ARGTYPE");
      } else
        return new ErrorValue("ARGCOUNT");
  };
}
```

As can be seen, the Applier checks that exactly one argument is supplied, evaluates it, attempts to extract a NumberValue from the result, and applies the given delegate `dlg` to the `double` in the NumberValue.

This way new functions can easily be defined:

```
new Function("SIN", MakeFunction(Math.Sin));
new Function("COS", MakeFunction(Math.Cos));
new Function("ROUND", MakeFunction(Math.Round));
```

2.13.2 Other strict functions

There are similar overloads of the `MakeFunction` method for defining strict `double`-valued and `bool`-valued functions:

```
private static Applier MakeFunction(Fun<double> dlg) { ... }
private static Applier MakeFunction(Fun<double,double,double> dlg) { ... }
private static Applier MakePredicate(Fun<double,double,bool> dlg) { ... }
```

The `Fun<double>` overload is used to define argumentless functions such as `RAND()` and `NOW()`; see section 2.13.3. The `Fun<double,double,double>` overload is used to define arithmetic operators:

```
new Function("+", 6,
    MakeFunction(delegate(double x, double y) { return x+y; }));
new Function("*", 7,
    MakeFunction(delegate(double x, double y) { return x*y; }));
new Function("^", 8,
    MakeFunction(Math.Pow));
...
```

The integer arguments (6, 7, 8) indicate the operator's fixity; see section 2.18. The `MakePredicate` method is used to define comparison operators:

```
new Function("==", 4,
    MakePredicate(delegate(double x, double y) { return x == y; }));
new Function("<", 5,
    MakePredicate(delegate(double x, double y) { return x < y; }));
...
```

Further overloads of the `MakeFunction` method are used to define variable-argument but `double`-valued functions such as `SUM` in section 2.13.4, or one-argument but matrix-valued functions such as `TRANSPOSE` in section 2.13.6, or completely general strict functions such as `MMULT`:

```
private static Applier MakeFunction(Fun<Value[], double> dlg) { ... }
private static Applier MakeFunction(Fun<Value, Value> dlg) { ... }
private static Applier MakeFunction(Fun<Value[], Value> dlg) { ... }
```

2.13.3 Volatile functions

A volatile function is implemented just like any other function. For instance, the `RAND()` function can be implemented like this, where `rnd` is a field in class Function of type System.Random:

```
new Function("RAND", MakeFunction(rnd.NextDouble));
```

The `NOW()` function, which returns the number of days since 30 December 1899, can be defined as follows, where `basedate` is the number of 100 ns ticks until the base date:

```
new Function("NOW",
      MakeFunction(delegate() {
        return (DateTime.Now.Ticks-basedate)*100E-9/60/60/24;
      }));
```

The most notable aspect of volatile functions is that they cause complications in the design of the recalculation mechanism; see section 3.3.

2.13.4 Functions with multiple arguments

Functions such as `SUM`, `AVG`, `MIN` and `MAX` take multiple arguments, some of which may be simple numbers, cell references, cell areas, or matrix values, as in `SUM(A1:B4, 8)` or `SUM(MMULT(A1:B2, C1:D2))`. These are evaluated by applying a suitable action to all arguments, recursively applying it to the elements of every matrix-valued argument.

```
new Function("SUM",
    MakeFunction(delegate(Value[] vs) {
      double sum = 0.0;
      Apply(vs, delegate(double x) { sum += x; });
      return sum; })
    );
```

The general `Apply` function performs the recursive evaluation:

```
private static void Apply(Value[] vs, Act<double> act) {
  foreach (Value v in vs) {
    if (v is NumberValue)
      act((v as NumberValue).value);
    else if (v is MatrixValue)
      (v as MatrixValue).Apply(act);
    else
      throw new ArgumentException();
  }
}
```

where the `Act<T>` delegate type represents `void` functions:

```
public delegate void Act<A1>(A1 x1);
```

2.13.5 Better implementation of `SUM`, `AVG`, and the like

The previous section's implementation of `SUM`, `AVG` and similar functions works well, but is rather slow, as revealed by Thomas Iversen's measurements reported in section 5.2.2. Figure 5.7 shows that the above implementation of `SUM` can be roughly 14 times slower than Excel when applied to large cell area arguments. The reason is that the evaluated cell area argument is built explicitly as a matrix value and then passed to the function, but this is wholly unnecessary for functions such as `SUM`.

A slight change of the implementation brings a four-fold speedup, so that the CoreCalc base implementation is only 3.4 times slower than Excel; see the `FASTSUM` Level 0 bar in figure 5.9. This better implementation simply calls the `Apply` method on each argument expression `e`, passing to it a delegate that gets called on every atomic part of the expression's value. Thus no intermediate matrix values are constructed during the evaluation:

```
new Function("FASTSUM",
    delegate(Sheet sheet, Expr[] es, int col, int row) {
      double sum = 0.0;
      foreach (Expr e in es) {
        e.Apply(sheet, col, row,
                delegate(Value v) {
                  NumberValue number = v as NumberValue;
                  if (number != null)
                    sum += number.value;
                  else
                    throw new ArgumentException();
                });
      }
      return new NumberValue(sum);
    });
```

2.13.6 Functions with matrix-valued results

Some built-in functions produce a matrix value as result. This is the case in particular for functions used in matrix formulas: matrix transposition (`TRANSPOSE`), matrix multiplication (`MMULT`), linear regression (`LINEST`), and so on. The result of such a function is a MatrixValue, which contains a two-dimensional array Value[,] of values.

For instance, function `TRANSPOSE` takes as argument one expression that evaluates to a MatrixValue argument with size $(cols', rows')$. The result is a new MatrixValue whose underlying value array sheet has size $(cols, rows)$ with $cols = rows'$ and $cols' = rows$. Element $[i, j]$ of the result array contains the value of element $[j, i]$ the given argument array:

```
new Function("TRANSPOSE",
    MakeFunction(delegate(Value v0) {
      MatrixValue v0mat = v0 as MatrixValue;
      if (v0mat != null) {
        int cols = v0mat.Rows, rows = v0mat.Cols;
        Value[,] res = new Value[cols,rows];
        for (int c=0; c<cols; c++)
          for (int r=0; r<rows; r++)
            res[c, r] = v0mat[r, c];
        return new MatrixValue(res);
      } else
        return new ErrorValue("ARGTYPE");
    }));
```

2.13.7 Non-strict functions

For a non-strict function, the Applier delegate is not created by a `MakeFunction` method but written outright. For instance, the three-argument function `IF` is defined like this:

```
new Function("IF",                  // Note: non-strict in arg 2 and 3
    delegate(Sheet sheet, Expr[] es, int col, int row) {
      if (es.Length == 3) {
        NumberValue v0 = es[0].Eval(sheet, col, row) as NumberValue;
        if (v0 != null)
          if (v0.value != 0)
            return es[1].Eval(sheet, col, row);
          else
            return es[2].Eval(sheet, col, row);
        else
          return new ErrorValue("ARGTYPE");
      } else
        return new ErrorValue("ARGCOUNT");
    });
```

There must be three argument expressions in `es`. The first one must be non-null and is evaluated to obtain a NumberValue. If the `double` contained in that value is non-zero, the second argument is evaluated by calling its `Eval` method; else the third argument is evaluated by calling its `Eval` method.

2.14 Copying formulas

The copying of formulas from one cell to one or more other cells is implemented using the Windows clipboard, which uses "Object Linking and Embedding", or OLE. For this reason, the application must run in a so-called "single-threaded apartment", which means that the application's `Main` method must have the STAThread attribute.

The clipboard can hold multiple formats at the same time, so to ease exchange with other applications, we copy to the clipboard a text representation of the cell contents, as well as the CoreCalc internal description of the cell. The internal representation of the cell is simply its: the name of the sheet from which it is copied and the cell address at which it occurs. This can lead to surprises if that particular sheet cell is edited before one pastes from the clipboard.

A seemingly more robust alternative would be to transfer the actual cell object via the clipboard by serialization (thus requiring all cell, formula and expression classes to have the Serializable attribute). However, that would lose sharing of expression abstract syntax, and in general causes mysterious problems, presumably because built-in functions use delegate objects which are not correctly deserialized.

2.15 Moving formulas

Thanks to the internal representation of references, the cell references and cell area references in a formula need not be updated when the formula is *copied* from one cell one or more other cells. However, when a formula is *moved* from one cell and to another cell, for instance by "cutting" and then "pasting" it, then references must be updated in two ways, as shown by the example in figure 2.3:

- References *from* the moved formula to other cells appear unchanged in the A1 format, but in the internal representation relative references must actually be changed, as they are stored as offsets. In the figure 2.3 example, the internal representation of cell reference `A1` changes from `R[-1]C` to `R[-2]C[-1]`.

- Cell references *to* the cell containing the formula before the move must be updated so they refer to the cell containing the formula after the move. In the example, the external as well as internal representation of the formulas in cells B1 and C1 change as a consequence of the move. Even references from other sheets in the workbook must be updated in this way. On the other hand, references to cell areas that include the formula are not updated when the

formula is moved. Thus if C1 had contained a cell area reference A2:B2, then C1 would be unaffected by the move of the formula in A2 to B3.

The second point above in particular is somewhat surprising, but is the semantics implemented by Excel, Gnumeric and OpenOffice.

	A	B	C
1	11	=A2	=$A$2
2	=A1+$A$1		
3			

	A	B	C
1	11	=B3	=$B$3
2			
3		=A1+$A$1	

Figure 2.3: Formulas before (left) and after (right) moving from A2 to B3.

The moving of formulas is only partially implemented in CoreCalc, by method `Move` on abstract class Expr and its concrete subclasses, and method `MoveContents` on abstract class Cell and its concrete subclasses. Currently we do not implement:

- The adjustment of all references that pointed to the old cell so that henceforth they point to the new cell. Also references from other sheets and from within the moved formula must be adjusted. This adjustment should preserve the sharing of the referring formulas.
- When a block of cells, all of which share the same underlying formula (due to virtual copying) is moved, one should maintain the sharing in the moved cells. This is not done currently; maybe the `visited` field can be used to implement this?

2.16 Inserting new rows or columns

It should be possible to insert additional rows into a given sheet. This must not only move, and hence change the numbering of, some rows within the given sheet, but should also update references *from* cells in that sheet and in other sheets to the moved rows. (Insertion of columns is entirely similar to insertion of rows and will therefore not be discussed explicitly here). In general, one must update references from the affected sheet as well as from other sheets.

Consider what happens when a new row 3 is inserted in the example sheet shown on the left in figure 2.4.

A row-absolute reference must be updated if it refers to a row that follows the inserted rows. In the example, this affects all the `$A$3` references in the sheet (before insertion).

A row-relative reference must be updated if the reference straddles the insertion: that is, if the referring cell precedes the insert and the referred-to cell follows the insert, or vice versa. In both cases the reference must be increased (numerically) by the number of inserted rows. In the example, this affects reference `$A3` in cell B1,

references `$A3` and `$A4` in B2, references `$A1` and `$A2` in B3, and reference `$A2` in cell B4 (before the insertion).

	A	B
1	11	`=$A$2+$A$3+$A1+$A2+$A3`
2	21	`=$A$2+$A$3+$A2+$A3+$A4`
3	31	`=$A$2+$A$3+$A1+$A2+$A3`
4	41	`=$A$2+$A$3+$A2+$A3+$A4`

	A	B
1	11	`=$A$2+$A$4+$A1+$A2+$A4`
2	21	`=$A$2+$A$4+$A2+$A4+$A5`
3		
4	31	`=$A$2+$A$4+$A1+$A2+$A4`
5	41	`=$A$2+$A$4+$A2+$A4+$A5`

Figure 2.4: Formulas before (left) and after (right) inserting new row 3.

The insertion of a row is illegal if it would split a matrix formula block; this is enforced in OpenOffice, for instance. Therefore we first check that no matrix formula straddles the insert; if one does, then the insert is rejected. To make this check, we let a cached matrix formula include the corner coordinates of the area of participating cells. The check is made by scanning all cells preceding the insert (if any), and checking that no matrix formula block in that row extends to cells following the insert.

One should avoid copying all formulas that are to be updated. That would lose the sharing of expressions carefully achieved by the representation of relative and absolute references; see section 2.8. On the other hand, a shared expression cannot simply be adjusted destructively, because a it might then be adjusted once for each cell that shares it.

Virtual formula copies near the insert may have relative references that straddle the insert and therefore require adjustment, whereas virtual copies of the same formula farther away from the insert do not have relative references that straddle the insert. Hence even virtual formula copies on the same side of the insert may need to be adjusted in different ways. The possible versions are further multiplied if a formula contains relative references with different offsets.

Figure 2.5 shows the internal representation of the formulas shown in the figure 2.4 example above. On the left hand side it can be seen that before the insertion, cells B1 and B2 contain virtual copies of the same formula, and cells B3 and B4 contain virtual copies of another formula. On the right hand side it can be seen that after the insert, no two formulas are the same internally.

> Observation 1: All virtual copies of an expression on the same row must be adjusted in the same way.

Using this observation, it is clear that sharing of copies of an expression on the same row can be obtained as follows: When processing each row, maintain a dictionary that maps old expressions to new (adjusted) expressions; if an old expression is found in the dictionary, use a virtual copy of the new expression (simply set the Expr reference in the Formula instance; formula instances are shared only in the

n	A	B
0	11	R1+R2+R[0]+R[+1]+R[+2]
1	21	R1+R2+R[0]+R[+1]+R[+2]
2	31	R1+R2+R[-2]+R[-1]+R[0]
3	41	R1+R2+R[-2]+R[-1]+R[0]

n	A	B
0	11	R1+R3+R[0]+R[+1]+R[+3]
1	21	R1+R3+R[0]+R[+2]+R[+3]
2		
3	31	R1+R3+R[-3]+R[-2]+R[0]
4	41	R1+R3+R[-3]+R[-1]+R[0]

Figure 2.5: Internal representation before and after inserting new row $R = 2$ (zero-based). References are in C0R0 format, but the C0 prefix has been omitted.

case of matrix formulas); else compute the new expression, add the entry (old,new) to the dictionary, and use a virtual copy of new.

> Observation 2: One can compute the range of rows for which the adjustment is valid, as shown by the case analysis below.

Assume that $N \geq 0$ rows are to be inserted just before row $R \geq 0$. For relative references, let δ denote the offset before adjustment and δ' the offset after adjustment.

Aa An absolute reference to row $n < R$ needs no adjustment. This (non-)adjustment is valid regardless of the row r in which the containing expression appears.

Ab An absolute reference to row $n \geq R$ must be adjusted to $n+N$. This adjustment is valid regardless of the row r in which the containing expression appears.

Raa A relative reference to row $n < R$ needs no adjustment if the containing expression appears in row $r < R$. The reference has $\delta' = \delta = n - r$ before and after the insertion.

Rab A relative reference to row $n < R$ must be adjusted (changed from $\delta = n - r$ to $\delta' = n - r - N$) if the containing expression appears in row $r \geq R$.

Rba A relative reference to row $n \geq R$ must be adjusted (changed from $\delta = n - r$ to $\delta' = n - r + N$) if the containing expression appears in row $r < R$.

Rbb A relative reference to row $n \geq R$ needs no adjustment if the containing expression appears in row $r \geq R$. The reference has $\delta' = \delta = n - r$ before and after the insertion.

In the example on the left of figures 2.4 and 2.5, case Aa applies to all the $A$2 references; case Ab applies to all the $A$3 references; case Raa applies to the $A1 and $A2 references in cells B1 and B2; case Rab applies to the $A1 and $A2 references in cells B3 and B4; case Rba applies to the $A3 and $A4 references in cells B3 and B4; and case Rbb applies to the $A3 and $A4 references in cells B1 and B2.

The cases Raa, Rab, Rba and Rbb for relative references can be translated into the following constraints on the offset $\delta = n - r$ and the containing row r:

Raa If $r < R$ and $\delta + r < R$ then no adjustment is needed. The resulting expression is valid for rows r for which $r < \min(R, R-\delta)$, that is, $r \in [0, \min(R, R-\delta)[$.

Rab If $r \geq R$ and $\delta + r < R$ then adjust to $\delta' = \delta - N$. The resulting expression is valid for rows r for which $R \leq r < R - \delta$, that is, $r \in [R, R-\delta[$.

Rba If $r < R$ and $\delta + r \geq R$ then adjust to $\delta' = \delta + N$. The resulting expression is valid for rows r for which $R - \delta \leq r < R$, that is, $r \in [R-\delta, R[$.

Rbb If $r \geq R$ and $\delta + r \geq R$ then no adjustment is needed. The resulting expression is valid for rows r for which $r \geq \max(R, R-\delta)$, that is, $r \in [max(R, R-\delta), M[$ where M is the number of rows in the sheet.

The variables R, N and r used above agree with the CoreCalc implementation of row insertion in method `InsertRowCols` in class Expr. For relative references we additionally have δ=`rowRef` and n=`r+rowRef`.

The adjustment of an entire expression is valid for the intersection of the rows for which the adjustments of each of its relative references is valid.

Note that an adjustment for a reference is valid for an entire sheet (Aa and Ab) or for a lower (Raa) or upper (Rbb) half-sheet, or for a band preceding (Rba) or a band following (Rab) the insertion. In all cases this range is a half-open interval, representable by its lower bound (inclusive) and upper bound (exclusive). The intersection of intervals is itself an interval (possibly empty, though not here), easily computed as $[\max(lower), \min(upper)[$.

Building further on Observation 1, we could maintain for each original expression a collection $[(r_1, e_1), \ldots, (r_m, e_m)]$ of ranges $r_1, \ldots, r_m$ and the adjusted versions $e_1, \ldots, e_m$ of the expression valid for each of those ranges.

But in fact, if we process the rows in increasing order, we only need to record, for each adjusted expression in the dictionary, the least row U not in its validity range. Once we reach a row r for which $r \geq U$, we recompute an adjusted expression and save that and the corresponding new U to the dictionary.

This scheme will preserve sharing of virtual copies completely within each row. However, sharing may be lost across rows, because the same adjusted version of an expression may be valid at non-contiguous row ranges of the sheet (for instance, if a row is inserted in a range of cells, each of which depends on a cell on the immediately preceding row). The reason for this small deficiency is that our case analysis above involves the row r in which the formula appears.

This could be partially alleviated by reusing the old expression whenever the adjusted one is structurally identical. A more general solution would be to use a form of hash-consing to (re)introduce sharing of expressions that turn out to be identical after adjustment.

The insertion of new rows and new columns according to the above scheme is implemented by methods called `InsertRowCols` on class Sheet, on abstract class Cell and its subclasses, on abstract class Expr and its subclasses, and on class RARef. A generic class Adjusted<T> is used to store adjusted copies of Expr and RARef objects to preserve sharing as described above.

2.17 Deleting rows or columns

Deletion of rows or columns is similar to insertion. Again we consider only deletion of rows, since deletion of columns in completely analogous. More precisely, we consider deleting $N \geq 0$ rows beginning with row $R \geq 0$, that is, deleting the rows numbered $R, R+N-1$. As in the insertion case, references *from* cells in rows following row $R+N$ on the affected sheet must be adjusted, as must references *to* those rows from any cell in the workbook. Moreover, references to the deleted rows cannot be adjusted in a meaningful way and must be replaced with a static error indication. Figures 2.6 and 2.7 show an example in the ordinary A1 reference format and in the internal C0R0 format.

	A	B
1	11	`=$A$2+$A$4+$A1+$A2+$A4`
2	21	`=$A$2+$A$4+$A2+$A3+$A5`
3	31	
4	41	`=$A$2+$A$4+$A1+$A2+$A4`
5	51	`=$A$2+$A$4+$A2+$A3+$A5`

	A	B
1	11	`=$A$2+$A$3+$A1+$A2+$A3`
2	21	`=$A$2+$A$3+$A2+#REF+$A4`
3	41	`=$A$2+$A$3+$A1+$A2+$A3`
4	51	`=$A$2+$A$3+$A2+#REF+$A4`

Figure 2.6: Formulas before (left) and after (right) deleting row 3.

n	A	B
0	11	`R1+R3+R[0]+R[+1]+R[+3]`
1	21	`R1+R3+R[0]+R[+1]+R[+3]`
2	31	
3	41	`R1+R3+R[-3]+R[-2]+R[0]`
4	51	`R1+R3+R[-3]+R[-2]+R[0]`

n	A	B
0	11	`R1+R2+R[0]+R[+1]+R[+2]`
1	21	`R1+R2+R[0]+#REF+R[+2]`
2	41	`R1+R2+R[-2]+R[-1]+R[0]`
3	51	`R1+R2+R[-2]+#REF+R[0]`

Figure 2.7: Internal representation before and after deleting row $R = 2$ (zero-based). References are in C0R0 format, but the `C0` prefix has been omitted.

The cases are analogous to those of insertion in section 2.16, with two additional cases (Ac and Rc) to handle references to cells that get deleted.

Aa An absolute reference to row $n < R$ needs no adjustment. This (non-)adjustment is valid regardless of the row r in which the containing expression appears.

Ab An absolute reference to row $n \geq R+N$ must be adjusted to $n-N$. This adjustment is valid regardless of the row r in which the containing expression appears.

Ac An absolute reference to row $R \leq n < R+N$ must be replaced by a `#REF` error indication. This adjustment is valid regardless of the row r in which the containing expression appears.

Raa A relative reference to row $n < R$ needs no adjustment if the containing expression appears in row $r < R$. The reference has $\delta' = \delta = n - r$ before and after the deletion.

Rab A relative reference to row $n < R$ must be adjusted (changed from $\delta = n - r$ to $\delta' = n - r + N$) if the containing expression appears in row $r \geq R + N$.

Rba A relative reference to row $n \geq R + N$ must be adjusted (changed from $\delta = n - r$ to $\delta' = n - r - N$) if the containing expression appears in row $r < R$.

Rbb A relative reference to row $n \geq R + N$ needs no adjustment if the containing expression appears in row $r \geq R + N$. The reference has $\delta' = \delta = n - r$ before and after the deletion.

Rca A relative reference to row $R \leq n < R + N$ from row $r < R$ must be replaced by an error indication `#REF`.

Rcb A relative reference to row $R \leq n < R + N$ from row $r \geq R + N$ must be replaced by an error indication `#REF`.

In the example on the left of figures 2.6 and 2.7, case Aa applies to all the `$A$2` references; case Ab applies to all the `$A$3` references; case Ac does not apply anywhere; case Raa applies to the `$A1` and `$A2` references in cells B1 and B2; case Rab applies to the `$A1` and `$A2` references in cells B4 and B5; case Rba applies to the `$A4` and `$A5` references in cells B1 and B2; case Rbb applies to the `$A4` and `$A5` references in cells B4 and B5; case Rca applies to the `$A3` reference in cell B2; and case Rcb applies to the `$A3` reference in cell B5.

The cases Raa, Rab, Rba, Rbb, Rca and Rcb for relative references can be translated into the following constraints on the offset $\delta = n - r$ and the referring row r:

Raa If $r < R$ and $\delta + r < R$ then no adjustment is needed. The resulting expression is valid for rows r for which $r < \min(R, R - \delta)$, that is, $r \in [0, \min(R, R - \delta)[$.

Rab If $r \geq R + N$ and $\delta + r < R$ then adjust to $\delta' = \delta + N$. The resulting expression is valid for rows r for which $R + N \leq r < R - \delta$, that is, $r \in [R + N, R - \delta[$.

Rba If $r < R$ and $\delta + r \geq R + N$ then adjust to $\delta' = \delta - N$. The resulting expression is valid for rows r for which $R + N - \delta \leq r < R$, that is, $r \in [R + N - \delta, R[$.

Rbb If $r \geq R + N$ and $\delta + r \geq R + N$ then no adjustment is needed. The resulting expression is valid for rows r for which $r \geq \max(R, R + N - \delta)$, that is, $r \in [\max(R, R + N - \delta), M[$ where M is the number of rows in the sheet.

Rca If $r < R$ and $R \leq \delta + r < R + N$ then the reference is invalid and must be replaced by `#REF`. The resulting expression is valid for rows r for which $R - \delta \leq r < \min(R, R + N - \delta)$, that is, $r \in [R - \delta, \min(R, R + N - \delta)[$.

Rcb If $r \geq R + N$ and $R \leq \delta + r < R + N$ then the reference is invalid and must be replaced by `#REF`. The resulting expression is valid for rows r for which $\max(R + N, R - \delta) \leq r < R + N - \delta$, that is, $r \in [\max(R + N, R - \delta), R + N - \delta[$.

2.18 Prettyprinting formulas

To show operators properly in infix form and without excess parentheses, we add to every Function an integer denoting its fixity and precedence. A fixity of 0 means not an infix operator, positive means infix left associative, and higher value means higher precedence (stronger binding). We could take negative to mean right associative and indicate precedence by the absolute value, but that does not seem to be needed. Even the exponentiation operator (`^`) is left associative in Excel and OpenOffice. In Gnumeric, it is right associative as is conventional in programming languages.

Then we add a parameter `ctxpre` to the `Show` method of the Expr class to indicate the context's precedence. When the function to be printed is an infix with precedence less than `ctxpre`, we must enclose it in parentheses; otherwise there is no need for parentheses. Applications of functions that are not infix are printed as `F(e1; ...; en)`. Function arguments and top-level expressions have a `ctxpre` of zero. To prettyprint `(1-2)-3` without parentheses and `1-(2-3)` with parentheses, the prettyprinter distinguishes left-hand operands from right-hand operands by increasing the `ctxpre` of right-hand operands by one.

Another parameter to the `Show` method, of type Format, controls other aspects of the display of formulas, such as whether references are shown in A1 or R1C1 format.

2.19 Graphical user interface

A simple graphical user interface has been created to facilitate experimentation with CoreCalc, as shown in figure 2.8.

The graphical user interface in implemented in files `GUI/GUI.cs` and `GUI/AboutBox.cs`. A workbook is displayed using a WorkbookForm instance, which derives from class Form in the System.Windows.Forms namespace. Apart from menus, a formula text box and a status bar, a WorkbookForm contains a TabControl that holds a SheetTab, derived from class TabPage, for each sheet in the workbook. The SheetTab contains a DataGridView control to display the sheet's cells in a rectangular grid. The DataGridView implementation from System.Windows.Forms is amazingly slow, but works well enough if you have a good graphics accelerator. We use the DataGridView without an underlying data model, and explicitly set the display contents of each cell from the CoreCalc implementation. Hence a leaner implementation of the graphical grid-of-cells component would suffice and would be welcomed.

Most CoreCalc functionality is available through menus and menu items, with associated shortcut keys; see figure 2.9.

Also, entering a cell displays the cell's formula (if any) so that it can be edited, and leaving a cell enters the edited formula (if any) into that cell or into the marked cell area. Marking a cell area and entering a formula automatically makes it a matrix formula; the incantation Ctrl+Shift+Enter is not needed.

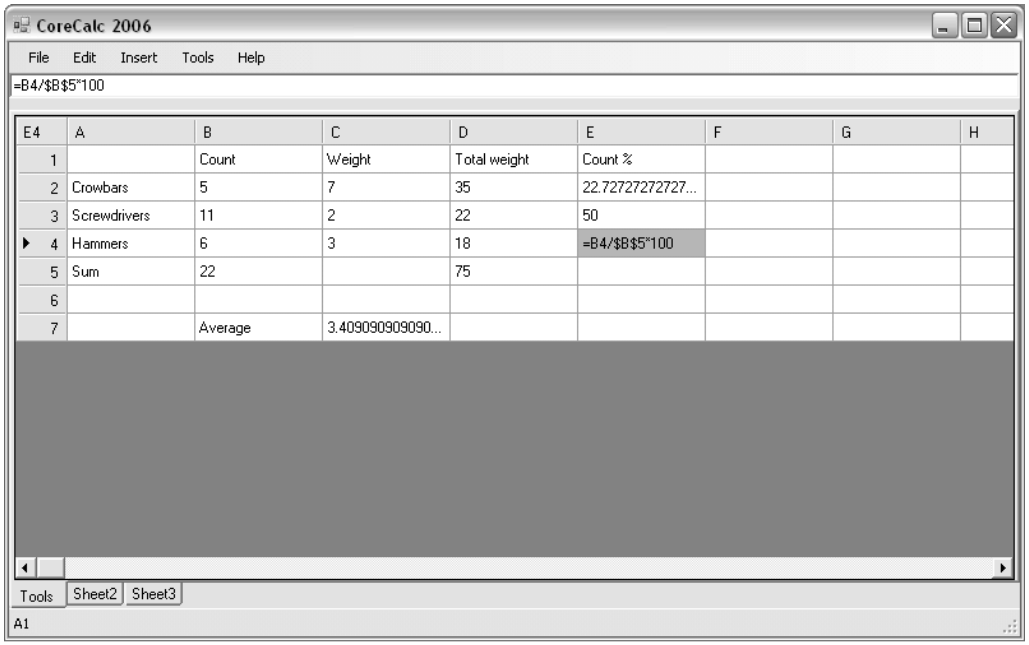

Figure 2.8: Graphical user interface for CoreCalc.

Menu	Shortcut	Operation
File ǀ New sheet	Ctrl+N	Add sheet to workbook
File ǀ Import workbook	Ctrl+O	Read new workbook from file
File ǀ Exit		Discard workbook and exit
Edit ǀ Copy	Ctrl+C	Copy cell to clipboard
Edit ǀ Cut	Ctrl+X	Cut cell to clipboard
Edit ǀ Paste	Ctrl+V	Paste from clipboard
Edit ǀ Delete	Del	Delete cell
Insert ǀ Column		Insert new column before current
Insert ǀ Row		Insert new row before current
Tools ǀ Recalculate	F9	Recalculate workbook
Tools ǀ Reference format ǀ A1		Show references in A1 format
Tools ǀ Reference format ǀ C0R0		Show references in C0R0 format
Tools ǀ Reference format ǀ R1C1		Show references in R1C1 format
Help ǀ About		Information about CoreCalc

Figure 2.9: Menu options in CoreCalc.

2.20 Reading spreadsheets in XML format

Thomas S. Iversen implemented support for reading existing spreadsheets from files in XML format, or more precisely, the export format XMLSS of Microsoft Excel 2003 and the export format of Gnumeric. Those format was chosen in preference to other formats because they are reasonably simple and documented. The Excel format moreover represents references in the R1C1 format (section 1.3), similar to the internal reference representation in CoreCalc. The implementation of import from these XML formats is found in file `IO/WorkbookIO.cs`.

In the future, it would make sense to support also some more standardized formats. Some formats of particular interest are:

- OpenDocument or OASIS Open Document Format for Office Applications (ODF) [34], which is implemented by Open Office and which has been approved as ISO and IEC international standard ISO/IEC 26300. Spreadsheet formula syntax will not be standardized until version 1.2 of this specification [92]. Even so, the standard comprises some 700 pages.

- Office Open XML (OOXML), which is implemented by Microsoft Office 2007 and which has been submitted to Ecma International TC45 for standardization [28]. The current draft standard comprises some 5400 pages, although only around 840 pages are closely related to spreadsheet representation.

2.21 Benchmarking CoreCalc

Support for performance measurements of recalculation in CoreCalc has been implemented by Thomas S. Iversen. A typical use is to build a complex spreadsheet under the control of a C# program, calling methods such as `InsertCell` and `PasteCell` from the Sheet class, and then recalculate the workbook multiple times by calling the `Recalculate` method on the Workbook class. The primary value of the benchmark environment is that it collects information about the operating system and hardware (CPU type and speed, and amount of RAM), and writes this and other information together with timing results to an XML file. This information can easily be postprocessed by various plotting programs, such as Ploticus [39]. The benchmark support is implemented by classes Benchmark, MachineInfo and TinyScript in file `BenchmarkUtils/Benchmark.cs`.

A small but complete example benchmark is found in the Benchmarks project, in directory `Benchmarks/SumFunction.cs`. Executing the `SumFunctions.bat` script from a Windows Command Prompt in that directory will run the benchmark, collect performance data in an XML file called `SumFunctions.xml`, create a Ploticus script, and run Ploticus to create bar charts in the files `SumFunctions.gif` and `SumFunctions.eps`. This assumes that CoreCalc has been built and that Ploticus is installed so that `pl.exe` is in the PATH.

2.22 Organization of source files

The source code of CoreCalc is organized as a Visual Studio "solution" called CoreCalc, which is divided into several "projects", as shown in figures 2.10 to 2.12.

File	Contents	Classes
`Benchmark.cs`	Benchmark support	Benchmark, MachineInfo, TinyScript
`CellAddressing.cs`	Cell addresses	Adjusted<T>, CellAddr, RARef
`Cells.cs`	Sheet cell contents	CachedMatrixFormula, Cell, ConstCell, Formula, MatrixFormula, NumberCell, TextCell
`Expressions.cs`	Formula expressions	CellArea, CellRef, Const, Error, FunCall, NumberConst, TextConst
`Functions.cs`	Built-in functions	Function
`Program.cs`	Main program	Program
`Sheet.cs`	Sheets of a workbook	Sheet
`Types.cs`	Auxiliary types	Act, Applier, Format, Fun, Shower, ...
`Values.cs`	Values of expressions	ErrorValue, MatrixValue, NumberValue, TextValue, Value
`Workbook.cs`	Workbooks	
`Coco/Spreadsheet.ATG`	Parser specification	
`GUI/AboutBox.cs`	An "about" dialog	AboutBox
`GUI/GUI.cs`	User interface	CopiedCell, SheetTab, WorkbookForm
`IO/WorkbookIO.cs`	Spreadsheet import	IOFormat, GnumericIOFormat, XMLSSIOFormat

Figure 2.10: Source files of the CoreCalc project, all in the CoreCalc namespace.

File	Contents	Classes
`hbars.ploticus.template.txt`	Ploticus template for bar charts	
`SumFunctions.bat`	Run benchmark and create graphs	
`SumFunctions.cs`	Benchmark `SUM` functions	SumFunctions

Figure 2.11: Source files of the Benchmarks project.

File	Contents	Classes
`Programs.cs`	From XML to Ploticus script	Data, Dataset, Ploticus, Program, Subdata

Figure 2.12: Source files of the BenchmarkToPloticus project.

Chapter 3

Alternative designs

The previous chapter presented details of the CoreCalc implementation. This chapter will review some aspects of the CoreCalc design, especially the recalculation mechanism, and explain why some seemingly plausible alternatives are difficult to implement, or unlikely to work well.

3.1 Representation of references

As described in sections 2.15 through 2.17, cumbersome adjustments of referring formulas must be performed when moving a formula from one cell to another, and when inserting or deleting rows or columns in an existing sheet.

3.1.1 Direct object references

These adjustments would be automatic if a cell reference such as `A1` was represented as a direct object reference from the abstract syntax of one formula to the abstract syntax of other formulas. However, such a representation would preclude sharing of virtual formula copies.

Alternatively, the adjustment of referring formulas described in section 2.15 would be considerably simplified if the implementation maintained explicit knowledge of which cells directly depend on the moved cell, for instance using a support graph as described in section 3.3.2. With the current implementation, a scan of the entire workbook is needed to find those cells, but cell move operations are infrequent, and the extra time required to scan the workbook is small compared to the time it takes a user to perform these operations, usually done manually.

3.1.2 Reference representation in Excel

The fact that the XML export format of Excel 2003 uses the R1C1 format (section 1.3) makes it reasonable to assume that a variant of R1C1 is the internal

reference format of Excel. However, patents 182 and 204 by Kaethler et al. indicate that formula copies are (or were) *not* shared by default in Excel, which seems to remove the main motivation for using R1C1. Also, the highly efficient formula implementation described in Schlafly's patents 194 and 213 is not directly applicable to sharable formulas, unlike Thomas Iversen's implementation of runtime code generation, briefly presented in chapter 5.

3.2 Evaluation of matrix arguments

The current CoreCalc implementation of aggregate functions such as `SUM` and `AVG` first evaluates all their arguments, and then applies a delegate to aggregate the results. This may imply wasteful allocation of large intermediate data structures, which can make CoreCalc slower than Gnumeric and OpenOffice, as shown by Thomas Iversen's experiments, summarized in section 5.2.2.

An obvious alternative is to iterate over the unevaluated cell area arguments, passing a delegate that evaluates the cells and aggregates the results in one pass, thus avoiding the allocation of data structures that simply hold intermediate results for a very short time.

3.3 Minimal recalculation

In the current CoreCalc implementation, each recalculation evaluates every formula exactly once, and follows each reference from each formula once, for a recalculation time that is linear in the sum of the sizes of all formulas. As shown in section 5.2, this provides efficiency comparable to that of several other spreadsheet implementations when all cells need to be recalculated. Still, it would be desirable to improve this so that each recalculation only considers cells that depend on some changed cell, as is possibly the case in Excel.

Several "obvious" solutions are frequently proposed in discussions:

- Update event listeners on cells; see section 3.3.1.
- Explicit representation of the support graph; see section 3.3.2 and chapter 4.
- Topological sorting of cells in dependency order; see section 3.3.3.
- Speculatively reuse evaluation order; see section 3.3.4.

In the sections below we will discuss the merits of each of these proposed mechanisms for minimal recalculation. To simplify discussion of space and time requirements, assume that only one cell has been edited before a recalculation, and let N_A be the number of non-null cells in the workbook, let F_A be the total size of formulas in the workbook, let N_D be the number of cells that depend on the changed (edited) cell in a given recalculation, and let correspondingly F_D be the total size of formulas in those cells.

3.3.1 Update event listeners

One idea that seems initially plausible is to use event listeners. For instance, if the formula in cell B2 depends on cells A1 and A2, then B2 could listen to value change events on cells A1 and A2. Whenever the value of a cell changes, a value change event is raised and can be handled by the listening cells. This makes each dependent cell an *observer* of all its supporting cells.

However, it is difficult to make this design work in practice:

- First of all, the number of event listeners may be $O(N_A^2)$, quadratic in the number of active cells. For instance, in the sheet shown in figure 5.6, the `SUM` formula in cell `B`n must have event listeners on n cells in column `A`. With N such rows, the number of event listeners is $O(N^2)$. This poses two problems: the space required to record the event handlers associated with cells (even if the handler objects themselves can be shared), and the large number of event handler calls. The space problem is by far the most severe one.

- Second, one needs a separate mechanism to determine the proper recalculation order anyway. The value change event listener cannot just initiate the recalculation of the listening cell, because the handler may be called at a time when some (other) supporting cells are not yet up to date. Hence an event handler may just record that the cell needs to be recalculated, and perhaps also that a particular supporting cell now is up to date.

- Third, a cell that contains a formula with a volatile function call must be recalculated even if the value of no supporting cell has changed. That is, one needs to keep a separate list of such cells and recalculate them whenever anything changes, or one could introduce artificial "events" on which such cells depend.

- Fourth, a dynamic cyclic dependency will cause an infinite chain of events, unless a separate cycle detection scheme is implemented.

- Fifth, event listeners would have to be attached based on static dependencies. For instance, if cell B2 contains the formula `IF(RAND()>0.5; A1; A2)`, then B2 should attach event handlers to both A1 and A2. However, a value change event on A1 may be irrelevant to B2, namely when the pseudo-random number generator `RAND()` returns a number less than or equal to 0.5. In general, the existence of non-strict functions means that some event handlers will be called to no avail.

- Finally, the lists of event handlers need to be maintained when the contents of cells are edited. This is fairly straightforward because the formula in a cell contains the necessary information about it directly supporting cells. So when a cell reference is added to or deleted from a formula, it is easy to find the cell(s) that must have event listeners added or removed.

3.3.2 Explicit support graph

A more general alternative to using event listeners is to build an explicit static *support graph*, whose nodes are sheet cells and where there is an edge from cell A1 to cell B2, say, if A1 statically supports B2, or equivalently, B2 statically depends on A1. The arrows drawn by the formula audit feature of modern spreadsheet programs essentially draw the support graph, as shown in figure 1.7.

An explicit support graph suffers from some of the same problems as the use of event listeners. In fact, systematic attachment of event listeners as described above would create precisely a support graph, where the edge from A1 to B2 is represented by A1 holding a reference to an event handler supplied by B2.

Some of the problems with using an explicit support graph are:

- First, as for event listeners, the support graph may have $O(N_A^2)$ edges when there are N_A active cells, witness the example in figure 5.6. Thus the space required to explicitly represent the support graph's edges would be excessive. But note that the dependency graph, represented by the formulas in the active cells, requires only space $O(F_A)$. The reason for this is chiefly the compact representation of sums and other formulas that take cell area arguments.

  An interesting question is whether the support graph, like the dependency graph, can be represented compactly?

- The support graph can be used to determine the proper recalculation order. When a cell has been edited, one can determine all the cells reachable from it, that is, all the cells transitively statically supported by that cell. Then one can linearize the subgraph consisting of those cells by topological sorting in time $O(F_D)$. The resulting linear order is suitable for a single pass recalculation.

- As for event listeners, one needs to keep a list of the cells containing formulas with volatile function calls, and recalculate those cells, and all cells reachable from them, at every recalculation.

- A static cyclic dependency manifests itselves as a cycle in the support graph, but a static dependency may be harmless. When there is no cycle in the support graph, there can be no dynamic cyclic dependency. When there is a cycle in the support graph, which should be rare, a separate mechanism can be used to determine whether this is also a harmful dynamic cyclic dependency. However, a static cycle would complicate the topological sorting proposed above.

- As for event listeners, the support graph would have to be based on static dependencies, with the same consequence: Some cells may be recomputed although they do not actually (dynamically) depend on cells that have changed.

- The static support graph must be maintained when the contents of cells are edited. As for event listeners, this is fairly straightforward.

In conclusion, an explicit static support graph seems more promising than event listeners, but is feasible only if a compact yet easily maintainable representation can be found. One such representation is discussed in chapter 4.

3.3.3 Topological sorting of cell dependencies

A topological sorting is a linearization and approximation of the support graph. The advantage of keeping only the topological sorting is that it requires only space $O(N_A)$ rather than space $O(N_A^2)$ for the more precise support graph. The chief additional disadvantage is that the topological sort can be very imprecise and hence is a poor basis achieving minimal recalculation. Linearizations of the dependencies in the figure 3.1 example have the form A1, A2, A3, ..., B1, B2, B3, ..., C1, C2, C3, ..., D1, D2, D3, ..., E1, E2, E3, ..., F1, F2, F3, ..., with some permutation of the blocks.

	A	B	C	D	E	F
1	11	12	13	14	15	16
2	=A1+1	=B1+1	=C1+1	=D1+1	=E1+1	=F1+1
3	=A2+1	=B2+1	=C2+1	=D2+1	=E2+1	=F2+1
...	...	...	...	...	...	...

Figure 3.1: Bad control of recalculation using topological sort.

This means that if A1 changes, then not only the cells supported by A1 will be recalculated, but also all the other cells following A1 in the topological sorting, most of them needlessly.

Building the topological sorting in the first place is not straightforward. Most simple algorithms for building the topological sorting assume a proper ordering (acyclic, that is, not just a preorder), but as shown in section 1.7.6, a spreadsheet can contain a static dependency cycle that is perfectly harmless, thanks to non-strict functions.

Rebuilding the topological sorting anew at each change to the spreadsheet is not attractive, as this requires time $O(F_A)$, in which time one can recalculate all cells, whether changed or not, anyway. Hence it is desirable to try to incrementally adapt the topological sorting as the cells of the spreadsheet are edited. There do exist on-line algorithms for maintaining topological sorts, but they are not fast. Also, simple edits to spreadsheet cells can radically change the topological sorting as shown in figure 3.2, which indicates that efficient maintenance of the topological sorting is not straightforward.

3.3.4 Speculative reuse of evaluation order

As an alternative to maintaining a correct topological sorting, or linearization, of the cell dependencies, one could simply record the actual order in which cells are recalculated, and attempt to reuse that order at the next recalculation. Maybe this is what Excel does, see section 3.3.5.

The idea should be that dependency structure changes very little, usually not at all, from recalculation to recalculation. Hence the most recent bottom-up recalculation order is likely to work next time also.

	A	B	C	D
1	11	12	=SUM(B1:B9999)	14
2		=B1+A$1		=D1+C$1
3		=B2+A$1		=D2+C$1
4		=B3+A$1		=D3+C$1
...		...		...

Topological order before edit: A1, B1, B2, ..., C1, D1, D2, ...

	A	B	C	D
1	=SUM(D1:D9999)	12	13	14
2		=B1+A$1		=D1+C$1
3		=B2+A$1		=D2+C$1
4		=B3+A$1		=D3+C$1
...		...		...

Topological order after edit: C1, D1, D2, ..., A1, B1, B2, ...

Figure 3.2: Radical change in topological order after editing A1 and C2.

This should avoid rediscovering dependency order most of the time. However, the correct amount and order of recalculation may change from recalculation to recalculation even if not cells are edited between recalculations. Again, non-strict and volatile functions are the culprits: if the sheet contains a formula such as `IF(RAND()>0.5; A1; A2)`, then the previous recalculation order will be wrong half the time.

3.3.5 Recalculation in Microsoft Excel

A paper by La Penna at the Microsoft Developer Network (MSDN) website [50] describes recalculation in Excel 2002. The paper presents an example but glosses over the handling of volatile and non-strict functions.

Recalculation is presented as a three-stage process: (1) identify the cells whose values need to be recalculated, (2) find the correct order in which to recalculate those cells, and (3) recalculate them.

The description of step (1) implies that from a given cell one can efficiently find the cells and the cell areas that depend on that cell, but the paper does not say how this is implemented. Presumably this is done using the "dependency tree" described later in this section.

The paper's advice on efficiency of user-defined functions indicate that step (2) is embedded in step (3) as follows. To recalculate a cell, start evaluating its formula. If this evaluation encounters a reference to a cell that must be recalculated, then abandon the current evaluation and start evaluating the other cell's formula. When that is finished, start over from scratch evaluating the original cell's formula. At least, that is what the advice implies for user-defined functions: "*One way to opti-*

mize user-defined functions is to prevent repeated calls to the user-defined function by entering them [the calls?] last in order in an on-sheet formula".

Interestingly, this requires a linked list (acting as a stack) to remember the yet-to-be-computed cells. This is somewhat similar to the CoreCalc design, which uses the method call stack but avoids discarding any work already done.

Another interesting bit of information is that the final recalculation order is saved and reused for the next recalculation, so that it avoids discarding partially computed results. It is not discussed how this scheme works for a formula such as `IF(RAND()>0.5; A1; A2)` that changes the dynamic dependencies in an unpredictable way. Nevertheless, such a scheme probably works well in practice.

The paper hints that usually Excel only recalculates a cell if it (transitively) depends on cells that have changed, but no explicit guarantees are given. That paper says that one can request an ordinary recalculation (recalculate only cells that transitively depend on changed ones) by pressing F9, and force a "full recalculation" (recalculate all cells, also those not depending on changed cells) by pressing Ctrl+Alt+F9, but the latter key combination does not seem to work in Excel 2003.

Other sources indicate that the methods `Calculate` and `CalculateFull` from Excel interop class ApplicationClass provide other ways to perform ordinary and full recalculation [25, 63]. In addition, the method `CalculateFullRebuild` rebuilds the so-called "dependency tree" and then performs a full recalculation; see below. These methods and the dependency tree are not mentioned in the La Penna paper. The benchmark data in section 5.2 show that rebuilding the dependency tree can increase recalculation time very considerably; by a factor of 80 in bad cases.

The patents by Schlafly (numbers 194 and 213 in appendix A) give a hint how recalculation order may be implemented in classic spreadsheet implementations. Our conjecture is: Each formula (really, cell) record has a pointer to the next formula, so that all cells together make up a linked list. The implementation attempts to keep this list ordered so that a cell always precedes any cells that depend on it. If during evaluation, a formula is found to depend on a cell that has not yet been evaluated, then the offending cell is moved from its current position in the linked list to just before the dependent formula, and evaluation starts over at the offending cell's new position in the list.

Third-party information from the company Decision Models [26] indicates that Excel 2003 does maintain a "dependency tree", probably corresponding to the support graph discussed in section 3.3.2 and chapter 4. (Note that the "tree" may in general be a graph, and may even contain cycles). The dependency tree is used to limit recalculation to those cells that depend on changed cells. However, according to the same source, in Excel 2003 there are some limitations on the representation of the dependency tree: *The number of different areas in a sheet that may have dependencies is limited to 65 536*, and *the number of cells that may depend on a single area is limited to 8192*. In Excel 2007 (Excel 12) those limits will be removed, but in versions of Excel prior to that, full recalculations will be performed rather than minimal recalculations when those limits are exceeded [25].

3.3.6 Recalculation in Gnumeric

The Gnumeric spreadsheet program [37] is open source, but we have not studied its recalculation mechanism in detail. An interesting technical note [56] by Meeks and Goldberg, distributed with the source code, discusses "the new dependency code". The purpose of that code is to find a minimal set of cells that must be recalculated when given cells have changed. Apparently two hash tables are used for individual cell dependencies, but some other form of search is needed to determine whether the value of a cell `A42` is used in a cell area reference such as `A1:A10000`.

3.3.7 Related work

Harris and Bastian has a patent, number 223 in appendix A, on a method for "optimal recalculation". The patent assumes that there is an explicit representation of the support graph (which the patent calls the *dependency set* for a cell), and then describes how to recalculate only those formulas that need to be recalculated, and in an order that respects dependencies. Basically, this combines a filtering (consider only non-uptodate cells) with transitive closure (cells that depend on non-uptodate cells are themselves non-uptodate) and topological sorting (to recalculate in dependency order), so algorithmically, this is not novel. Nothing is said about volatile and non-strict functions, and the handling of cyclic dependencies is unclear. Nothing is said about how to represent the dependency set.

Chapter 4

The support graph

The *support graph* shows which cells directly statically depend on a given cell. A support graph facilitates minimal recalculation as well as ordering of formula recalculation. As discussed in section 3.3.2, the number of edges in the support graph may be very large. This chapter investigates a compact representation of the support graph, as well as efficient algorithms for building, maintaining and using it.

These ideas may be the subject of a patent application. At this time, the ideas have not been implemented and evaluated in practice. In particular they are not part of the current CoreCalc implementation and not covered by the license shown on page 3. To make good use of them, the recalculation mechanism implemented by CoreCalc must be radically changed, so as to initiate recalculation in bottom-up order instead of the current top-down order (see section 1.7.1). Such a change will provide a number of further benefits; see section 4.7.

4.1 Compact representation of the support graph

As shown in section 3.3.2, there may be a large number of edges in the static support graph. There are two reasons for this: A cell area argument in a formula may refer to a large number of cells, and copying of such a formula may multiply that by a large factor. First, a cell that contains the formula `SUM($A$1:$A$10000)` will belong to the support of 10 000 cells. Second, that formula may be copied to 5 000 other cells, thus making each of the 10 000 cells support 5 000 cells, for a total of $10\,000 \cdot 5\,000 = 50\,000\,000$ support graph edges between only $10\,000 + 5\,000 = 15\,000$ cells. Clearly a naive explicit representation of the support graph would require far too much memory even for modest-size spreadsheets.

Here we shall investigate how to compactly represent support graph edges from a cell to families of other cells that all hold virtual copies of the same expression. That is, we shall attack the second source of the support graph edge problem, and reduce the 50 000 000 support graph edges needed above to 10 000 compactly represented families of support graph edges.

4.2 Arithmetic progressions and FAP sets

A finite *arithmetic progression* has the form $a, a+b, \ldots, a+(k-1)b$ where a, b and $k \geq 0$ are integers. Arithmetic progressions are interesting because the row numbers (and column numbers) of virtual copies of an expression can be described by an arithmetic progression with $b \geq 1$ and $k \geq 1$. To make this observation seem profound, we shall refer to the set of elements in such a finite arithmetic progression as a FAP set, and call a its offset, b its period, and k its cardinality. Clearly, FAP sets generalize singleton sets ($k = 1$) and integer intervals ($b = 1$).

First observe that a FAP set can be represented compactly by the triple (a,b,k). We shall abuse notation and denote the set itself by the triple, like this:

$$(a,b,k) = \{a, a+b, \ldots, a+(k-1)b\}$$

For the empty set ($k = 0$) and for one-element sets ($k = 1$) the representation by a triple is not unique. We shall usually not represent the empty set by a triple at all, and we therefore say that the representation is *normalized* if $b \geq 1$, and $k \geq 1$, and $k = 1 \Rightarrow b = 1$. Figure 4.1 shows some equivalences for FAP sets.

$(m,b,0)$	$=$	$\{\}$	empty set
$(m,b,1)$	$=$	$\{m\}$	singleton
$(m,1,n-m+1)$	$=$	$\{m, m+1, \ldots, n\}$	interval
(a,b,k_1+k_2)	$=$	$(a,b,k_1) \cup (a+bk_1,b,k_2)$	chop
$(a,1,k_1+k_2)$	$=$	$(a,2,k_1) \cup (a+1,2,k_2)$	zip two

Figure 4.1: Some equivalences for FAP sets.

The "zip two" equivalence in the figure is a special case of this "zip multiple" equivalence (with $b = 1$ and $n = 2$):

$$(a,b,k) = (a,nb,k_0) \cup (a+b,nb,k_1) \cup \ldots \cup (a+(n-1)b,nb,k_{n-1})$$

where $k_i \geq 0$ is the greatest integer such that $n(k_i - 1) \leq k-1-i$, which can be computed as `k_i=(k-1-i+n)/n`. This can be used to find a non-redundant FAP set representation of the union of two FAP sets, in the form of a set of mutually disjoint FAP sets.

For example, to represent the union of $(a_1,b_1,k_1) = (0,2,10)$ and $(a_2,b_2,k_2) = (0,3,8)$, notice that the least common multiple of b_1 and b_2 is $b = lcm(2,3) = 6$. We use the "zip multiple" equivalence to rewrite the two FAP sets to use the common period $b = 6$, with the multipliers n being $n_1 = b/b_1 = 3$ and $n_2 = b/b_2 = 2$ respectively:

$$\begin{aligned} (0,2,10) &= (0,6,4) \cup (2,6,3) \cup (4,6,3) \\ (0,3,8) &= (0,6,4) \cup (3,6,4) \end{aligned}$$

We see that the component FAP sets $(0,6,4)$ are identical whereas all the other component FAP sets are disjoint. Hence one non-redundant representation of the union of the sets is this:

$$(0,2,10) \cup (0,3,8) = (0,6,4) \cup (2,6,3) \cup (3,6,4) \cup (4,6,3)$$

In the above case the offsets of the two FAP sets were the same, namely $a_1 = a_2 = 0$. When the offsets are distinct, there is not necessarily any overlap between component FAP sets in the expansion. The two set overlap if there exist i_1 and i_2 with $0 \leq i_1 < k_1$ and $0 \leq i_2 < k_2$ such that $a_1 + i_1 b_1 = a_2 + i_2 b_2$. To see when this is the case, let again $b = lcm(b_1, b_2)$ and further let $\beta = gcd(b_1, b_2) = b_1 b_2 / b$ so we have $n_1 = b/b_1 = b_2/\beta$ and $n_2 = b/b_2 = b_1/\beta$.

Now if the two sets overlap, then there exist $0 \leq i_1 < k_1$ and $0 \leq i_2 < k_2$ such that $a_2 - a_1 = i_1 b_1 - i_2 b_2 = i_1 \beta n_2 - i_2 \beta n_1 = \beta(i_1 n_2 - i_2 n_1)$, so $a_1 \equiv a_2 (mod\ \beta)$.

Hence if $a_1 \not\equiv a_2 (mod\ \beta)$ then the FAP sets (a_1, b_1, k_1) and (a_2, b_2, k_2) are disjoint and there is no need to expand them to obtain an irredundant representation.

On the other hand, if $a_1 \equiv a_2 (mod\ \beta)$, it depends also on the cardinalities k_1 and k_2 whether the sets overlap. Loosely speaking, if the cardinalities are large enough the sets will overlap, otherwise not. The sets overlap iff there are $0 \leq i_1 < k_1$ and $0 \leq i_2 < k_2$ such that $i_1 n_2 - i_2 n_1 = (a_2 - a_1)/\beta$, and whether this is the case depends on the bounds k_1 and k_2 on i_1 and i_2. Namely, since n_1 and n_2 are coprime, this equation would always have a solution if there were no bounds on i_1 and i_2.

4.3 Support graph edge families and FAP sets

The core idea is to represent the family of support graph edges from a cell to virtual copies of an expression by a pair of FAP sets, and hence by a pair $((a_c, b_c, k_c), (a_r, b_r, k_r))$ of triples. Namely, each copy operation giving rise to virtual copies creates a regular rectangular grid of virtual copies, and we let the triple (a_c, b_c, k_c) represent all the columns containing virtual copies, and let the triple (a_r, b_r, k_r) represent all the rows containing virtual copies. Hence the virtual copies occupy precisely the cells with these absolute, zero-based column and row numbers:

$$\{\ (c, r) \mid c \in (a_c, b_c, k_c),\ r \in (a_r, b_r, k_r)\ \}$$

We shall refer to such a product of FAP sets as a FAP grid. For a simple example, assume that cell B1 contains the formula `SUM(A$1:A$10000)`, and assume that formula is copied to the area B2:B5000, as shown in figure 4.2.

	A	B
1	0.5	=SUM(A$1:A$10000)
2	=A1*1.00001	=SUM(A$1:A$10000)
3	=A2*1.00001	=SUM(A$1:A$10000)
...	...	...
5000	=A4999*1.0001	=SUM(A$1:A$10000)
...	...	
10000	=A9999*1.00001	

Figure 4.2: A sheet with 15 000 active cells and 50 million support graph edges.

Then cell A1 must have support graph edges to cells B1, B2, ..., B5000, and likewise for A2, ..., A10000. In each case, this family of support graph edges can be represented by this FAP grid, or pair of FAP sets:

$$((a_c, b_c, k_c), (a_r, b_r, k_r)) = ((1,1,1), (0,1,5000))$$

of triples, that is, column 1, rows 0–4999. This representation must be used for each of the 10 000 cells in column A that support the cells in column B, but the space needed per cell in column A has been reduced from 5 000 cell addresses to six integers. In this case, a single pair of triples can even be shared among all the column A cells. Clearly, cell A1 also supports A2, A2 supports A3, and so on, so the support edges must be represented as the union of families of support graph edges, where each family can be represented by a pair of triples.

For a more interesting example, let cell B1 contain the formula `SUM(A$1:A1)`, as in figure 1.5, and assume that formula is copied to the area B2:B5000. Then cell A1 has support graph edges to cell B1; cell A2 has support graph edges to cells B1 and B2; and more generally, cell A*n* has support graph edges to cells B1, B2, ..., B*n*. For cell A*n*, where $1 \leq n$, the family of support graph edges can be represented by the pair $((1,1,1),(n-1,1,5000-n+1))$. Hence six integers per cell in column A still suffice to represent the support graph edge family, although the pairs of triples can no longer be shared between all the cells in column A.

To illustrate the need for FAP sets rather than just integer intervals, assume again that cell B1 contains the formula `SUM($A$1:$A$30)`, that the cells C1, D1, B2, C2 and D2 contain other formulas, and that the 3×2 block B1:D2 of formulas is copied to the cell area B1:M30 which has 12 columns and 30 rows, or $4 \cdot 15 = 60$ virtual copies of each of the formulas from B1:D2. As outlined in figure 4.3, the virtual copies of cell B1 are in cells B1, E1, H1, K1, B3, E3, ..., K29. This family of cells can be represented by the pair of triples $((1,3,4),(0,2,15))$, where the column triple $(1,3,4) = \{1,4,7,10\}$ represents the columns B, E, H and K, and the row triple $(0,2,15) = \{0,2,4,\ldots,28\}$ represents rows 1,3,5,..., 29.

	A	B	C	D	E	F	G	...	M
1	`0.5`	`=SUM($A$1:$A$30)`			`=SUM($A$1:$A$30)`			...	
2	`=A1`							...	
3	`=A2`	`=SUM($A$1:$A$30)`			`=SUM($A$1:$A$30)`			...	
4	`=A3`							...	
...									
29	`=A28`	`=SUM($A$1:$A$30)`			`=SUM($A$1:$A$30)`			...	
30	`=A29`							...	

Figure 4.3: Making virtual copies of a 3×2 cell area.

4.4 Creating and maintaining support graph edges

Let S_t be the set of absolute cell addresses of cells directly supported by the cell at address t. The support graph must have an edge (t,s) for each $s \in S_t$.

Assume that the formula f, if at cell ca, contains references to a set T of cells; that is, it directly statically depends on the cells in T. Then creation, deletion, copying and moving of that formula affects the support set of each cell $t \in T$ as follows:

- When *creating* the formula in the cell at address ca, we must add ca to S_t for each cell $t \in T$.

- When *deleting* the formula from the cell at ca, we must remove ca from S_t for each cell $t \in T$.

- When *copying* the formula from cell $ca = (c,r)$ within a $cols \times rows$ block that is being copied to a cell area whose upper left hand corner is (c_{ul}, r_{ul}), then the pair of triples $((c_{ul} - c, cols, k_c), (r_{ul} - r, rows, k_r))$ must be added to S_t for each $t \in T$. Here k_c is the number of columns that receive copies of the formula, and k_r is the number of rows that receive copies of the formula. This is true for absolute cell references in formula f to the cell addresses in T.

  Since relative references get adjusted by the copying, the story for those is a little more complicated. Define $f[c',r']$ to be the formula at target cell (c',r'), that is, with relative reference adjusted by the copying, and let $refers(f[c',r'])$ denote the set of cell addresses referenced from $f[c',r']$. Then for each $c' \in (c_{ul}, cols, k_c)$ and each $r' \in (r_{ul}, rows, k_r)$ we must add (c',r') to each S_{ca} where $ca \in refers(f[c',r'])$. If we do this naively as described here, then the support graph representation may require quadratic space. Using the technique from section 4.5.5, this can be done efficiently in a way that results in a much more compact support graph representation.

  Obviously, this operation also overwrites any formulas within the target cell area of the copying operation, which affects the support graph.

- When *moving* a formula from cell ca_1 to cell ca_2, we must remove ca_1 from and add ca_2 to the support set of each cell $t \in T$.

- When *inserting* $N \geq 1$ new rows just before row $R \geq 0$, each S_t that includes a row $r \geq R$ must be adjusted.

  More precisely, when the FAP set pair $((a_c, b_c, k_c), (a_r, b_r, k_r)) \subseteq S_t$ satisfies that $a_r + b_r(k_r - 1) \geq R$, then S_t must be adjusted.

  When $a_r \geq R$ too, simply add N to each member of the row FAP set (a_r, b_r, k_r) to obtain $(a_r + N, b_r, k_r)$.

  Otherwise, when $a_r < R \leq a_r + b_r(k_r - 1)$ we must split the row FAP set into two. One set represents those rows preceding the insertion, and another set representing those rows following the insertion, and then we must add N to

each element of the latter set. Determine the integer k such that $a_r + b_r(k-1) < R \leq a_r + bk$, then the resulting row FAP sets are (a_r, b_r, k) and $(a_r + N, b_r, k_r - k)$.

Insertion of columns is completely similar.

- When *deleting* the $N \geq 1$ rows numbered $R, R+1, \ldots, R+N-1$, those rows must be removed from each row FAP set (a_r, b_r, k_r), and for each row FAP set, N must be subtracted from the numbers of any rows following the deleted ones.

  Let k_1 be the greatest integer such that $a_r + b_r(k_1 - 1) < R$. The idea is that k_1, if positive, is the number of rows in the row FAP set that precede the deleted rows. Similarly, let k_2 be the least integer such that $R + N \leq a_r + b_r k_2$; then $k_r - k_2$, if positive, is the number of rows in the row FAP set that follow the deleted rows.

  The original row FAP set must be replaced by zero, one or two non-empty row support sets, as follows:

  - If $1 \leq k_1$ then (a_r, b_r, k_1) is part of the resulting row FAP set. These are the rows preceding the deleted rows.

    One can compute k_1 by the expression `k_1 = (R-a_r+b_r-1)/b_r`; see section 4.5.6.
  - If $k_2 < k_r$ then $(a_r + b_r k_2 - N, b_r, k_r - k_2)$ is part of the resulting row FAP set. These are the rows following the deleted rows.

    One can compute k_2 by the expression `k_2 = (R+N-a_r+b_r-1)/b_r`; see section 4.5.6.

  Figure 4.4 shows examples of adjustment of row FAP sets for some formula `=Z1` when the shaded rows 4 through 7 are deleted. The original FAP set triples (a_r, b_r, k_r) and the resulting k_1 and k_2 are shown below the spreadsheet fragment.

  Deletion of columns is completely similar.

It seems that it is never necessary to have more than one instance of a given row FAP set representation in the implementation. Namely, assume FAP set S appears in the support of two different cells. Then if any formula in a cell in S is updated so that set S must be changed, then this change affects both cells in the same way. Hence updates to the support graph can be made very simple; they just require some mechanism to avoid performing the update more than once.

4.5 Reconstructing the support graph

The previous section describes how the support graph can be *maintained* while inserting, deleting, moving and copying formulas, and so on. An equally relevant challenge is to efficiently *create* the support graph from a spreadsheet that does not have one, such as a newly loaded spreadsheet created by an external program. It

	A	B	C	D	E
1	=Z1				
2		=Z1			
3			=Z1		
4	=Z1			=Z1	=Z1
5		=Z1			
6			=Z1		
7	=Z1			=Z1	=Z1
8		=Z1			
9			=Z1		
10	=Z1			=Z1	
a_r	0	1	2	3	3
b_r	3	3	3	3	3
k_r	4	3	3	3	2
k_1	1	1	1	0	0
k_2	3	2	2	2	2

Figure 4.4: Effect on row FAP set for $S_{=\mathtt{Z1}}$ of deleting the grey rows ($N = 4$ and $R = 3$).

is trivial to find a poor solution to this problem, but finding an optimal one is quite likely an NP-complete problem, as it involves finding a kind of minimal exact set cover.

We propose a two-stage approach in which one first builds an *occurrence map* for each formula (section 4.5.1), and then uses the occurrence map to build the support graph (sections 4.5.2 through 4.5.5).

4.5.1 Building a formula occurrence map

The following procedure seems usable for building a reasonably compact occurrence map for typical spreadsheets:

- Scan columns from left to right.
- In the scan of a column c, we maintain a map m from expressions e (in the internal, copy-invariant representation) to a sequence $m(e)$ of triples, each triple representing a FAP set.

  The goal is that after scanning the column, the union of the members of the triples in $m(e)$ is exactly the set of those cells (in that column) that contain formula e.

  We maintain a map from expressions (to a sequence of triples), rather than a map from the cells that those expressions refer to. The reason is that the latter map could have a much larger domain: an expression such as `SUM(A1:A10000)` in effect represents 10 000 cells, and clearly it is more efficient to map from one such expression than from 10 000 individual cells.

The map from expressions to sequences of FAP sets can be maintained as a hash dictionary, with equality being expression object reference identity or expression abstract syntax tree equality. The former is faster but less precise than the latter, but imprecision just leads to a less compact representation of the support graph, not to wrong results.

At the beginning of the scan of the column, the map m is empty.

- The rows of the column are scanned in order from row 0. Assume the cell in row r contains a formula e, then we proceed as follows:
  - If $e \notin dom(m)$ then set $m(e) := [(r,1,1)]$.
    The expression e has not been seen before in this column.
  - Otherwise assume $m(e) = [\ldots,(r',b',k')]$.
    The expression has been seen before and the most recent occurrence was at $r'+b'(k'-1)$.
  - If $k' = 1$ then update the last item of $m(e)$ to $(r', r-r', 2)$.
    The most recent FAP set has only one element, and we can extend it to have $k = 2$, with the step b being the difference $r-r'$ between this row and the row in the FAP set.
  - Otherwise, if $r = r'+b'k'$ then update the last item of $m(e)$ to $(r', b', k'+1)$.
    The new row is an additional member of the most recent FAP set, so we extend that set.
  - Otherwise, add a new last item $(r,1,1)$ to $m(e)$.
    The new row is not a member of the most recent FAP set, so we do not extend that sequence, but begin a new one.

NOTE: This simple scheme can be defeated by writing a formula in A1, copying it to A3, then copying the block A1:A3 to many further cells. It would give alternating distances 2,1,2,1,2,1, ...

Hence instead we might collect the sequence of distances as above, and infer the FAP sets $(0,3,k_1)$ and $(2,3,k_2)$ from them. This is quite easy, presumably, by considering derived sequences of 2-sums, 3-sums etc. that are constant.

Another approach would be to use a Fast Fourier Transform (FFT) [19] of the column's formula identities to find the spectrum of the formula occurrences. In the example sketched here, one would expect to find the periods 1, 2 and 3, with 3 having twice the power of 1 and 2, which would indicate that 3 is the most interesting period for that column. (Periods that are multiples of 3 would appear also, but with lesser power). After noting this, one can perform a column scan that keeps for each formula up to 3 partially constructed FAP sets.

A conceptually simpler auto-correlation computation might serve the same purpose as the FFT, but might be slower if the correlation window needs to be large. In both cases, a possible weakness of this method is that the pattern of virtual formula copies may not be uniform over the column.

Even more speculatively, could we perform a two-dimensional Fourier Transform to find repetitive structure in columns and rows at the same time? It would seem that a two-dimensional auto-correlation function could easily be computed. However, it would increase computational cost a good deal, especially if we want to handle copying of blocks up to, say, 20x20 cells.

- Let m_c be the map resulting from scanning column c as outlined above. Now scan all columns, building a map M, from pairs (f, ts) of formulas f and triple sequences ts, to a list of set of pairs $((a_c, b_c, k_c), (a_r, b_r, k_r))$ of triples, using a scheme similar to the above.
- When the map M has been built, the support graph can be created as explained in the next section.

Instead of building a map m from formulas e to sequences of triples, we could build a map from ccars, where a ccar is a cell reference such as `B7` or cell area reference such as `B7:D8`. The chief advantage of this would be to permit better sharing of support graph edge descriptions. The disadvantages are that it would exacerbate the problem mentioned in the Note above and make a solution to that problem more urgent, and that it would require a traversal of each expression $e \in dom(m)$ when processing a cell in a column, instead of just looking up the expression's reference in m. However, even if we stick to letting m map from expressions e, the processing of each e described in the sections below would consist of processing each ccar in e.

4.5.2 From formula occurrence map to support graph

Now let us consider how to build the support graph. This is done by two nested loops:

- For each $e \in dom(M)$, for each member $(C,R) \in M(e)$, find the set CA of absolute cell addresses that is referred by at least one occurrence of formula f within the grid of cells described by (C,R). This can be computed in constant time for each *ccar*, as shown in section 4.5.4.
- Then for each such cell address $ca \in CA$, compute the pair of triples that represents the subset of (C,R) that ca actually supports. For each ca this computation is a constant time operation, as shown in section 4.5.5.

4.5.3 Some examples

First let us consider some concrete examples of virtual formula copies that contain cell area references, such as those in figure 4.5. The task is to find which cells in column A support which cells in column B, C and D. The column B and C formula copies are described by the triple $(0,2,5)$; the column D formula copies are described by the triple $(1,2,5)$.

In column B, each of cells A1–A10 support all of the cells B1, B3, B5, B7 and B9, which can be described by the triple $(0,2,5)$.

In column C, cell A1 supports C1, C3, C5, C7 and C9, described by $(0,2,5)$; cells A2 and A3 both support C3, C5, C7 and C9, described by $(2,2,4)$; cells A4 and A5 both support C5, C7 and C9, described by $(4,2,3)$; cells A6 and A7 both support C7 and C9, described by $(6,2,1)$; cells A8 and A9 both support C7 and C9, described by $(6,2,1)$; and cell A10 supports nothing in column C.

In column D, cell A1 and A2 both support D2, D4, D6, D8 and D10, described by $(1,2,5)$; cells A3 and A4 both support D4, D6, D8 and D10, described by $(3,2,4)$; cells A5 and A6 both support D6, D8 and D10, described by $(5,2,3)$; cells A7 and A8 both support D8 and D10, described by $(7,2,2)$; and cells A9 and A10 both support D10, described by $(9,2,1)$.

	A	B	C	D
1	11	SUM($A$1:$A$10)	SUM($A$1:$A1)	
2	12			SUM($A$1:$A2)
3	13	SUM($A$1:$A$10)	SUM($A$1:$A3)	
4	14			SUM($A$1:$A4)
5	15	SUM($A$1:$A$10)	SUM($A$1:$A5)	
6	16			SUM($A$1:$A6)
7	17	SUM($A$1:$A$10)	SUM($A$1:$A7)	
8	18			SUM($A$1:$A8)
9	19	SUM($A$1:$A$10)	SUM($A$1:$A9)	
10	20			SUM($A$1:$A10)

Figure 4.5: Finding the support edges from each cell in column A.

In continuation of the above example, consider figure 4.6 and let us find which cells in column A support which cells in column E, F and G. The column E formula copies are described by the triple $(0,2,5)$; the column F formula copies by $(2,2,4)$; and the column G formula copies are described by $(1,3,3)$.

In column E, cells A1 and A2 both support E1 only, described by $(0,2,1)$; cell A3 supports E1 and E3, described by $(0,2,2)$; cell A4 supports E1, E3, E5, E7 and E9, described by $(0,2,5)$; cell A5 supports E5, E7 and E9, described by $(4,2,3)$; cells A6 and A7 both support E7 and E9, described by $(6,2,2)$; cells A8 and A9 both support E9, described by $(8,2,1)$; and cell A10 supports nothing in column E.

In column F, cells A1 and A2 both support F3, described by $(2,2,1)$; cell A3 supports F3 and F5, described by $(2,2,2)$; cell A4 supports F5, described by $(4,2,1)$; cell A5 supports F5 and F7, described by $(4,2,2)$; cell A6 supports F7, described by $(6,2,1)$; cell A7 supports F7 and F9, described by $(6,2,2)$; cells A8 and A9 both support F9, described by $(8,2,1)$; and cell A10 supports nothing in column F.

In column G, cells A1 and A2 both support G2, described by $(1,3,1)$; cell A3 supports nothing in column G; cells A4 and A5 both support G5, described by $(4,3,1)$; cell A6 supports nothing in column G; cells A7 and A8 both support G8, described by $(7,3,1)$; and cells A9 and A10 support nothing in column G. In this case, with non-overlapping areas of supporting cells, each triple represents a single edge, and no space is saved by our supposedly compact support graph representation. However,

the number of support graph edges is already only linear in the number of formula occurrences.

	A	...	E	F	G
1	11	...	SUM($A$4:$A1)		
2	12	...			SUM($A1:$A2)
3	13	...	SUM($A$4:$A3)	AVG($A1:$A3)	
4	14	...			
5	15	...	SUM($A$4:$A5)	AVG($A3:$A5)	SUM($A4:$A5)
6	16	...			
7	17	...	SUM($A$4:$A7)	AVG($A5:$A7)	
8	18	...			SUM($A7:$A8)
9	19	...	SUM($A$4:$A9)	AVG($A7:$A9)	
10	20	...			

Figure 4.6: Finding the support edges from each cell in column A; more cases.

4.5.4 From occurrence map to referred cells

Now let us consider more generally the problem of finding the set of cell addresses referred to by the occurrences of a formula; this is the first step in section 4.5.2. For simplicity we will consider this problem in one dimension only, working on the formula (or ccar) occurrences in one column at a time. Hence we consider triples of the form (a_r, b_r, k_r), describing a pattern of occurrences of a single formula or ccar. In fact, regardless of whether the occurrence map M created in the previous section maps from formulas f or ccars *ccar*, we shall consider one *ccar* at a time, if necessary extracted from the formulas in the domain of M.

Hence we consider a *ccar* that appears in a given column c and an associated triple (a,b,k) that describes the rows of that column in which the *ccar* appears. The procedure then becomes:

(1) First we find the set CA of all those absolute cell addresses *ca* that are referred to by some occurrence $ccar[c,r]$ for $r \in (a,b,k)$. This set can be represented by an interval (in the one-dimensional case) or the product of two intervals (in the general case).

The point of computing CA in advance is to avoid analysing any *ca* more than once in step (2). Namely the cell areas referenced by different occurrences of the same formula may overlap, and certainly do in columns B, C, D, E and F above, and processing each such cell area in turn could change a linear time algorithm to a quadratic time algorithm.

(2) Next, for each $ca \in CA$ we compute the triple (a',b,k') representing the subset of (a,b,k) that *ca* actually supports. This triple then is used to represent support graph edges from *ca* to cells in column c supported by *ca*.

Step (1) above in principle must compute $r = a + bi$ for all $0 \leq i < k$, and then find the union of the sets of cells referred to by each occurrence $ccar[c,r]$ of the cell/cell area reference:

$$CA = \bigcup_{0 \leq i < k} refers(ccar[c, a+bi])$$

This looks like a potentially expensive operation, but it turns out that CA is an interval and can be computed in constant time in all cases where computing it matters, namely when the referred-to cell areas overlap, so that processing the areas one by one would duplicate work. The CA sets for the examples in figures 4.5 and 4.6 are shown in figure 4.7, in A1 and C0R0 format. The column G case shows that CA in general may not be an interval.

	Formula (A1)	Formula (C0R0)	Occurs	Referred-to cells (CA)
B	`SUM($A$1:$A$10)`	`SUM(C0R0:C0R9)`	$(0,2,5)$	{ A1, A2, . . . , A9, A10 }
C	`SUM($A$1:$A1)`	`SUM(C0R0:C0R[0])`	$(0,2,5)$	{ A1, A2, . . . , A9 }
D	`SUM($A$1:$A2)`	`SUM(C0R0:C0R[0])`	$(1,2,5)$	{ A1, A2, . . . , A9, A10 }
E	`SUM($A$4:$A1)`	`SUM(C0R4:C0R[0])`	$(0,2,5)$	{ A1, A2, . . . , A9 }
F	`AVG($A1:$A3)`	`AVG(C0R[-2]:C0R[0])`	$(2,2,4)$	{ A1, A2, . . . , A9 }
G	`SUM($A1:$A2)`	`SUM(C0R[-1]:C0R[0])`	$(1,3,3)$	{ A1, A2, A4, A5, A7, A8 }

Figure 4.7: Formula occurrences and referred-to cells in figures 4.5 and 4.6.

To see that the set CA can be computed efficiently, consider the four possible forms of *ccar*, using C0R0-format references (figure 2.2) for the two corners. We assume here that *ccar* is a cell area reference, since a simple cell reference such as A1 can be represented by a cell area reference A1:A1.

- When *ccar* is `C0R`i_1`:C0R`i_2, that is, both corners are absolute references, the occurrences (a,b,k) do not matter. Obviously the exact result is an interval, namely, assuming wlog $i_1 \leq i_2$:

  $$CA = \{i_1, \ldots, i_2\}$$

- When *ccar* is `C0R`i_1`:C0R[`i_2`]`, that is, one corner is an absolute reference, the other is a relative one, the occurrences do matter. Still the exact result is an interval, namely:

  $$CA = \{n_1, \ldots, n_2\}$$

  where

  $$n_1 = \min(i_1, a+i_2, a+b(k-1)+i_2) \quad \text{and} \quad n_2 = \max(i_1, a+i_2, a+b(k-1)+i_2)$$

- When *ccar* is `C0R`$[i_1]$`:C0R`i_2, that is, one corner is a relative reference, the other is an absolute one, we have the same situation as above; simply swap i_1 and i_2 in the formulas.

- When *ccar* is `C0R`$[i_1]$`:C0R`$[i_2]$, and assuming wlog $i_1 \leq i_2$, the exact set is

$$CA = \bigcup_{0 \leq i < k} \{a + ib + i_1, \ldots, a + ib + i_2\}$$

  This can be approximated by an interval:

$$CA \subseteq \{a + i_1, \ldots, a + i(k-1) + i_2\}$$

  In fact, this interval is the exact answer when the cell areas referred to from the formula occurrences are adjacent or even overlap, that is, when $i_2 - i_1 + 1 \geq b$ as in column F of figure 4.6. When this is not the case, there is no point in building the set *CA* explicitly; instead step (2) described in section 4.5.5 below should iterate over the *ca* in each set $\{a + ib + i_1, \ldots, a + ib + i_2\}$ individually, for $0 \leq i < k$. Precisely because those sets do not overlap, this means that no *ca* will be analysed twice.

For the formulas in figures 4.5 and 4.6, we find precisely the *CA* sets shown in figure 4.7.

4.5.5 The support graph edges from a referred cell

Now let us consider how to find the support graph edges from a given referred-to cell; this is the inner loop (2) in section 4.5.4 above. We consider an absolute cell address $(c', r') = ca \in CA$, where *CA* was computed in the previous section, and recall that the occurrences of *ccar* are described by (a, b, k). We must find the set

$$S_{ca} = \{\ j \mid ca \in \mathit{refers}(\mathit{ccar}[c, j]),\ j = a + bi,\ 0 \leq i < k\ \}$$

The challenge is to compute this set efficiently, and to find a compact representation of it. Preferably, we want to find a triple (a', b, k') that is equivalent to S_{ca}. Again this computation can be performed by case analysis in the form of the *ccar*.

- When *ccar* is `C0R`i_1`:C0R`i_2 and $ca \in CA$, clearly *ca* supports every occurrence of the *ccar*, so

$$S_{ca} = (a', b, k') = (a, b, k)$$

- When *ccar* is `C0R`i_1`:C0R`$[i_2]$, there are three cases, according as r' equals, precedes, or follows the anchor point i_1:

  - If $r' = i_1$ then

    $$S_{ca} = (a, b, k)$$

  - If $r' < i_1$, find the greatest $k_1 \leq k$ such that $i_2 + a + b(k_1 - 1) \leq r'$. Then if $k_1 \geq 1$ then the set is non-empty:

    $$S_{ca} = (a, b, k_1)$$

    The number k_1 can be computed as `k_1 = Math.Min(k, (r'-i_2-a+b)/b)`.
  - If $i_1 < r'$, find the least $k_1 \geq 0$ such that $r' \leq i_2 + a + bk_1$. Then if $k_1 < k$ then the set is non-empty:

    $$S_{ca} = (a + bk_1, b, k - k_1)$$

    The number k_1 can be computed as `k_1 = Math.Max(0, (r'-i_2-a+b-1)/b)`.

- When *ccar* is `C0R`$[i_1]$`:C0R`i_2, then the cases and solutions are exactly as above, only with i_1 and i_2 swapped.
- When *ccar* is `C0R`$[i_1]$`:C0R`$[i_2]$, and assuming wlog $i_1 \leq i_2$, determine the greatest k_1 such that $i_2 + a + b(k_1 - 1) < r'$ and the least k_2 such that $r' < i_1 + a + bk_2$, and define $k'_1 = \max(0, k_1)$ and $k'_2 = \min(k, k_2)$. Then if $k'_1 < k'_2$ we have

  $$S_{ca} = (a + bk'_1, b, k'_2 - k'_1)$$

  The number k'_1 can be computed as `k_1' = Math.Max(0, (r'-i_2-a+b-1)/b)`.

  The number k'_2 can be computed as `k_2' = Math.Min(k, (r'-i_1-a+b)/b)`.

  See figures 4.9 for an example where the referred-to row ranges overlap, and figure 4.10 for an example where the referred-to row ranges do not overlap.

Considering the formulas in figure 4.7, we find for each cell in column A the support graph edge families shown in figure 4.8. Fortunately, these agree with the sets of supported cells found in the informal discussion of those figures.

4.5.6 Computer integer arithmetics

Computing with integers and inequalities requires some care, both because the algebra rules are different from those of arithmetics on reals, and because programming languages (except for Standard ML [58]) traditionally implement integer division with negative numerators in a peculiar way. Essentially, most languages let integer division truncate towards zero rather than towards minus infinity, and therefore satisfy $(-n)/d = -(n/d)$ but not $(n+d)/d = n/d + 1$ for $d > 0$.

Let an integer $d > 0$ be given.

- To find the least integer q such that $n \leq dq$, where $n \geq 0$,
  compute `q = (n+d-1)/d`.

Cell	r'	B	C	D	E	F	G
A1	0	(0,2,5)	(0,2,5)	(1,2,5)	(0,2,1)	(2,2,1)	(1,3,1)
A2	1	(0,2,5)	(2,2,4)	(1,2,5)	(0,2,1)	(2,2,1)	(1,3,1)
A3	2	(0,2,5)	(2,2,4)	(3,2,4)	(0,2,2)	(2,2,2)	{}
A4	3	(0,2,5)	(4,2,3)	(3,2,4)	(0,2,5)	(4,2,1)	(4,3,1)
A5	4	(0,2,5)	(4,2,3)	(5,2,3)	(4,2,3)	(4,2,2)	(4,3,1)
A6	5	(0,2,5)	(6,2,2)	(5,2,3)	(6,2,2)	(6,2,1)	{}
A7	6	(0,2,5)	(6,2,2)	(7,2,2)	(6,2,2)	(6,2,2)	(7,3,1)
A8	7	(0,2,5)	(8,2,1)	(7,2,2)	(8,2,1)	(8,2,1)	(7,3,1)
A9	8	(0,2,5)	(8,2,1)	(9,2,1)	(8,2,1)	(8,2,1)	{}
A10	9	(0,2,5)	{}	(9,2,1)	{}	{}	{}

Figure 4.8: Support graph edge families for column A cells in figures 4.5 and 4.6.

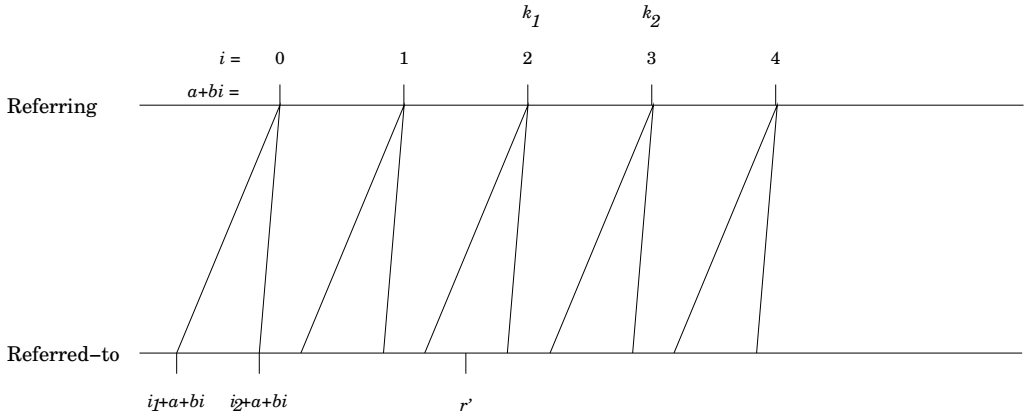

Figure 4.9: Five virtual copies of a cell area reference *ccar* of form `C0R`$[i_1]$`:C0R`$[i_2]$, that is, with both endpoints relative. Non-overlapping cell areas. The row number r' is included in the cell areas referred by i for which $k_1 \leq i < k_2$, in this case, $i = 2$.

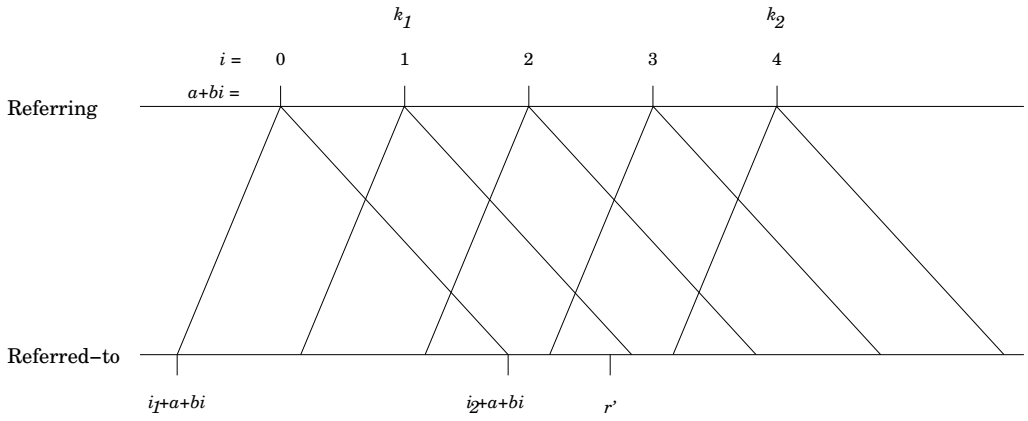

Figure 4.10: Five virtual copies of a cell area reference *ccar* of form `C0R`$[i_1]$`:C0R`$[i_2]$, that is, with both endpoints relative. Overlapping cell areas. The row number r' is included in the cell areas referred by i for which $k_1 \leq i < k_2$, in this case, $i = 1,2,3$.

- To find the greatest integer q such that $dq \leq n$, where $n \geq 0$, compute `q = n/d`.
- To find the greatest integer q such that $dq < n$, where $n \geq 0$, compute `q = (n-1)/d`.
- To find the greatest integer q such that $dq \leq n$, where $n \geq -d$, compute `q = (n+d)/d-1`.

  Computing this as `q = n/d` would be wrong because integer division in most programming languages truncates the quotient towards zero rather than towards minus infinity, and therefore does not satisfy the expected equivalence $(n+d)/d = n/d+1$ for integers n and $d > 0$.
- To find the greatest integer q such that $dq < n$, where $n \geq -d$, compute `q = (n+d-1)/d-1`.
- To find the greatest integer q such that $d(q-1) < n$, where $n \geq -d$, compute `q = (n+d-1)/d`.

4.6 Related work

It is clear that some explicit representation of the support graph in used both in Excel [25] and in Gnumeric [56]. However, it seems that the representation in both cases is considerably different from what is suggested here; see section 3.3.5.

Burnett and others [14] introduces the concept of cp-similar cells, essentially meaning that their formulas could have arisen by a copy-paste operation from one

cell to the other. This is equivalent to saying that their R1C1-representations (or C0R0-representations) are identical. They further define a region to be a group of adjacent cp-similar cells; which is a special case of a FAP grid of formula copies. The purpose of grouping cells into regions is not to support recalculation, but to reduce the task of testing to one representative from each region.

A paper by Mittermeir and Clermont proposes the highly relevant idea of a "semantic class" of cells [59], which corresponds to Burnett's notion of cp-similar cells, but the cells in a semantic class are not necessarily adjacent. As above, the paper's goal is to assist users in discovering irregularities and bugs in spreadsheets, not to implement spreadsheet programs. The paper defines semantic class using first order logic but does not suggest how to represent semantic classes and does not provide algorithms for reconstructing or maintaining semantic classes.

Abraham and Erwig [3] use the concept of cp-similarity to infer templates for spreadsheets, in the sense of Gencel [31]. This paper seems to work with regular grids of cp-similar cells, which makes this idea very similar to FAP grids, but there is no explanation of an algorithm for discovering such regular grids. The purpose of template inference (and Gencel) is to guide and limit the editing of a spreadsheet and hence to prevent the introduction of errors.

4.7 Applications of the support graph

- The most obvious application of the support graph is to perform minimal recalculation. One needs to recalculate only those nodes (spreadsheet formulas) in the support graph that are reachable from changed cells or volatile cells.
- This can be combined with the existing bottom-up recalculation mechanism of CoreCalc, simply by performing (for instance) a depth-first traversal of the support graph, recalculating cells on the way. There's even a good chance that cells will be recalculated in the proper order this way, though no guarantee, because the same cell may be both directly and indirectly supported by a given cell. The management of the `uptodate` and `visited` fields would have to be modified, though, since only some of the workbook's cells will be visited in each recalculation.
- However, when the static support graph contains no (static) cycles, a better alternative is to perform a topological sort of the reachable cells in the support graph, to obtain a safe recalculation order. Then all referring expressions can assume that referred-to cells are up to date, thus saving checks and recalculation time, and avoiding the need for a recursion stack. The Expr subclasses can have a special version of the `Eval` method for this purpose.
- When the static support graph contains no (static) cycles, the cycle check can be left out, thus saving recalculation time.
- If we create "efficient" versions of expressions by inter-cell type analysis, the support graph can be used to efficiently find the cells whose type analysis may

be affected by a type change in a given cell.

- The support graph could help compile expressions to sequential code, for instance to generate code from function sheets.
- The support graph could help schedule recalculation for multiprocessor architectures and FPGA implementation [51].

4.8 Limitations and challenges

The FAP grid representation of the support graph provides a compact representation of the support graph for highly regular grids of formulas, that can furthermore be efficiently constructed and maintained. However, for spreadsheets with an irregular structure, which cannot be built using only a few copy operations, the FAP grid representation may degenerate to a representation of all single edges. For instance, a spreadsheet to compute the discrete Fourier transform [8, 19] has a structure that cannot easily be built using copy operations, as shown in figure 4.11.

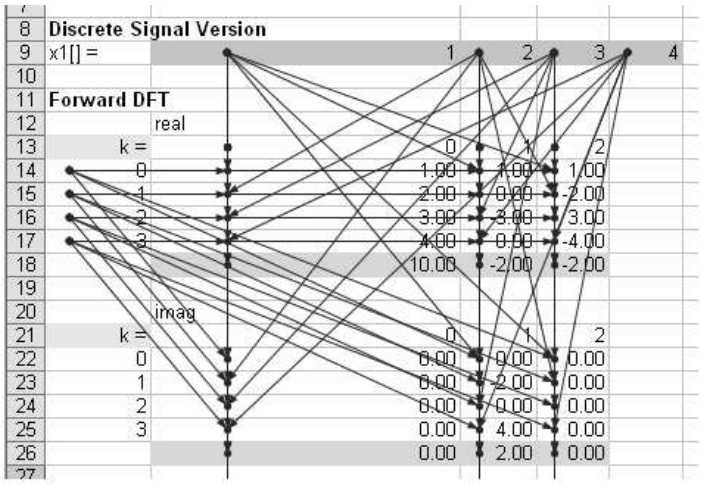

Figure 4.11: Dependencies in a discrete Fourier transform. (Spreadsheet from [89]).

In general, and especially in the presence of `SUM` formulas and other functions with cell area arguments, it may be necessary to supplement the FAP grid representation of the support graph with other representations. One way to obtain a compact support graph representation is to maintain only an overapproximation of the actual support graph. This will not lead to wrong results, only to unnecessary work during recalculation.

Such an approximation may be needed in any case due to various highly dynamic features of spreadsheet programs. In particular, the functions `HLOOKUP` and `VLOOKUP` are used to search for a given value in a range, and the function `INDEX` is used to

retrieve a cell from a range given row and column. The exact dependencies are determined by the given value or the given row and column number, so to obtain a static support graph, one needs to make an (easy) overapproximation: Every cell in these functions' range argument supports the cell in which the function is used.

Similar problems are caused by functions such as `COUNTIF` and `SUMIF`, whose second argument is a string that dynamically gets interpreted as a formula, as in `COUNTIF(A1:A10000, "> 42")`. Fortunately it seems that (in Excel) the formula string cannot contain a reference to a cell. Since such a cell might contain any string denoting any formula, it would potentially make the `COUNTIF` cell dependent on the entire sheet. Thanks to this restriction, it is safe to assume that a `COUNTIF` function application depends only on its first argument, which is a cell area. It would be a much better design to provide a function object rather than a text as the second argument, and preferably a sheet-defined one, see chapter 6.

Chapter 5

Runtime code generation

In the CoreCalc spreadsheet implementation described in chapter 2, a spreadsheet formula is compiled to an abstract syntax tree, which can then be evaluated by calling its `Eval` method. This interpretive approach is easy to implement and experiment with, but performance suffers from the repeated dispatch on the abstract syntax, and from the need to wrap floating-point results as NumberValue objects (section 2.7) which in turn necessitates their unwrapping. In particular the allocation of very many small (NumberValue) objects may cause serious runtime overhead.

Performance can be considerably improved by compiling formulas to the abstract machine code for the .NET platform, called the Common Intermediate Language or CIL [52]. This avoids the interpretive overhead, and furthermore offers the opportunity to avoid allocating many of the intermediate NumberValue objects. Compilation of formulas to CIL can be thought of as a kind of specialization or partial evaluation [46] of the Expr class's `Eval` methods (section 2.6) with respect to the formula.

In fact, the ability to perform runtime code generation (RTCG) for formulas was a design goal when CoreCalc was created in 2005, and this is the real reason for the design of sharable expressions (section 2.8) and for preserving sharing at row or column insertions (sections 2.16 and 2.17). Runtime code generation is expensive in time and memory, and we want to share that expense among all virtual copies of a formula.

This chapter summarizes findings from the MSc thesis of Thomas S. Iversen [45] written at DIKU, University of Copenhagen, on runtime code generation for efficient spreadsheet implementation. Thomas Iversen's implementation, called TinyCalc, extended the CoreCalc code-base with runtime code generation, and with some other interesting facilities that have been merged back into CoreCalc.

5.1 Runtime code generation levels

To gradually approach the problem of compiling formulas to intermediate code, Thomas Iversen defined progressive levels of runtime code generation as shown in figure 5.1. The definition of increasing levels of code generation aggressiveness also permitted a study of which optimizations actually improved performance, for different kinds of sheets and formulas. The remainder of this section describes the RTCG levels, and section 5.2 report on their performance impact.

Level	Description
0	Interpretive evaluation of formulas as described in chapter 2.
1	Generate one code fragment per subexpression; so three for `A1+A2`.
2	Inline subexpression code fragments; so one for `A1+A2`.
3	As Level 2, and also performs generation time type analysis.
4	As Level 3, and also avoids creating intermediate Value objects.
5	As Level 4, and also inlines constant values.
6	As Level 5, and also avoids generating `stloc` followed by `ldloc`.
7	As Level 5 (not 6), and also applies type analysis to cell references.
8	As Level 7, and also avoids generating dead branches in conditionals.
9	As Level 8, and also specializes spreadsheet built-in functions.

Figure 5.1: Code generation levels in TinyCalc; 8 and 9 are not implemented.

5.1.1 Calling the generated code

Generated CIL code must be enclosed in methods that must be called somehow. Since a method generated from a spreadsheet formula is likely to be small, it is important that generated functions can be called efficiently. Previous experiments have shown that interface method calls are markedly faster than delegate calls on the .NET platform version 1 as well as 2 [80], and Thomas Iversen's experiments have confirmed this [45, section 5.5.1].

This makes code generation via the DynamicMethod class (introduced in .NET 2.0) less attractive, since such code must be called through delegate calls. On the other hand, to generate methods that can be called via interface calls, one needs to generate many classes, and experiments show that the time to do so increases quadratically with the number of classes [45, section 5.2.2]. This indicates that the .NET runtime uses a poor algorithm or data structure internally somewhere.

Nevertheless, the experiments were conducted by generating code that can be called through interface calls.

5.1.2 The code generation levels

Level 1 mainly serves to develop the infrastructure for code generation and to check that things work at all; the resulting code performs too many method calls to be of

practical interest, and no Level 1 performance results are reported below.

Level 2 code generates a special `Eval` method for each expression in cells, shared between all virtual copies of that expression. The generated function has the same header as the `Eval` method of the Expr class:

```
Value Eval(Sheet sheet, int col, int row) { ... }
```

Calling this method evaluates the formula at a given cell, specified by sheet, column and row.

Level 3 additionally performs a type analysis while generating the CIL code, thus sometimes avoiding result value tests. For instance, the expression `(5+6)+7` gives rise to the Level 3 code [45, Example 9] shown below, in which the objects representing the constants are inlined. In reality, CIL code is generated, but for clarity we show corresponding C# pseudocode:

```
Value Eval(Sheet sheet int col, int row) {
  NumberValue v0 = <constant5>.Value;
  NumberValue v1 = <constant6>.Value;
  NumberValue v2 = new NumberValue(v0.value + v1.value);
  NumberValue v3 = <constant7>.Value;
  return new NumberValue(v2.value + v3.value);
}
```

Level 4 works like Level 3 but attempts to avoid creating intermediate NumberValues when possible. At Level 4, the expression `(5+6)+7` is compiled to this code [45, Example 10] which avoids wrapping the sum `v2` of 5 and 6 as a NumberValue:

```
Value Eval(Sheet sheet int col, int row) {
  NumberValue v0 = <constant5>.Value;
  NumberValue v1 = <constant6>.Value;
  double v2d = v0.value + v1.value;
  NumberValue v3 = <constant7>.Value;
  return new NumberValue(v2d + v3.value);
}
```

Level 5 extends Level 4 by inlining constants, so that one can treat them as `double` and `String` values rather than NumberValue and TextValue objects. At Level 5, the expression `(5+6)+7` is compiled to this code [45, Example 11], whose execution creates only one NumberValue object:

```
Value Eval(Sheet sheet, int col, int row) {
  double v0d = <constant5>.Value.value;
  double v1d = <constant6>.Value.value;
  double v2d = v0d + v1d;
  double v3d = <constant7>.Value.value;
  return new NumberValue(v2d + v3d);
}
```

Level 6 attempts to improve on Level 5 by generating better CIL code. What is not obvious from the above pseudo-code is that Level 5 uses many local variables and frequently generates instruction sequences such as `stloc` followed by `ldloc`, where a value is stored to a local variable only to immediately be reloaded from that local variable. Instead Level 6 aims to use the CIL stack. In practice this rarely improves performance (and sometimes worsens it) because the .NET runtime system's just-in-time compiler can optimize such apparently suboptimal code itself.

Level 7 improves on Level 5 (not 6) by performing a form of inter-cell type analysis, in contrast to Level 3 which only performs an intra-cell type analysis. Thus the correctness of the generated code is dependent on the types of referred-to cells remaining the same. If the contents of a referred-to cell is changed, then new code must be generated for the referring cells. The prototype implementation does not include code to do this.

5.2 Performance measurements

Thomas Iversen made many performance measurements with TinyCalc, primarily to study the potential of runtime code generation. However, he included also three "real" spreadsheet implementations in the measurements, namely OpenOffice, Gnumeric, and Microsoft Excel. This throws some light on CoreCalc performance, since TinyCalc level 0 is essentially CoreCalc.

5.2.1 Some performance results

The first measurement recalculates 262144 copies of the formula `=SIN($A$1)`. The recalculation times are shown in figure 5.2. What is remarkable is that TinyCalc level 0, and hence CoreCalc, outperform all other spreadsheet implementations, although written in a managed language where the others are written in C, C++ and possibly assembly language. Also, it is clear that runtime code generation is not at all useful for a formula that simply calls a built-in function.

The second measurement concerns the evaluation of the first 14 terms of the Taylor series for e^x, when x is the value of cell A1:

$$\frac{1}{1}+\frac{A1}{1}+\frac{A1\cdot A1}{2\cdot 1}+\frac{A1\cdot A1\cdot A1}{3\cdot 2\cdot 1}+\cdots+\frac{A1\cdot\ldots\cdot A1}{14\cdot 13\cdot\ldots\cdot 2\cdot 1} \tag{5.1}$$

A total of 131072 copies of this formula was made, and recalculation time for the workbooks was measured. The results are shown in figure 5.3. Again CoreCalc comfortably outperforms Gnumeric and OpenOffice, but not Excel. Runtime code generation removes a considerable amount of overhead, and TinyCalc at the most aggressive level of optimization outperforms Excel.

The third measurement is similar, but now the denominators are made up of cell references also:

$$\frac{1}{1}+\frac{A1}{B1}+\frac{A1\cdot A1}{B2\cdot B1}+\frac{A1\cdot A1\cdot A1}{B3\cdot B2\cdot B1}+\cdots+\frac{A1\cdot\ldots\cdot A1}{B13\cdot B12\cdot\ldots\cdot B2\cdot B1} \tag{5.2}$$

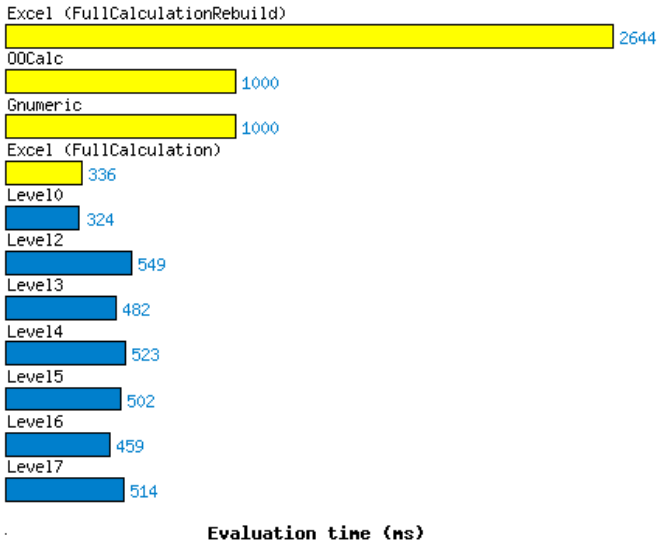

Figure 5.2: Evaluating mathematical function with reference argument. From [45].

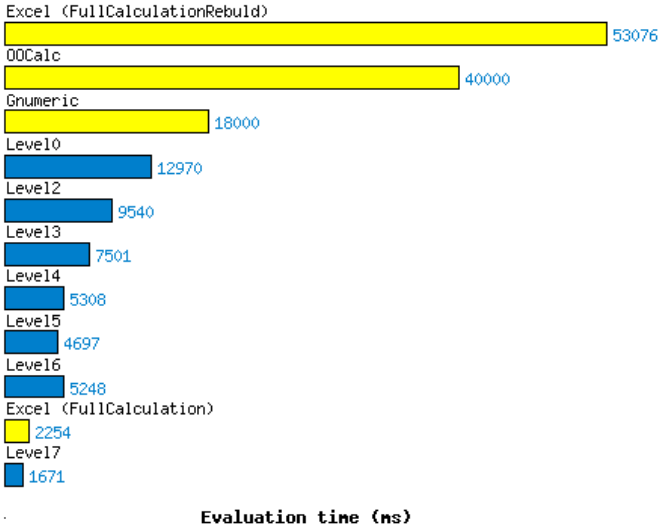

Figure 5.3: Evaluating Taylor series in equation 5.1. From [45].

The recalculation times for 131072 copies of this formula are shown in figure 5.4. CoreCalc outperforms OpenOffice and Gnumeric, and the most aggressive RTCG level gives performance comparable to that of Excel.

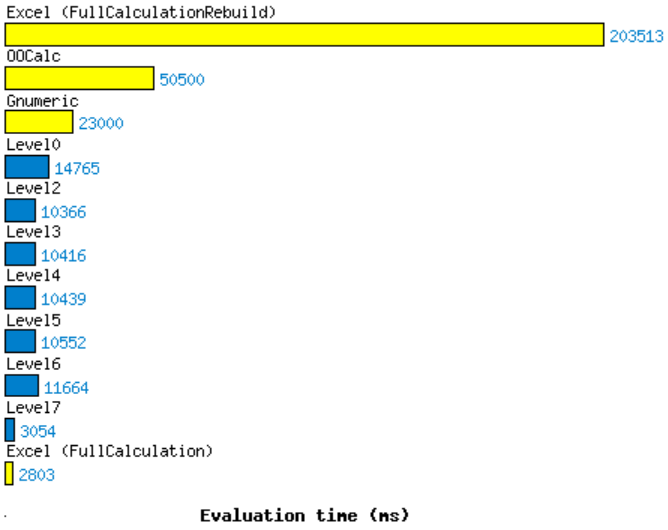

Figure 5.4: Evaluating Taylor series equation 5.2. From [45].

In the fourth measurement, each denominator simply refers to a cell in which the factorials are computed:

$$\frac{1}{1}+\frac{A1}{B1}+\frac{A2}{B2}+\frac{A3}{B3}+\cdots+\frac{A13}{B13} \tag{5.3}$$

The recalculation times for 131072 copies of this formula are shown in figure 5.5. Again, CoreCalc outperforms OpenOffice and Gnumeric, but even the most aggressive RTCG level is 1.7 times slower than Excel.

5.2.2 Performance results for cell area arguments

In all the preceding measurements, the dependency chains were short, and the number of direct dependencies was linear in the number of non-null cells. The fifth measurement combines long dependency chains and a large number of direct dependencies, obtained by using the formula `SUM($A$1:A1)`, as outlined in figure 5.6. The resulting sheet has dependency chains of length up to 12 287, and the `SUM` formulas induce $12\,288 \cdot 12\,289/2 = 75\,503\,616$ direct dependencies. A recalculation after a change to A1 requires 12287 multiplications and $12287 \cdot 12288 = 75491328$ additions.

The recalculation times for the sheet in figure 5.6 are shown in figure 5.7. Here TinyCalc level 0 and hence CoreCalc perform much worse than in previous measurements, although still better than Gnumeric.

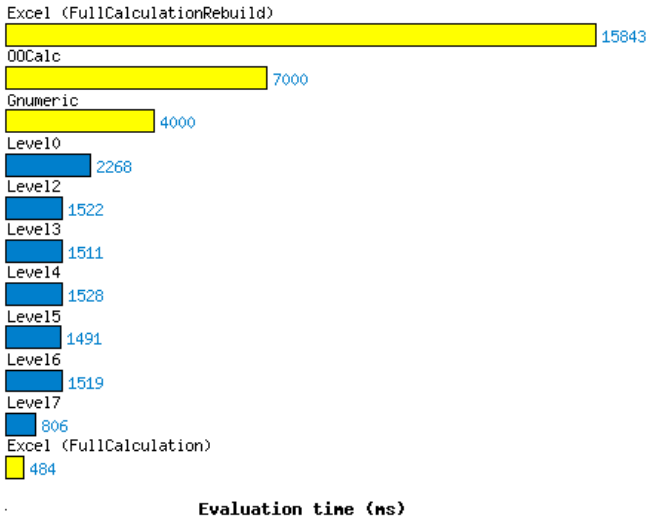

Figure 5.5: Evaluating Taylor series in equation 5.3. From [45].

	A	B
1	0.5	=SUM(A$1:A1)
2	=A1*1.00001	=SUM(A$1:A2)
3	=A2*1.00001	=SUM(A$1:A3)
...	...	...
12288	=A12287*1.00001	=SUM(A$1:A12288)

Figure 5.6: A sheet with long dependency chains and sum functions.

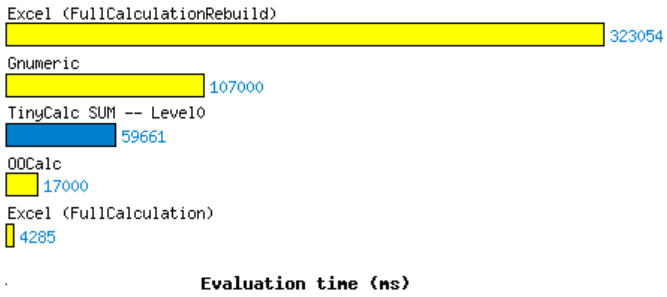

Figure 5.7: Evaluating a sheet with long dependencies and sum functions, using the default (heavy) matrix result representation in TinyCalc. From [45].

The reason is that CoreCalc implements function calls such as `SUM(A$1:A12288)` in an overly general way. The cell area argument evaluates to a two-dimensional array of type Value[,], each element of which contains a reference to a NumberValue object representing the value of a spreadsheet cell. This allows a matrix to contain a mixture of numbers and matrices, which may in turn contain matrices and so on. However, allocating 12288 Value[,] arrays and 75.5 million NumberValue objects is slow.

To get a better impression of the benefits that can be obtained by runtime code generation, Thomas Iversen made prototype implementations of two leaner mechanisms for evaluating cell area arguments. Two alternative implementations of aggregate functions (such as `SUM` and `AVG`) were implemented, each with several levels of code generation aggressiveness, here called MLevels, shown in figure 5.8.

MLevel	FASTSUM	DOUBLESUM
0	Interpretive as in CoreCalc	Interpretive as in CoreCalc
2	IL code, inline argument evaluation	IL code
3	MLevel 2 plus inline addition	MLevel 2 plus inline addition
4	MLevel 3 plus offset calculations moved outside of the loops	(Not implemented)
5	MLevel 4 plus assume non-null cells contain numbers	(Not implemented)
6	MLevel 5 plus eliminate virtual calls	(Not implemented)

Figure 5.8: Runtime code generation levels for matrix arguments in TinyCalc.

The DOUBLESUM representation assumes that the formulas of a cell area contain only numbers, and replaces the Value[,] result representation with `double[,]`. It therefore avoids allocating 75.5 million NumberValue objects and thereby halves the TinyCalc recalculation time, although it still allocates 12288 arrays of `doubles`; see figure 5.9.

The FASTSUM representation returns the result of the cell area argument as a view of rectangular area of a sheet, represented as a triple `(sheet, ulCa, lrCa)` of the sheet and the upper left and lower right corners of the area. This further reduces the MLevel 0 recalculation time for TinyCalc so that it is faster than both Gnumeric and OpenOffice, but still 3.4 times slower than Excel. Only by further inlining the addition operation (MLevel 3), by moving loop-invariant code (MLevel 4), by assuming cells are well-typed (MLevel 5), and by eliminating a virtual call to the Sheet indexer's get accessor (MLevel 6), TinyCalc reaches roughly the same speed as Excel, which really shines in this benchmark.

5.2.3 Discussion of results

In conclusion, the above measurements show that already TinyCalc level 0, and hence the baseline implementation of CoreCalc, is usually faster than open source spreadsheet implementations such as Gnumeric and OpenOffice. Moreover, for com-

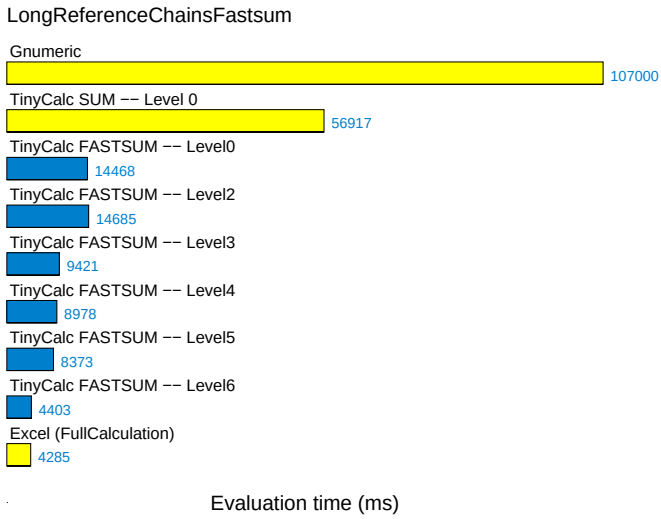

Figure 5.9: Evaluating a sheet with long dependencies and sum functions, using lean matrix result representations in TinyCalc. From Iversen [44].

plex formulas, runtime code generation can provide considerable speed-ups, often giving recalculation times comparable to those of Excel.

The main exception is functions that take cell areas or matrices as arguments, whose relatively poor performance can be attributed to two factors. The first factor is that the representation of argument values in TinyCalc and CoreCalc is overly general. But even when restricting to a more standard representation, the best result is only within a factor of two of Excel. Thus there must be a second factor, which may be that Excel uses the "bare" IEEE754 number representation instead of an object-oriented representation of runtime values; see section 5.3.

A tentative conclusion from Thomas Iversen's experiments is that a spreadsheet program can be implemented as managed code (C#) while roughly maintaining the speed of a highly engineered and widely used commercial spreadsheet implementation written in C or C++. Runtime code generation to the .NET intermediate language seems to offer nearly the same advantages as runtime generation of x86 machine code — something we believe Excel uses, see section 5.3 below.

Note, however, that the benchmark examples used above represent a form of "worst case" examples for Excel, OpenOffice and Gnumeric. Namely, in each recalculation, every single formula had to be recalculated because all cells transitively depend on the cell that was changed. It is likely that Excel, and perhaps also Gnumeric and OpenOffice, have some way of recalculating only those cells that depend on the changed cell (and cells containing calls to volatile functions). CoreCalc and TinyCalc currently do not, and so may perform much worse than the others when

only a fraction of all cells depend on the changed ones. Section 3.3 and chapter 4 discuss how CoreCalc could be enhanced to limit recalculation to those cells that actually depend on changed or volatile cells.

Another mystery is the high time consumption of Excel's `CalculateFullRebuild` (called FullCalculationRebuild in this section's figures), which rebuilds a dependency tree for the workbook; see section 3.3.5.

5.3 Related work

Runtime code generation from spreadsheet formulas is not a new idea, as shown by the Schlafly patents, numbers 194 and 213 in appendix A, which we discovered after doing the work presented above. The patents describe the generation of x86 machine code from spreadsheet formulas, were originally assigned to Borland, and were presumably exploited in Borland's QuattroPro spreadsheet. A remarkable feature of the patents is that they exploit the NaN values of the IEEE754 floating-point representation to avoid any wrapping of runtime values.

It is plausible that a similar runtime code generation technique and value representation is used in Excel but not Gnumeric and OpenOffice, and that that accounts for the considerable speed advantage of Excel. Moreover, Schlafly's patent states that the runtime code generation technique is not used when formula contains transcendental functions such as `SIN`. This might explain how, without any use of runtime code generation, TinyCalc level 0 and hence CoreCalc can outperform Excel on the formula `=SIN(A1)`, as shown in figure 5.2.

Chapter 6

Sheet-defined functions

Simon Peyton Jones, Alan Blackwell and Margaret Burnett proposed in a 2003 paper [70] and in patent application 91 that user-defined functions should be definable as so-called *function sheets* using ordinary spreadsheet formulas. Their goal was to allow "lay" spreadsheet users to define their own functions without forcing them to use a separate programming language such as VBA. Rather, functions should be definable using familiar concepts such as sheets, formulas, references, and so on. To accommodate sheet-defined matrix functions, a spreadsheet cell should be allowed to contain an entire matrix. These ideas are the subject of a patent application, number 91 in appendix A, by the same authors.

Rather similar ideas seem to be incorporated in Nuñez's Scheme-based spreadsheet system ViSSh [64, section 5.2.2], subject of his 2000 MSc thesis. However, ViSSh generalizes and modifies the spreadsheet paradigm in many other ways and would not appear familiar to most Excel users. Hence it would possibly fail some of the design goals of Peyton Jones et al.

6.1 Sheet-defined functions based on CoreCalc

Daniel S. Cortes and Morten W. Hansen [20] in their MSc thesis written at the IT University of Copenhagen set out to further elaborate and implement the concept of sheet-defined functions. This section summarizes findings from that project.

The project proceeded by developing four prototypes, described in the subsections below. All are based on a version of the CoreCalc code base.

6.1.1 Prototype 1: Atomic values, fixed argument count

- Allow the creation of a function sheet. The declaration of a function sheet named `F` must state in a named range `F_signature` which function sheet cells hold the function's arguments and which cell holds its result.
- Allow import of formulas and so on from other sheets.

- Allow definition of functions that have a fixed number of arguments, where the argument values are atomic (number or text) and where the result value is atomic.

6.1.2 Prototype 2: Atomic values, variable argument count

- Allow definition of functions that have a variable number of arguments (the number being known only when the function is called), with argument values and the result value being atomic.

- Allow function sheets to contain references to other sheets in the workbook.

- Allow non-strict argument passing, so that function arguments are evaluated only if actually referred by the sheet-defined function, as for the built-in function `IF`.

6.1.3 Prototype 3: Matrix values

- Allow definition of matrix functions, whose argument values and result value may be atomic or matrices. A matrix argument is passed as a single object in a cell of the function sheet.

6.1.4 Prototype 4: Recursive and higher order functions

- Allow the definition of function sheets that can invoke themselves recursively. This is a natural fit for the declarative computation model of spreadsheets and fairly easy to implement. Peyton Jones et al. [70] consider that recursive sheets should not be permitted, as they would cause confusion when inspecting call relations between function sheets; Cortes and Hansen proposes using a call stack metaphor to avoid such confusion.

- Allow a formula to evaluate to a function object. A formula, such as `=SIN`, that simply mentions a function will evaluate to a function object. Moreover, there is a new built-in function `BIND` to create function closures by binding the first n arguments of a given function. For instance, `BIND(POWER, 3)` evaluates to the function $y \mapsto 3^y$. The resulting function object can be invoked using another built-in function, `CALL`, on the function object and the remaining arguments. The defining equation is that `CALL(BIND(`$f, e_1, \ldots, e_m$`), `$e_{m+1}, \ldots, e_n$`)` computes the same value as $f(e_1, \ldots, e_n)$.

- Allow the definition of higher order functions by function sheets. That is, allow argument values and result value to be function objects. This goes well together with matrix functions, because higher-order functions such as `map` and `fold` can naturally be used to process matrix rows and columns.

6.1.5 Summary of Cortes and Hansen's work

The thesis report of Cortes and Hansen [20] demonstrates the utility of all four prototypes by application case studies, mostly actuarial computations relevant to the life insurance business. In all cases, sheet-defined functions led to conceptual and practical simplifications.

The prototype implementations do not provide a user interface for defining function sheets. Instead, function sheets can be defined in Excel, exported in XMLSS format, and imported into the prototype implementations. When defining a function `F`, the defining sheet must be called `@F`, and the function's signature, which specifies the argument and result cells, must be given in a named range called `F_signature`. This simple mechanism works very well, but should be replaced by a more integrated user interface in a realistic implementation. However, the internal machinery for evaluation of sheet-defined functions is complete enough to run substantial examples.

In summary, the thesis by Cortes and Hansen shows that sheet-defined functions can be added in a natural way to spreadsheet programs while maintaining their familiar look and feel.

6.2 Sheet-defined functions within Excel

As an alternative to using the CoreCalc codebase, Quan Vi Tran and Phong Ha investigated how to implement function sheets using the infrastructure provided by Microsoft Excel [40]. This section summarizes their work.

6.2.1 Implementation approach

Tran and Ha created a plug-in for Excel that allows users to define functions using ordinary Excel sheets, and to call them as if they were defined as macros using VBA. Other Excel macros would evaluate and copy the function's arguments to designated input cells in the function sheet, evaluate the sheet, and copy back the result.

However, this task turned out to be needlessly complicated because of restrictions imposed by Excel. Those restrictions mean that a VBA-defined function cannot have side effects such as updating cells in a different sheet, and conversely, VBA-defined macros, that can have such side effects, cannot be called from spreadsheet formulas. The restrictions themselves are sensible as they prevent VBA-defined functions from undermining the Excel recalculation strategy. However, the restrictions can be circumvented using so-called events, and events can be used in a rather roundabout and non-trivial way to achieve what is outlined above.

The implementation exploits that spreadsheet formulas are side effect free and uses a form of memoization, building a cache of (argument, result) pairs for the sheet-defined function. Apparently this is necessary to correctly evaluate a formula that contains nested calls to sheet-defined functions; the implementation will evaluate such formulas multiple times.

6.2.2 Summary of Tran and Ha's work

In Tran and Ha's work, a functioning implementation was built, which could demonstrate the definition and use of sheet-defined functions within Excel, but benchmarks showed that the implementation is considerably slower than calling functions defined using VBA. The overhead is most likely due to the complicated means needed to circumvent restrictions in Excel.

6.3 Summary of sheet-defined functions

From this chapter we conclude that the concept of sheet-defined function [70] holds considerable promise in modularizing spreadsheet applications and making them more reusable and maintainable. Also, users without programming background are reported to find the concept natural and attractive.

The work of Cortes and Hansen presented in section 6.1 shows that a coherent design for sheet-defined functions can be implemented on top of the CoreCalc code base and can be applied to real-world problems.

The work of Tran and Ha presented in section 6.2 shows that although most of the necessary machinery for implementing sheet-defined functions must be present inside Excel, it is complicated and inefficient to do so using only Excel's public function interfaces.

Chapter 7

Extensions and projects

This chapter lists possible extensions to CoreCalc, and projects that could be undertaken based on CoreCalc.

7.1 Moving and copying cells

- Check that a formula about to be edited, or about to be cut (Ctrl-X), or about to be copied into or pasted into, is not part of a matrix formula.
- Before copy and move operations on a cell or cell area, one can inspect the cell or the cell area's border cells for matrix formulas; if any matrix formula straddles the border, the copy or move should be rejected.
- Before row or column insertion, check whether the insert would split a matrix formula, and reject the operation in that case.
- Before a row or column deletion, one should check whether the deletion would affect a matrix formula, and reject it in that case.
- Moving of formulas is not fully implemented: The adjustment of references previously to the donor cell has not been implemented. Implementing it should preserve the sharing of virtual copies of formulas.
- Maybe implement a general sharing-preservation mechanism (hash-consing style) instead of handling all the cases individually?

7.2 Evaluation mechanism

- Add new "efficient" specialized subclasses to Expr. For instance, arithmetic operators such as + could be represented by separate classes rather than the general `FunCall` class, thus avoiding argument array creation, delegate calls and other runtime overhead.

- Also, one could perform type checks while building such specialized expression representations, thus avoiding the overhead of building NumberValue wrappers and allocating them.
- However, inter-cell type checks arising from this would require invalidation of such an "efficient" expression when cells that it depends on are edited to contain a different type of value. An explicit support graph would enable the system to efficiently find the cells possibly affected by such an edit operation.
- The support graph enables further optimizations to the evaluation mechanism, see section 4.7. Assuming an acyclic support graph, one can schedule recalculation so that the evaluation of each cell can assume that any cell it refers to has already been evaluated. This means that the generated code can avoid some checks, which would seem to open new opportunities for inlining and generation of efficient (real) machine code at the JIT compiler level.
- Implement support for very large but sparse sheets. For instance, instead of using a single two-dimensional array of cells, use a more clever data structure.
- We should have more benchmarks on more realistic workbooks, containing a richer mixture of dependency types and dependency directions.
- Consider multiprocessor recalculation for spreadsheets. Shared memory multiprocessors (SMP) are become very widespread on desktops, but also Cell-style multiprocessor machines, and even FPGA-based "soft hardware" architectures would be interesting to exploit [51]. It is much easier to implement a spreadsheet on such hardware than it is to implement general programs written in C, C++, Fortran, Java, C# or similar. In particular, an explicit support graph could help scheduling of operations, whether to avoid lock contention on standard SMP or scheduling on an FPGA.

7.3 Graphical user interface for CoreCalc

- Add support for key-based navigation, and for using arrow keys or mouse pointing to mark cells or areas.
- Make the user interface more complete, to support more of the non-patented features from Microsoft Excel. Test it systematically.

7.4 Additional functions

7.4.1 Reflective functions

Most spreadsheet programs have reflective capabilities, via so-called "spreadsheet" functions. For instance, `INDEX(A10:C13; r; c)` returns the value of row *r* column *c* of the cell area A10:C13. Some of these functions, such as `ROWS(C7)`, are subtler than

in Excel or OpenOffice because in CoreCalc a cell reference `C7` may refer to a cell that contains a formula `A1:B2` whose value is a matrix. In this case one would need to evaluate the argument expression `C7`, but does one need to evaluate the `A1:B2` cellarea also? That would create a circularity in case the `ROWS(C7)` formula is in one of the cells in `A1:B2` – but in Excel and OpenOffice it does not. Avoiding this strictness would require a different recalculation mechanism than the one we have now.

7.4.2 Additional function-related built-in functions

Additional built-in functions may be just first-order, in which case `fun` below must be the name of a function, or higher-order, in which case `fun` may be any expression that evaluates to a function value. Some possible function-related built-in functions:

- Add to CoreCalc some of the higher-order function machinery implemented by Cortes and Hansen (section 6.1.4).
- `MAP(fun, e0)` applies `fun` to `e0` and produces a result of the same shape as `e0`: matrix-valued if `e0` is matrix-valued etc. Prerequisites: `fun` is a one-argument function, and may be matrix-valued.
- `MAP2(fun, e0, e1)` which applies `fun` to corresponding pairs of values of `e0` and `e1`, producing a result of the same shape as them; matrix-valued if value of `e0` is matrix-valued etc. Prerequisites: `fun` is a two-argument function and may be matrix-valued, and the values of `e0` and `e1` have the same shape.
- `FOLD(fun, e0, e1)` folds `fun` over `e1` with `e0` as starting value. Prerequisites: `fun` is a two-argument function and may be matrix-valued, the value of `e1` is a matrix of values of the same type as `e0`. Built-in functions `SUM` and `PRODUCT` could be defined using `FOLD` over `(PLUS, 0)` and `(TIMES, 1)` respectively.
- `FMATRIX(fun, cols, rows)` returns a cols-by-rows matrix each of whose elements are produced by applying `fun` to pairs (c,r) of column numbers and row numbers $c = 1,\ldots,$`cols` and $r = 1,\ldots,$`rows`; where `cols` and `rows` must be non-negative.

7.4.3 Additional matrix functions

- `VMATRIX(e0, cols, rows)` returns a cols-by-rows matrix all of whose elements equal the value of `e0`; where cols and rows must be non-negative.
- `INDEX(e0, e1, e2)` returns the value in column `e1` row `e2` of `e0`. Prerequisites: `e0` is matrix-valued and `e1` and `e2` are number-valued. Could extend this to work also when `e1` and `e2` are one-dimensional matrices of columns or rows.

  `INDEX(VMATRIX(v,c,r),1,1)` is the same as `v`. `INDEX(FMATRIX(fun,c,r),1,1)` is the same as `APPLY(fun,1,1)`.

- `COLINDEX(e0, e1)` returns column number `e1` of `e0`. Prerequisites: `e0` is matrix-valued and `e1` is number-valued. Could extend this to work also when `e1` is a one-dimensional matrix.
- `ROWINDEX(e0, e1)` returns row number `e1` of `e0`. Prerequisites: `e0` is matrix-valued and `e1` is number-valued. Could extend this to work also when `e1` is a one-dimensional matrix.
- `HCAT(e0, ..., en)` horizontally concatenates the values `v0...vn` of the arguments (side-by-side), returning a matrix value. Each `ei` must either evaluate to an atomic value or to a matrix value. All matrix values among `v0...vn` must have the same number of rows. An atomic value among `v0...vn` will be replicated to make a one-column matrix with that number of rows.
- `VCAT(e0, ..., en)` horizontally concatenates the values v0...vn of the arguments (one atop the next one), returning a matrix value. Each `ei` must either evaluate to an atomic value or to a matrix value. All matrix values among `v0...vn` must have the same number of columns. An atomic value among `v0...vn` will be replicated to make a one-row matrix with that number of columns. This can be used to add a new constant row to a matrix.

7.5 Other project ideas

- Formalize a semantics for spreadsheets. Such a formalization should probably include a logical specification of sheet (and workbook) *consistency* after recalculation, as well as a more operational specification of *what actions* must lead to a recalculation. The semantics should avoid modelling *how* recalculation is performed, leaving the implementation as much latitude in this respect as possible.
- Implement a version of CoreCalc that exploits the IEEE floating-point number representation, and especially the NaNs, in a way similar to Schlafly's patents (numbers 194 and 213) to avoid object wrapping of values. Investigate whether this can be made compatible with (a) matrix-valued functions, (b) runtime-time bytecode generation in Chapter 5, and (c) sheet-defined functions as in Chapter 6.
- Augment the CoreCalc source code with non-null specifications, exception specifications, invariants, and pre- and post-specifications using Spec# [83].
- Implement import of other workbook formats, such as Open Document Format [34] or Office Open XML (see Ecma [28] TC45).
- To handle named sheets, add a Dictionary<String,Sheet> to class Workbook.
- To handle (absolute) named cells and cell areas as in Excel and OpenOffice, add a Dictionary<String,CellAddr> or Dictionary<String,Pair<CellAddr,CellAddr>> mapping to class Sheet.

- Could a cell contain an actual sheet, complete with formulas, or just a formula that happens to be matrix valued? The latter would seem sufficient if new (matrix) functions can be defined by the user as sheets.

- Add serious matrix functions. Maybe using code from Lutz Roeder's Mapack for .NET [73]. That code has SVD but seems somewhat old and untested, although it could at least be used for a sanity check of other implementations. Or maybe from Christoph Rüegg's developing Math.Net project [78]. Another possibility is to port something from the CERN Colt library for Java [18], or check out other Java code at NIST Java Numerics [65].

- Add serious statistics functions, presumably by translating Java code from the CERN Colt library [18], other Java code at NIST Java Numerics [65], or code from Statlib. Consider also the results of McCullough's comparison between Excel and Gnumeric statistical functions [55].

- Implement a good `GOAL.SEEK` function to perform numerical equation solving, that is, find zeroes of functions. This makes it possible to design sheets that have non-trivial computational demands, because `GOAL.SEEK` is likely to evaluate the object function a many times before arriving at a result.

- Find or implement a good `SOLVE` function to perform optimization in multiple dimensions. Linear programming, possibly bounded optimization problems, possibly non-linear problems, and possibly integer problems. This makes for sheets that have high computational demands, and for which a support graph-based minimal recalculation mechanism would be extremely valuable. Possibly base it on the Java code from http://www1.fpl.fs.fed.us/optimization.html, or directly on Fortran Minpack code from Netlib [62] that must then be translated to C#, or on some BFGS implementation.

- Consider how the recalculation in a spreadsheet could be staged [47]. For instance one could choose a set of random parameters, such as the environment of a simulated population, in the first stage (using `RANDOM` or similar). Then the second stage may perform a large number of simulations, such as the growth of the population, involving further random numbers, while keeping the stage one environment parameters fixed. Currently there seems to be no obvious mechanism for this in spreadsheet programs.

Appendix A

Patents and applications

This is a list of US patents (numbered USnnnnnnn) and US patent applications (label USyyyynnnnnn) in which the word "spreadsheet" appears in the title or abstract. Documents that were obviously not about spreadsheet implementation have been omitted from the list, but probably some documents remain that only *use* spreadsheets for some purpose. The list was created on 26 July 2006 by a search on Espacenet [66], and is presented in reverse order of date of inclusion in the Espacenet database. The date shown below is the date granted for patents, and the date of submission for applications. Misspellings in document titles have been preserved.

The full text of the patent documents themselves can be obtained in PDF from the European Patent Office [66] and in HTML from the US Patents and Trademarks Office [87]. In both cases, simply do a "number search" using the patent number USnnnnnnn or the patent application number USyyyynnnnnn.

Documents marked with an asterisk (*) are discussed in the main text. In most cases we give a brief summary of each patent or patent application below. We take no responsibility for the legal or technical correctness of those summaries.

1. Embedded ad hoc browser web to spreadsheet conversion control; US2006156221; 2006-07-13. By Yen-Fu Chen , John Handy-Bosma and Keith Walker. A web browser plug-in that allows any displayed HTML table to be turned into a spreadsheet component.
2. Method, system, and computer-readable medium for determining whether to reproduce chart images calculated from a workbook; US2006136535; 2006-06-22. By Sean Boon, application by Microsoft. Using a hash value of data to avoid re-creating a chart when data are unchanged.
3. Method, system, and computer-readable medium for controlling the calculation of volatile functions in a spreadsheet; US2006136534; 2006-06-22. By Sean Boon, application by Microsoft. How to use time stamps to mostly avoid needless recalculation of volatile functions that vary only slowly, such as `TODAY()`.
4. Block properties and calculated columns in a spreadsheet application; US2006136808; 2006-06-22. By Joseph Chirilov and others; application by Microsoft. How to prescribe properties, such as formatting, for blocks, where a block is a logical area of a spreadsheet that grows or shrinks as rows and columns are added to or removed from it.

5. System and method for automatically completing spreadsheet formulas; US2006129929; 2006-06-15. By Brandon Weber and Charles Ellis; application by Microsoft. Proposing possible completiong of a partially entered formula, in the style of "intellisense" as known from integrated development environments.
6. Method and system for converting a schema-based hierarchical data structure into a flat data structure; US2006117251; 2006-06-01. By Chad Rothschiller, Michael McCormack and Ramakrishnan Natarajan; application by Microsoft. Using schema-derived layout rules to allocate the elements of an XML document to cells in a spreadsheet.
7. Method and system for inferring a schema from a hierarchical data structure for use in a spreadsheet; US2006117250; 2006-06-01. By Chad Rothschiller and others; application by Microsoft. Inferring a schema for XML data stored in a spreadsheet program.
8. System and method for performing over time statistics in an electronic spreadsheet environment; US2006117246; 2006-06-01. By Frederic Bauchot and Gerard Marmigere; application by IBM. Computing statistics from a stream of values, appearing one value at a time in a particular cell.
9. Importing and exporting markup language data in a spreadsheet application document; US2006112329; 2006-05-25. By Robert Collie and others; application by Microsoft. Processing and using XML maps and XML schemas in a spreadsheet program.
10. Method for expanding and collapsing data cells in a spreadsheet report; US2006107196; 2006-05-18. By Lakshmi Thanu and others; application by Microsoft. Showing and hiding subitems in a report, as generated by the subtotals and group-and-outline features of Excel.
11. Method, system, and apparatus for providing access to asynchronous data in a spreadsheet application program; US7047484; 2006-05-16. By Andrew Becker and others; patent assigned to Microsoft. Protocol for reading and using external data streams from a spreadsheet.
12. Spreadsheet application builder; US2006101391; 2006-05-11. By Markus Ulke, Kai Wachter and Gerhild Krauthauf. Application development by drag-and-drop in a spreadsheet style development environment.
13. Error correction mechanisms in spreadsheet packages; US2006101326; 2006-05-11. By Stephen Todd; application by IBM. Introduces pairs of a referencing array and a bound array. If a cell within a referencing array refers to some cell outside the corresponding bound array, an error is signaled.
14. Embedded spreadsheet commands; US2006095832; 2006-05-04. By Bill Serra, Salil Pradhan and Antoni Drudis; application by Hewlett-Packard. Store commands in the comment fields of spreadsheet cells, and interpreting those commands as ties to external events, thus making the spreadsheet update itself – for instance, when a signal from an RFID device indicates that an item has been moved.
15. Method and apparatus for automatically producing spreadsheet-based models; US2006095833; 2006-05-04. By Andrew Orchard and Geoffrey Bristow. A way to describe expandable formulas. Possibly similar ideas as in application 69 and Gencel [31].
16. Program / method for converting spreadsheet models to callable, compiled routines; US2006090156; 2006-04-27. By Richard Tanenbaum. Closely related to application 46.
17. Two pass calculation to optimize formula calculations for a spreadsheet; US2006085386; 2006-04-20. By Lakshmi Thanu, Peter Eberhardy and Xiaohong Yang; application by Microsoft. How to efficiently access external data such as relational databases for OLAP queries and similar.

18. Method and system for enabling undo across object model modifications; US2006085486; 2006-04-20. By Lakshmi Thanu, Peter Eberhardy and Vijay Baliga; application by Microsoft. Improved undo mechanism using two stacks of acts.
19. Methods, systems and computer program products for processing cells in a spreadsheet; US2006080595; 2006-04-13. By Michael Chavoustie and others. Describes a kind inlining of expressions from referred-to cells.
20. Methods, systems and computer program products for facilitating visualization of interrelationships in a spreadsheet; US2006080594; 2006-04-13. By Michael Chavoustie and others. Various ways to display parts of the dependency graph or support graph.
21. Design of spreadsheet functions for working with tables of data; US2006075328; 2006-04-06. By Andrew Becker and others; application by Microsoft. Using database-style queries on named tables in spreadsheets. Related to application 23.
22. One click conditional formatting method and system for software programs; US2006074866; 2006-04-06. By Benjamin Chamberlain and others; application by Microsoft. Using logical conditions to control formatting of spreadsheet cells, and using graphical components (such as histograms) in spreadsheet cells.
23. Method and implementation for referencing of dynamic data within spreadsheet formulas; US2006069696; 2006-03-30. By Andrew Becker and others; application by Microsoft. Notation for referring to tables by symbolic name in spreadsheets, as well as parts of tables and data computed from tables. Related to application 21.
24. * Method and system for multithread processing of spreadsheet chain calculations; US2006069993; 2006-03-30. By Bruce Jones and others; application by Microsoft. Describes multiprocessor recalculation of spreadsheet formulas, and as a side effect, also describes a uniprocessor implementation, probably similar to that of Excel.
25. Graphically defining a formula for use within a spreadsheet program; US2006053363; 2006-03-09. By Christopher Bargh, Gregory Johnston and Russell Jones. How to call a function defined using an external graphical tool.
26. Management of markup language data mappings available to a spreadsheet application workbook; US7007033; 2006-02-28. By Chad Rothschiller and others; application by Microsoft. Processing and using XML maps and XML schemas in a spreadsheet program.
27. Logical spreadsheets; US2006048044; 2006-03-02. By Michael Genesereth, Michael Kassoff and Nathaniel Love. A spreadsheet in which logical constraints on the values of cells can be specified, and the values of cells can be set in any order, possibly restricting or conflicting with values in other cells. Binary decision diagrams [12] would seem ideal for implementing this in the case of discrete cell values.
28. Support for user-specified spreadsheet functions; US2006036939; 2006-02-16. Craig Hobbs and Daniel Clay; application by Microsoft. Permits a user to define a function with named parameters in a spreadsheet cell and call it from other cells. Calls have the syntax `F(funcell, "arg1name", arg1, ..., "argNname", argN)`. Within the function cell, an argument is referred to using the expression `R("argname")`.
29. Method, system, and apparatus for exposing workbooks as data sources; US2006024653; 2006-02-02. By Daniel Battagin and others; application by Microsoft. Appears related to application 40.
30. Method and apparatus for integrating a list of selected data entries into a spreadsheet; US2006026137; 2006-02-02. By Juergen Sattler and others. Appears related to application 37.

31. Sending a range; US2006020673; 2006-01-26. By Terri Sorge and others; application by Microsoft. Facility in a spreadsheet program to extract data from a spreadsheet, format it and automatically send it by email to an indicated recipient.
32. Method and system for presenting editable spreadsheet page layout view; US2006015804; 2006-01-19. By Kristopher Barton, Aaron Mandelbaum and Tisha Abastillas; application by Microsoft. Indicating spreadsheet page layout while maintaining the ability to edit sheets.
33. Transforming a portion of a database into a custom spreadsheet; US2006015525; 2006-01-19. By Jo-Ann Geuss and others. Creating a spreadsheet from a database view.
34. Networked spreadsheet template designer; US2006015806; 2006-01-19. By Wallace Robert. Plug-in for designing spreadsheet templates.
35. Client side, web-based spreadsheet; US6988241; 2006-01-17. By Steven Guttman and Joseph Ternasky; assigned to IBM. Describes a spreadsheet that can be run in a browser, permitting people collaborate and share spreadsheets on the web, and permitting the sheet to use real-time data from the web (such as stock quotes). How similar is Google's recently announced web-based spreadsheet to this?
36. System and method for role-based spreadsheet data integration; US2006010118; 2006-01-12. By Juergen Sattler and Joachim Gaffga. Closely related to application 37.
37. System and method for spreadsheet data integration; US2006010367; 2006-01-12. By Juergen Sattler and Joachim Gaffga. Interfacing spreadsheet program with server data, using access control. Closely related to application 37.
38. System and method for automatically populating a dynamic resolution list; US2006004843; 2006-01-05. By John Tafoya and others; application by Microsoft. Closely related to application 50.
39. Method and apparatus for viewing and interacting with a spreadsheet from within a web browser; US2005268215; 2005-12-01. By Daniel Battagin and Yariv Ben-Tovim; application by Microsoft. Using server-side scripts and a server-side spreadsheet engine to generate HTML, possibly with scripts for inactivity, that when displayed in a client-side browser will provide a spreadsheet user interface.
40. Method, system, and apparatus for exposing workbook ranges as data sources; US2005267853; 2005-12-01. By Amir Netz and others; application by Microsoft. Accessing parts of a spreadsheet document using the same interface (such as ODBC) as for a database.
41. Representing spreadsheet document content; US2005273695; 2005-12-08. By Jeffrey Schnurr. Transmitting part of a spreadsheet to display it on a mobile device.
42. Worldwide number format for a spreadsheet program module; US2005257133; 2005-11-17. By Marise Chan and others; application by Microsoft. Using locale metadata to control the conversion from a numeric time value (as used in spreadsheet programs, see section 2.13.3) to a displayed date appropriate for the user: month names, weekday names, Gregorian or non-Gregorian calendar, and so on.
43. System and method for OLAP report generation with spreadsheet report within the network user interface; US2005267868; 2005-12-01. By Herbert Liebl, Inbarajan Selvarajan and Lee Harold; application by Microstrategy. Presenting server-side enterprise data from an OLAP cube, using a spreadsheet interface on the client side.
44. Method and apparatus for spreadsheet automation; US2005273311; 2005-12-08. By Robert Lautt and Zoltan Grose; application by A3 Solutions. Integrating spreadsheet

models with enterprise data, to avoid inconsistent data, replication of work, manual re-integration of spreadsheet models.

45. * Method for generating source code in a procedural, re-entrant-compatible programming language using a spreadsheet representation; US2005188352; 2005-08-25. By Bruno Jager and Matthias Rosenau. Describes a method to implement database queries by compiling a spreadsheet to source code in a procedural language.
46. Program / method for converting spreadsheet models to callable, compiled routines; US2005193379; 2005-09-01. By Richard Tanenbaum. How to compile spreadsheet formulas to C source code. Closely related to application 16.
47. Reporting status of external references in a spreadsheet without updating; US2005097115; 2005-05-05. By Jesse Bedford and others; application by Microsoft. Closely related to applications 48 and 136.
48. Reporting status of external references in a spreadsheet without updating; US2005108623; 2005-05-19. By Jesse Bedford and others; application by Microsoft. Describes how to check existence of external workbooks and so on before attempting to update links to them. Closely related to applications 47 and 136.
49. Method of updating a database created with a spreadsheet program; US2005149482; 2005-07-07. By Patrick Dillon; application by Thales. Ensuring the correctness of database updates performed from a spreadsheet.
50. System and method for facilitating user input by providing dynamically generated completion information; US2005108344; 2005-05-19. By John Tafoya and others; application by Microsoft. Dynamic input completion based on multiple data sources such as sent and received emails, text documents and other spreadsheet files. Closely related to application 38.
51. System and method for integrating spreadsheets and word processing tables; US2005125377; 2005-06-09. By Matthew Kotler and others; application by Microsoft. Closely related to applications 52, 56, 57, 58 and 63.
52. System and method for integrated spreadsheets and word processing tables; US2005055626; 2005-03-10. By Matthew Kotler and others; application by Microsoft. Closely related to applications 51, 56, 57, 58 and 63.
53. Spreadsheet fields in text; US2005066265; 2005-03-24. By Matthew Kotler and others; application by Microsoft. Closely related to applications 54 and 55.
54. Spreadsheet fields in text; US2005044497; 2005-02-24. By Matthew Kotler and others; application by Microsoft. Closely related to applications 53 and 55.
55. Spreadsheet fields in text; US2005044496; 2005-02-24 By Matthew Kotler and others; application by Microsoft. Closely related to applications 53 and 54. Provide individual text elements, such as a text field in an HTML forms, to have spreadsheet functionality: formulas, references to other text elements, and recalculation.
56. System and method for integrating spreadsheets and word processing tables; US2005050088; 2005-03-03. Closely related to applications 51, 52, 57, 58 and 63.
57. User interface for integrated spreadsheets and word processing tables; US2005034060; 2005-02-10. By Matthew Kotler and others; application by Microsoft. Closely related to applications 51, 52, 56, 58 and 63.
58. User interface for integrated spreadsheets and word processing tables; US2005044486; 2005-02-24. By Matthew Kotler and others; application by Microsoft. Closely related to applications 51, 52, 56, 57, and 63.

59. Method and system for handling data available in multidimensional databases using a spreadsheet; US2005091206; 2005-04-28. By Francois Koukerdjinian and Jean-Philippe Jauffret. Extracting data from a database to create a local database, from which a spreadsheet can then draw its data.
60. Storing objects in a spreadsheet; US2005015714; 2005-01-20. By Jason Cahill and Jason Allen; application by Microsoft. A mechanism to store general objects (in addition to numbers, texts and errors) in spreadsheet cells, and to invoke methods on them from other cells. The objects may be external and the method calls performed using COM, so this is probably a generalization of Piersol [71] and Nuñez [64]. Very similar to patent 135.
61. Spreadsheet to SQL translation; US2005039114; 2005-02-17. By Aman Naimat and others; application by Oracle. A form of query-by-example, where a model is developed on sample database data within a spreadsheet program. Then the spreadsheet model is compiled into SQL queries that can be run on the entire database, possibly through a web interface.
62. Modular application development in a spreadsheet using indication values; US2004225957; 2004-11-11. By Ágúst Egilsson. Closely related to application 130 and patent 178.
63. User interface for integrated spreadsheets and word processing tables; US2004210822; 2004-10-21. By Matthew Kotler and others; application by Microsoft. General table architecture providing spreadsheet functionality (formulas, recalculation and so on) also within word processors and other programs, and permitting nested tables. Closely related to applications 51, 52, 56, 57 and 58.
64. Code assist for non-free-form programming; US2005240984; 2005-10-27. By George Farr and David McKnight; application by IBM. Suggesting completions while typing data into a spreadsheet or similar.
65. System and method for schemaless data mapping with nested tables; US2005172217; 2005-08-04. By Yiu-Ming Leung ; application possibly by Microsoft. Handling and displaying nested tables of data, as from an XML document, without the need for a predetermined schema or XML map.
66. System and method for generating an executable procedure; US2005028136; 2005-02-03. By Ronald Woodley. Generating C++ (or other) source code from spreadsheet data (not formulas).
67. Methods of updating spreadsheets; US2005210369; 2005-09-22. By John Damm. How to update a cell by tapping on it and/or selecting from a drop-down list, intended for PDAs.
68. Clipboard content and document metadata collection; US2005203935; 2005-09-15. By James McArdle; application by IBM. An enhanced clipboard collects information about the source of clippings (date, time, source document, URL, or the like) so that such metadata can be saved in the target document along with the pasted text or data.
69. System and method in a spreadsheet for exporting-importing the content of input cells from a scalable template instance to another; US2005015379; 2005-01-20. By Jean-Jacques Aureglia and Frederic Bauchot. Extended mechanism for copying and pasting a range of cells between scalable templates. Is a “scalable template” somehow related to the hex and vex groups of Erwig’s Gencel [31] system?
70. Method and system in an electronic spreadsheet for handling graphical objects referring to working ranges of cells in a copy/cut and paste operation; US2004143788; 2004-07-22. By Jean-Jacques Aureglia, Frederic Bauchot and Catherine Soler; application

possibly by IBM. Extended mechanism for copying and pasting a range of cells and graphical objects.

71. Systems, methods and computer program products for modeling an event in a spreadsheet environment; US2005102127; 2005-05-12. By Trevor Crowe; application by Boeing. Event-driven computation in a spreadsheet program.
72. Compile-time optimizations of queries with SQL spreadsheet; US2004133568; 2004-07-08. By Andrew Witkowski and others; application by Oracle. Closely related to application 73.
73. Run-time optimizations of queries with SQL spreadsheet; US2004133567; 2004-07-08. By Andrew Witkowski and others; application by Oracle. Efficient queries and recalculation in a spreadsheet drawing data from a relational data base; pruning; parallelization; use of a dependency graph. Closely related to application 72. This looks like a rather substantial patent application. **
74. Determining a location for placing data in a spreadsheet based on a location of the data source; US2005097447; 2005-05-05. By Bill Serra, Salil Pradhan and Antoni Drudis; application possibly by Hewlett-Packard. Handling streams of input values, as from multiple external sensors, in a continually updated spreadsheet. Mentions "dependency trees". **
75. Visual programming system and method; US2005081141; 2005-04-14. By Gunnlaugur Jonsson; application by Einfalt EHF. Object-oriented software development from spreadsheets. Seems related to Piersol [71].
76. Extension of formulas and formatting in an electronic spreadsheet; US2004060001; 2004-03-25. By Wayne Coffen and Kent Lowry; application by Microsoft. Closely related to patent 154.
77. Method and apparatus for data; US2005039113; 2005-02-17. By Corrado Balducci and others; application by IBM. Transforming a spreadsheet into server-side components (such as Java servlets) that generate HTML for spreadsheet display in a browser at client-side.
78. System and method for cross attribute analysis and manipulation in online analytical processing (OLAP) and multi-dimensional planning applications by dimension splitting; US2005038768; 2005-02-17. By Richard Morris; application by Retek. Manipulating and displaying hierarchical multi-dimensional data.
79. Flexible multiple spreadsheet data consolidation system; US2005034058; 2005-02-10. By Scott Mills and others; application by SBC Knowledge Ventures. Consolidating multiple spreadsheets into one.
80. Method for generating a stand-alone multi-user application from predefined spreadsheet logic; US2004064470; 2004-04-01. By Kristian Raue; application by Jedox GmbH. Compiling a spreadsheet to web scripts (in PHP) supporting multi-user distributed access.
81. System and method for formatting source text files to be imported into a spreadsheet file; US2005022111; 2005-01-27. By Jean-Luc Collet, Jean-Christophe Mestres and Carole Truntschka; application by IBM. Using a file format profile to guide the import of text files into a spreadsheet program.
82. * Method in connection with a spreadsheet program; US2003226105; 2003-12-04. By Mattias Waldau. Describes a system for compiling a spreadsheet to executable code for a different platform, such as a mobile phone.

83. Methods, systems and computer program products for incorporating spreadsheet formulas of multi-dimensional cube data into a multi-dimentional cube; US2004237029; 2004-11-25. By John Medicke, Feng-Wei Chen Russell, and Stephen Rutledge. Converting a spreadsheet formula into a query on multi-dimensional data.
84. Method of feeding a spreadsheet type tool with data; US2003212953; 2003-11-13. By Jacob Serraf; application by Eurofinancials. Transmitting spreadsheet data on a network.
85. Software replicator functions for generating reports; US2004111666; 2004-06-10. By James Hollcraft. Specifying automatic replication of formulas to grow with data.
86. Method for automatically protecting data from being unintentionally overwritten in electronic forms; US2003159108; 2003-08-21. By Gerhard Spitz. Rules for automatically determining cells whose contents should be protected from overwriting.
87. Methods and apparatus for generating a spreadsheet report template; US2004088650; 2004-05-06. By Brian Killen and others; application by Actuate. Extracting data from a relational database and creating reports in a spreadsheet program.
88. Thin client framework deployment of spreadsheet applications in a web browser based environment; US2004181748; 2004-09-16. By Ardeshir Jamshidi and Hardeep Singh; application by IBM. Client and server collaborating to support a browser-based spreadsheet program.
89. Method and system for the direct manipulation of cells in an electronic spreadsheet program or the like; US2003164817; 2003-09-04. By Christopher Graham, Ross Hunter and Lisa James; application by Microsoft. Describes how keys and mouse can be used to control whether insertion of cell blocks overwrite cells or add new rows and columns. Implemented in Excel. Appears closely related to patent 206.
90. System, method, and computer program product for an integrated spreadsheet and database; US2004103365; 2004-05-27. By Alan Cox. Integrating relational queries in a spreadsheet program, so that a query can create and populate a new worksheet, containing appropriately copied and adjusted formulas .
91. * User defined spreadsheet functions; US2004103366; 2004-05-27. By Simon Peyton Jones, Alan Blackwell and Margaret Burnett; application by Microsoft. Describes the concepts presented also in their paper [70].
92. System and method in an electronic spreadsheet for displaying and/or hiding range of cells; US2003188259; 2003-10-02. Much the same as application 94.
93. System and method in an electronic spreadsheet for displaying and/or hiding range of cells; US2003188258; 2003-10-02. Much the same as application 94.
94. System and method in an electronic spreadsheet for displaying and/or hiding range of cells; US2003188257; 2003-10-02. By Jean-Jacques Aureglia and Frederic Bauchot; application by IBM. Hiding and displaying cells in a multi-dimensional spreadsheet program.
95. System and method in an electronic spreadsheet for copying and posting displayed elements of a range of cells; US2003188256; 2003-10-02. By Jean-Jacques Aureglia and Frederic Bauchot; application by IBM. Cell copy-and-paste in a multi-dimensional spreadsheet when some cells of the source region are hidden.
96. System and method for editing a spreadsheet via an improved editing and cell selection model; US2003051209; 2003-03-13. By Matthew Androski and others; application by Microsoft. Detailed description of editing gestures in a spreadsheet program.

97. System and method for automated data extraction, manipulation and charting; US2004080514; 2004-04-29. By Richard Dorwart. Automatically creating appropriate charts from tabular spreadsheet data, and exporting the charts to a presentation program.
98. System and method for displaying spreadsheet cell formulas in two dimensional mathematical notation; US2003056181; 2003-03-20. By Sharad Marathe. Displaying spreadsheet formulas in usual mathematical notation. This would seem to be what symbolic mathematics programs such as Maple and Mathematica routinely perform.
99. Functions acting on arbitrary geometric paths; US2005034059; 2005-02-10. By Craig Hobbs; application by Microsoft. Functions for a spreadsheet component used to calculate and transform graphical objects as in Microsoft Visio.
100. Data-bidirectional spreadsheet; US2004044954; 2004-03-04. By Michael Hosea; application possibly by Texas Instruments. Interfacing a spreadsheet program to an external calculcation engine, as in an electronic graphical pocket calculcator.
101. * Parser, code generator, and data calculation and transformation engine for spreadsheet calculations; US2003106040; 2003-06-05. By Michael Rubin and Michael Smialek. Describes compilation of spreadsheets to Java source code.
102. Spreadsheet data processing system; US2004205524; 2004-10-14. By John Richter, Christopher Tregenza and Morten Siersted; application by F1F9. A method for processing, not implementing, a spreadsheet.
103. System and method for efficiently and flexibly utilizing spreadsheet information; US2003110191; 2003-06-12. By Robert Handsaker, Gregory Rasin and Andrey Knourenko. Creating and using a family of parametrized spreadsheet workbooks.
104. Method and system for creating graphical and interactive representations of input and output data; US2003169295; 2003-09-11. By Santiago Becerra. Controlling input to spreadsheet cells, and displaying output from them, using graphical components such as charts, sliders, and so on. This was proposed also by Piersol [71] and Nuñez [64].
105. Interface for an electronic spreadsheet and a database management system; US2003182287; 2003-09-25. By Carlo Parlanti. Accessing relational databases from a spreadsheet with ODBC and UDA.
106. Systems and methods providing dynamic spreadsheet functionality; US2002169799; 2002-11-14. By Perlie Voshell. Dynamic report creation in relation to databases.
107. Individually locked cells on a spreadsheet; US2003117447; 2003-06-26. By Gayle Mujica and Michelle Miller; application possibly by Texas Instruments. Allow individual locking of cells (instead of bulk locking and individual unlocking as in Excel), and graphical marking of locked cells.
108. Calculating in spreadsheet cells without using formulas; US2003120999; 2003-06-26. By Michelle Miller and others; application possibly by Texas Instruments. Formula entry on a graphical calculator.
109. Spreadsheet Web server system and spreadsheet Web system; US2002065846; 2002-05-30. By Atsuro Ogawa and Hideo Takata. A server-side spreadsheet component that generates HTML tables for display in a web browser on the client side.
110. Method and system in an electronic spreadsheet for persistently filling by samples a range of cells; US2002103825; 2002-08-01. By Frederic Bauchot; application by IBM. Fill a cell range by sampling and interpolating from existing values.
111. Method and apparatus for handling scenarios in spreadsheet documents; US2002055953; 2002-05-09. By Falko Tesch and Matthias Breuer. Preserving, and displaying a tree

structure, the scenarios explored using a spreadsheet.

112. User interface for a multi-dimensional data store; US2003088586; 2003-05-08. By Alexander Fitzpatrick and Sasan Seydnejad. Spreadsheet user interface to multi-dimensional data, so-called "planning data repository".
113. Methods and systems for inputting data into spreadsheet documents; US2002055954; 2002-05-09. By Matthias Breuer. User input, undo and recalculation based on previous value.
114. Parallel execution mechanism for spreadsheets; US2001056440; 2001-12-27. By David Abramson and Paul Roe. A method for explicitly initiating a parallel computation from a spreadsheet cell; not a parallel implementation of the standard recalculation mechanism.
115. Method and apparatus for entry and editing of spreadsheet formulas; US2003033329; 2003-02-13. By Eric Bergman and Paul Rank. Terminate the editing of a formula on a PDA when user selects a different cell while the cursor in the formula is at a point inappropriate for insertion of a cell reference.
116. Method and system in an electronic spreadsheet for persistently self-replicatig multiple ranges of cells through a copy-paste operation; US2002049785; 2002-04-25. By Frederic Bauchot; application by IBM. Closely related to application 120.
117. Dynamic conversion of spreadsheet formulas to multidimensional calculation rules; US2003009649; 2003-01-09. By Paul Martin, William Angold, and Nicolaas Kichenbrand. Calculating on multidimensional data, as obtained from a relational database. Closely related to application 118.
118. Multidimensional data entry in a spreadsheet; US2002184260; 2002-12-05. By Paul Martin, William Angold, and Nicolaas Kichenbrand. Accessing, displaying, editing and writing back multidimensional data, as obtained from a relational database. Closely related to application 117.
119. Dynamic data display having slide drawer windowing; US2002198906; 2002-12-26. By Robert Press; application by IBM. A graphical display of data in which multiple "drawers", each displaying a fragment of a spreadsheet, may be visible simultaneously, and may automatically resize themselves.
120. Method and system in an electronic spreadsheet for persistently copy-pasting a source range of cells onto one or more destination ranges of cells; US2002049784; 2002-04-25. By Frederic Bauchot; application by IBM. Mechanism to make persistent copies of a formula, so that the copies are automatically updated when the original is updated. Closely related to application 116. This seems similar to Montigel's Wizcell [60].
121. Method and system for automated data manipulation in an electronic spreadsheet program or the like; US2002174141; 2002-11-21. By Shing-Ming Chen . Spreadsheet as database front-end, addressing cells with ranges and cell collections, and recording macros.
122. Method and system in an electronic spreadsheet for comparing series of cells; US2002023106; 2002-02-21. By Bauchot and Daniel Mauduit; application by IBM. Use Boolean functions to determine whether two ranges of cells overlap, are disjoint, are equal or are contained one in the other.
123. Method and system in an electronic spreadsheet for handling user-defined options in a copy/cut - paste operation; US2002007380; 2002-01-17. By Bauchot and Albert Harari; application by IBM. How to control the setting of user-defined options in a cell copying operation, when the options were set for the source range of cells.

124. Method and system in an electronic spreadsheet for managing and handling user-defined options; US2002007372; 2002-01-17. By Bauchot and Albert Harari; application by IBM. How to create or change user-defined options using a table.
125. Method and system in an electronic spreadsheet for applying user-defined options; US2002059233; 2002-05-16. By Bauchot and Albert Harari; application by IBM. How to set options to true or false.
126. The applications US2002143811, US2002143831, US2002143810, US2004205676, US2002143809, US2002140734, US2002143730 and US2002143830 are all by Paul Bennett, Round Rock, Texas, USA:

    - System and method for vertical calculation using multiple columns in a screen display; US2002143811; 2002-10-03.
    - System and method for calculation using spreadsheet lines and vertical calculations in a single document; US2002143831; 2002-10-03.
    - System and method for calculation using vertical parentheses; US2002143810; 2002-10-03.
    - System and method for calculation using a subtotal function; US2004205676; 2004-10-14. Describes a graphical way to specify subtotal computations.
    - System and method for calculation using multi-field columns with hidden fields; US2002143809; 2002-10-03.
    - System and method for calculation using formulas in number fields; US2002140734; 2002-10-03.
    - System and method for calculation using a calculator input mode; US2002143730; 2002-10-03.
    - System and method for calculation using multi-field columns with modifiable field order; US2002143830; 2002-10-03. Unclear what is new relative to the general concept of a spreadsheet.

127. Method and system in an electronic spreadsheet for handling absolute references in a copy/cut and paste operation according to different modes; US2001032214; 2001-10-18. By Frederic Bauchot and Albert Harari; application by IBM. Describes a method and conditions for replacing absolute cell references to the source range by (other) absolute cell references in the target range when copying formulas.
128. Spreadsheet error checker; US2002161799; 2002-10-31. By Justin Maguire; application by Microsoft. A rule-based error checker for individual cells of a spreadsheet.
129. Multi-dimensional table data management unit and recording medium storing therein a spreadsheet program; US2001016855; 2001-08-23. By Yuko Hiroshige. Selecting and manipulating three-dimensional data.
130. Graphical environment for managing and developing applications; US2002010713; 2002-01-24. By Ágúst Egilsson. Closely related to patent 178 and application 62.
131. Method and system for distributing and collecting spreadsheet information; US2002010743; 2002-01-24. By Mark Ryan, David Keeney and Ronald Tanner. Assigning individual sheets of a master workbook to one or more contributors, sending copies of the sheets to the contributors for updating, and reintegrating them into the master workbook.
132. * Method and apparatus for formula evaluation in spreadsheets on small devices; US2002143829; 2002-10-03. By Paul Rank and John Pampuch. Describes compilation of a spreadsheet for space-conserving execution (but not editing) on a PDA.

133. Universal graph compilation tool; US6883161; 2005-04-19. By Andre Chovin and Chatenay Alain; assigned to Crouzet Automatismes. Compilation of a graph of formulas to code for embedded devices.
134. Enhanced find and replace for electronic documents; US2002129053; 2002-09-12. By Marise Chan and others; application by Microsoft. A find-and-replace function that handles multiple sheets in a workbook; can find and change formatting attributes; and can be suspended for editing and later resumed.
135. Storing objects in a spreadsheet; US6779151; 2004-08-17. By Jason Cahill and Jason Allen; assigned to Microsoft. Very similar to application 60.
136. Reporting status of external references in a spreadsheet without updating; US2002091730; 2002-07-11. By Jesse Bedford and others; application by Microsoft. Closely related to applications 47 and 48.
137. Method and apparatus for a file format for storing spreadsheets compactly; US2002124016; 2002-09-05. By Paul Rank and others. Storing a spreadsheet on a PDA in a number of database records.
138. Method for dynamic function loading in spreadsheets on small devices; US2002087593; 2002-07-04. By Paul Rank. On demand loading of functions and features in spreadsheet program for PDAs.
139. Functional visualization of spreadsheets; US2002078086; 2002-06-20. By Jeffrey Alden and Daniel Reaume. Construction, visual display and maintenance of the support graph (chapter 4) or dependency graph, in the appliciaton called "influence diagram". Focus is on the visual display, not on compact representation or efficient construction.
140. Method and system in an electronic spreadsheet for adding or removing elements from a cell named range according to different modes; US2001007988; 2001-07-12. By Frederic Bauchot and Albert Harari; application by IBM. Mechanism for updating referring formulas when rows or columns are added to or deleted from a cell range.
141. Methods and systems for generating a structured language model from a spreadsheet model; US6766512; 2004-07-20. By Farzad Khosrowshahi and Murray Woloshin; assigned to Furraylogic Ltd. Compiling a spreadsheet model to code in a procedural programming language.
142. Method and system in an electronic spreadsheet for introducing new elements in a cell named range according to different modes; US6725422; 2004-04-20. By Frederic Bauchot and Albert Harari; assigned to IBM. Differentiating between closed and open named ranges of cells; the latter can be expanded by insertion of new rows and columns in the open direction.
143. Computerized spreadsheet with auto-calculator; US6430584; 2002-08-06. By Ross Comer and David Williams Jr; assigned to Microsoft. Closely related to patent 181.
144. * Methodology for testing spreadsheet grids; US6766509; 2004-07-20. By Andrei Sheretov, Margaret Burnett and Gregg Rothermel; assigned to University of Oregon. Two methods for using du-associations to test spreadsheets; in the more advanced method, the testing of a single representative cell can increase the testedness of a range of cells containing similar formulas.
145. * Methodology for testing spreadsheets; US6948154; 2005-09-20. By Gregg Rothermel, Margaret Burnett, and Lixin Li; assigned to University of Oregon. Using du-associations to gradually test a spreadsheet, displaying each cell's testedness.
146. Spreadsheet recalculation engine version stamp; US6523167; 2003-02-18. By Timothy Ahlers and Andrew Becker, assigned to Microsoft. Explains how recalculation – or not

– at loading can be controlled by calculation engine version stamp. This technique appears to be used in Excel 2000 and later to enforce a full recalculation when loading a workbook that was last saved by Excel'97 or older, and avoid that recalculation otherwise. (Maybe the intention is to guard against an Excel'97 recalculation flaw; see note under patent 182).

147. Apparatus and method for dynamically updating a computer-implemented table and associated objects; US6411959; 2002-06-25. By Todd Kelsey; assigned to IBM. Automatically copying formulas and extending references to a table when new rows or columns are added. Much the same idea as Microsoft's patent 154.
148. Method and system in an electronic spreadsheet for processing different cell protection modes US6592626; 2003-07-15. By Frederic Bauchot and Albert Harari; assigned to IBM. Changing the protection mode of single cells.
149. Binding data from data source to cells in a spreadsheet; US6631497; 2003-10-07. By Ardeshir Jamshidi, Farzad Farahbod and Hardeep Singh; assigned to IBM. Dynamically importing data from external sources (such as databases), with no need for programming.
150. Binding spreadsheet cells to objects; US6701485; 2004-03-02. By Mark Igra, Eric Matteson and Andrew Milton; assigned to Microsoft. Binding a spreadsheet cell to an external event source, such as a stock ticker, for instance when a spreadsheet program (Excel) runs as component in a web browser (Internet Explorer).
151. Automatic formatting of pivot table reports within a spreadsheet; US6626959; 2003-09-30. By Wesner Moise, Thomas Conlon and Michelle Thompson; assigned to Microsoft. The automatic formatting of finished pivot tables as known from Excel.
152. User interface for creating a spreadsheet pivottable; US6411313; 2002-06-25. By Thomas Conlon and Paul Hagger; assigned to Microsoft. The pivot table user interface as known from Excel.
153. Method and apparatus for organizing and processing information using a digital computer; US6166739; 2000-12-26. By Kent Lowry and others; assigned to Microsoft. Initiate cell editing by two single-clicks rather than one double-click when a spreadsheet program (Excel) runs as component in a web browser (Internet Explorer).
154. Extension of formulas and formatting in an electronic spreadsheet; US6640234; 2003-10-28. By Wayne Coffen and others; assigned to Microsoft. Describes a system by which a previously blank but newly edited cell, which extends a list of consistently typed and formatted cells, will automatically be formatted like those cells and will be included in existing formulas and aggregating expressions that include all of those cells. Closely related to patent application 76.
155. System and method for editing a spreadsheet via an improved editing and cell selection model; US6549878; 2003-04-15.
156. Method and apparatus for accessing multidimensional data; US6317750; 2001-11-13. By Thomas Tortolani and Nouri Koorosh; assigned to Hyperion Solutions. Manipulating and displaying data from an external (database) source, with automatic replication of formulas.
157. Visualization spreadsheet; US2001049695; 2001-12-06. By Ed Chi and others. The authors and the patent seem unrelated to Nuñez [64], but the general idea is the same.
158. Analytic network engine and spreadsheet interface system; US6199078; 2001-03-06. By Philip Brittan and others; assigned to Sphere Software Engineering. A calculation mechanism that attempts to handle circular cell dependencies.

159. Multidimensional electronic spreadsheet system and method; US2002091728; 2002-07-11. By Henrik Kjaer and Dan Pedersen. A three-dimensional spreadsheet in which a usual cell (in the two-dimensional grid) can contain a stack of cells.
160. Visual aid to simplify achieving correct cell interrelations in spreadsheets; US2002023105; 2002-02-21. By Robert Wisniewski. Describes a system for visualizing which cells a given cell depends on, and vice versa.
161. System and methods for improved spreadsheet interface with user-familiar objects; US6282551; 2001-08-28. By Charles Anderson and others; assigned to Borland. Closely related to patent 186.
162. Automatic spreadsheet forms; US5966716; 1999-10-12. By Ross Comer, John Misko and Troy Link; assigned to Microsoft. Closely related to patent 179.
163. Spreadsheet view enhancement system; US6185582; 2001-02-06. By Polle Zellweger; assigned to Xerox. Related to patent 164.
164. Animated spreadsheet for dynamic display of constraint graphs; US6256649; 2001-07-03. By Jock Mackinlay and others; assigned to Xerox. Related to patent 163.
165. System and method for processing data in an electronic spreadsheet in accordance with a data type; US6138130; 2000-10-24. By Dan Adler and Roberto Salama; assigned to Inventure Technologies. Seems related to patent 189 but additionally mentions the Java programming language.
166. Method and system for detecting and selectively correcting cell reference errors; US6317758; 2001-11-13. By Robert Madsen, Daren Thayne and Gary Gibb; assigned to Corel. Changing a reference in a formula from relative to absolute after copying the formula.
167. System for displaying desired portions of a spreadsheet on a display screen by adjoining the desired portions without the need for increasing the memory capacity; US6115759; 2000-09-05. By Kazumi Sugimura and Shuzo Kugimiya; assigned to Sharp. How to hide and later redisplay selected rows and columns.
168. * Constraint-based spreadsheet system capable of displaying a process of execution of programs; US5799295; 1998-08-25. By Yasuo Nagai; assigned to Tokyo Shibaura Electric Co. A spreadsheet based on constraints in addition to formulas.
169. On-screen indentification and manipulation of sources that an object depends upon; US6057837; 2000-05-02. By Darrin Hatakeda and others; assigned to Microsoft. Using colors to indicate the various cell areas that a formula or graph depends on. Implemented in Excel.
170. Method and apparatus for using label references in spreadsheet formulas; US5987481; 1999-11-16. By Eric Michelman, Joseph Barnett and Jonathan Lange; assigned to Microsoft. Using names (symbolic labels) to refer to ranges in a spreadsheet. The intersection of a row name and a column name denotes a cell.
171. Spreadsheet-calculating system and method; US5970506; 1999-10-19. By Hiroki Kiyan, Takaki Tokuyama and Motohide Tamura; assigned to Justsystem Corporation. A cell area can be held fixed when the sheet is scrolled.
172. Method and system for establishing area boundaries in computer applications; US6005573; 1999-12-21. By William Beyda and Gregory Noel; assigned to Siemens. Limiting scrolling and editing in a graphical user interface.
173. System and methods for building spreadsheet applications; US5883623; 1999-03-16. By Istvan Cseri; assigned to Borland. Seems closely related to patent 207.

174. Method and system for linking controls with cells of a spreadsheet; US5721847; 1998-02-24. By Jeffrey Johnson; assigned to Microsoft. Associating graphic controls (a view and a controller) with spreadsheet cells (a model).
175. Method and system for the direct manipulation of cells in an electronic spreadsheet program or the like; US6112214; 2000-08-29. By Christopher Graham, Ross Hunter and Lisa James; assigned to Microsoft. Moving or copying a marked block of cells by dragging its border and using control keys. Implemented in Excel. Appears related to patent 206.
176. Spreadsheet image showing data items as indirect graphical representations; US5880742; 1999-03-09. By Ramana Rao and Stuart Card; assigned to Xerox. Displaying multidimensional data graphically and manipulating the graphs in the user interface.
177. Transformation of real time data into times series and filtered real time data within a spreadsheet application; US5926822; 1999-07-20. By Mark Garman; assigned to Financial Engineering. A spreadsheet program that permit real-time update of cells reflecting a stream of input values.
178. Graphical environment for managing and developing applications; US6286017; 2001-09-04. By Ágúst Egilsson. An extended spreadsheet paradigm in which a spreadsheet may refer to external program fragments and the like. Closely related to applications 62 and 130.
179. Automatic spreadsheet forms; US5819293; 1998-10-06. By Ross Comer, John Misko and Troy Link; assigned to Microsoft. Generating multiple spreadsheet form instances from a template, associating each form with a database row. Closely related to patent 162.
180. Method and apparatus for suggesting completions for a partially entered data item based on previously-entered, associated data items; US5845300; 1998-12-01. By Ross Comer, Adam Stein and David Williams Jr; assigned to Microsoft. How to propose completion of partially typed cell entries from a dynamically updated list.
181. Computerized spreadsheet with auto-calculator; US6055548; 2000-04-25. By Ross Comer and David Williams Jr; assigned to Microsoft. Describes a mechanism to apply a function (such as `SUM`) to a selected cell area, where the user may interactively and graphically change the selection. Excel and other spreadsheet programs implement this functionality for `SUM` (only?), displaying the value in the status bar. Closely related to patent 143.
182. * Method and system of sharing common formulas in a spreadsheet program and of adjusting the same to conform with editing operations; US5742835; 1998-04-21. By Richard Kaethler,, assigned to Microsoft; very similar to patent 204. First, describes a technique to identify identical formulas in a contiguous block of cells, and to share a single representation of the formula between all cells in the block. The need for this presupposes a particular formula representation, which is not made explicit, but which clearly is different from that chosen in CoreCalc. Second, notes that the sharing makes insertion and deletion of entire rows and columns more complicated, should they happen to intersect with a block.

    This problem is the same as that discussed in section 2.16 here, but the patent's solution makes a point of creating small cell blocks, distinguishing between blocks with 1 to 4, 5 to 16, and 16 or more columns; and with 1 to 15, 16 to 31, 31 to 48, and 49 to 200 rows. The point of this is not yet clear.

    Maybe a faulty implementation of this approach caused bugs number KB171339 ("Some values not recalculated when using multiple formulas") and KB154134 ("Functions in

filled formulas may not be recalculated") in Excel'97; see Microsoft Developer Network Knowledge base at http://support.microsoft.com/kb/q174868/.

183. System and methods for reformatting multi-dimensional spreadsheet information; US5604854; 1997-02-18. By Colin Glassey; assigned to Borland. Transforming data from relational to multi-dimensional tabular form, and swapping axes, in the manner of pivot tables.
184. Method and system for detecting and correcting errors in a spreadsheet formula; US5842180; 1998-11-24. By Karan Khanna and Edward Martinez; assigned to Microsoft. Parsing of formula expressions with error recovery and display of dialog box.
185. Method and system for allowing multiple users to simultaneously edit a spreadsheet; US6006239; 1999-12-21. By Anil Bhansali and Rohit Wad; assigned to Microsoft. Describes a kind of concurrent versioning system for multiple users to edit and save the same spreadsheet.
186. System and methods for improved spreadsheet interface with user-familiar objects; US5664127; 1997-09-02. By Charles Anderson and others; assigned to Borland. A workbook containing multiple spreadsheets. Closely related to patent 161.
187. System and methods for improved scenario management in an electronic spreadsheet; US6438565; 2002-08-20. By Joseph Ammirato and Gavin Peacock; assigned to Borland. Closely related to patent 218.
188. Method and apparatus for retrieving data and inputting retrieved data to spreadsheet including descriptive sentence input means and natural language interface means; US5734889; 1998-03-31. By Tomoharu Yamaguchi; assigned to Nippon Electric Co. Translating a natural language phrase to a database query an executing it in a spreadsheet.
189. Computer-based system and method for data processing; US5768158; 1998-06-16. By Dan Adler, Roberto Salama and Gerald Zaks; assigned to Inventure America Inc. A spreadsheet program in which a cell may contain any object.

    Piersol's 1986 paper [71] is mentioned in the application but apparently not considered prior art, because a formula cannot change the value of another cell in Piersol's system.
190. Method and apparatus for entering and manipulating spreadsheet cell data; US5717939; 1998-02-10. By Daniel Bricklin, William Lynch and John Friend; assigned to Compaq Computer. Similar to patent 197.
191. Method and system for constructing a formula in a spreadsheet; US5890174; 1999-03-30. By Karan Khanna and Edward Martinez; assigned to Microsoft. Displaying information about a function and its argument types during formula editing.
192. System and methods for automated graphing of spreadsheet information; US5581678; 1996-12-03. By Philippe Kahn; assigned to Borland. Automatically proposing a graph type (pie chart, curve, 2D or 3D bar chart, ...) based on the number of data points and the complexity of selected data. Similar to patent 202.
193. Method and system for mapping non-uniform table-structure input data to a uniform cellular data structure; US5881381; 1999-03-09. By Akio Yamashita and Yuki Hirayama; assigned to IBM. Pasting a table from a text document into a spreadsheet, such that each spreadsheet cell receives one table item.
194. * Methods for compiling formulas stored in an electronic spreadsheet system; US5633998; 1997-05-27. By Roger Schlafly; assigned to Borland. Related to patent 213.
195. Process and device for the automatic generation of spreadsheets; US5752253; 1998-05-12. By Jean Paul Geymond and Massimo Paltrinieri; assigned to Bull SA. Generating

a spreadsheet from the schema of a relational database.

196. System and method of integrating a spreadsheet and external program having output data calculated automatically in response to input data from the spreadsheet; US5893123; 1999-04-06. By Paul Tuinenga. Using OLE to call an external function from a spreadsheet (when recalculating) and getting the result back into the spreadsheet.
197. Method and apparatus for entering and manipulating spreadsheet cell data; US5848187; 1998-12-08. By Daniel Bricklin, William Lynch and John Friend; assigned to Compaq Computer. How to read and then assign hand-written data to spreadsheet cells. Similar to patent 190.
198. Method and system for automatically entering a data series into contiguous cells of an electronic spreadsheet program or the like; US5685001; 1997-11-04. By Brian Capson and others; assigned to Microsoft. Use mouse and/or keyboard to quickly enter series such as 1, 2, . . . ; or Monday, Tuesday, . . . , as used in Excel.
199. Graphic indexing system; US5867150; 1999-02-02. By Dan Bricklin and others; assigned to Compaq Computer. Similar to patent 209.
200. System and methods for improved spreadsheet interface with user-familiar objects; US5590259; 1996-12-31. By Charles Anderson and others; assigned to Borland. Also published as US5416895.
201. Location structure for a multi-dimensional spreadsheet; US6002865; 1999-12-14. By Erik Thomsen.
202. Systems and methods for automated graphing of spreadsheet information; US5461708; 1995-10-24. By Philippe Kahn; assigned to Borland. Similar to patent 192.
203. Method and system for direct cell formatting in a spreadsheet; US5598519; 1997-01-28. By Raman Narayanan; assigned to Microsoft. Sharing cell formatting information between cells by storing the formatting information in a separate formatting table, and mapping cell coordinates to entries in that table.
204. * Method and system of sharing common formulas in a spreadsheet program and of adjusting the same to conform with editing operations; US5553215; 1996-09-03. By Richard Kaethler, assigned to Microsoft; very similar to patent 182.
205. Methods for composing formulas in an electronic spreadsheet system; US5603021; 1997-02-11. By Percy Spencer and others; assigned to Borland. Displaying information about a function and its argument types during formula editing.
206. Method and system for the direct manipulation of cells in an electronic spreadsheet program or the like; US5623282; 1997-04-22. By Christopher Graham, Ross Hunter and Lisa James; assigned to Microsoft. Appears closely related to application 89.
207. System and methods for building spreadsheet applications; US5623591; 1997-04-22. By Istvan Cseri; assigned to Borland. Linking objects with events and actions.
208. Auto-formatting of tables in a spreadsheet program; US5613131; 1997-03-18. By Ken Moss and Andrew Kwatinetz; assigned to Microsoft. Describes table autoformatting using heuristics, as known from Excel.
209. Graphic indexing system; US5539427; 1996-07-23. By Dan Bricklin and others; assigned to Compaq Computer. Using a lasso gesture in handwriting entry to indicate items to index. Similar to patent 199.
210. System and methods for improved scenario management in an electronic spreadsheet; US5499180; 1996-03-12. By Joseph Ammirato and Gavin Peacock; assigned to Borland. Closely related to patent 218.

211. Code generation and data access system; US5544298; 1996-08-06. By Walter Kanavy and Timothy Brown; assigned to Data Management Corporation. A system to speed up the creation of database queries and the like.
212. Visually aging scroll bar; US5532715; 1996-07-02. By Cary Bates and others; assigned to IBM. Visually "heating" a controlled cell as long as a scrollbar slider remains the same position.
213. * Electronic spreadsheet system and methods for compiling a formula stored in a spreadsheet into native machine code for execution by a floating-point unit upon spreadsheet recalculation; US5471612; 1995-11-28. By Roger Schlafly; assigned to Borland. Comments: Unusually well-written and technically substantial. See sections 1.12 and 5.3.
214. Electronic spreadsheet system producing generalized answers including formulas; US5418902; 1995-05-23. By Vincent West and Edward Babb; assigned to Int Computers Ltd. Translate spreadsheet formulas into logic and evaluate symbolically; allows symbolic and bidirectional computations.
215. Spreadsheet command/function callback capability from a dynamic-link library; US5437006; 1995-07-25. By Andrzej Turski; assigned to Microsoft. Supporting callbacks into a spreadsheet program.
216. Fuzzy spreadsheet data processing system; US5381517; 1995-01-10. By Karl Thorndike and Joseph Vrba; assigned to Fuziware. Computing with fuzzy numbers and displaying fuzzy results in a spreadsheet program.
217. Sorting a table by rows or columns in response to interactive prompting with a dialog box graphical icon; US5396621; 1995-03-07. By Kathryn Macgregor and Elisabeth Waymire; assigned to Claris. Choosing and indicating graphically whether sorting is by row or column.
218. System and methods for improved scenario management in an electronic spreadsheet; US5303146; 1994-04-12. By Joseph Ammirato and Gavin Peacock; assigned to Borland. A form of version control permitting maintenance of several scenarios on the same spreadsheet. Closely related to patent 210.
219. Data processing apparatus and method for a reformattable multidimensional spreadsheet; US5317686; 1994-05-31. By R. Pito Salas and others; assigned to Lotus Development Corporation. Naming and display of cells.
220. * Method of bidirectional recalculation; US5339410; 1994-08-16. By Naoki Kanai; assigned to IBM. Proposes to replace the standard unidirectional computation by bidirectional constraints. This seems to require formulas to be inverted, which isn't possible in general.
221. Spreadsheet program which implements alternative range references; US5371675; 1994-12-06. By Irene Greif, Richard Landsman and Robert Balaban; assigned to Lotus Development Corporation. Use a menu to choose between different source cell ranges in a calculation.
222. System and method for storing and retrieving information from a multidimensional array; US5319777; 1994-06-07. By Manuel Perez; assigned to Sinper. Networked multidimensional spreadsheet program allowing concurrent updates.
223. * Method for optimal recalculation; US5276607; 1994-01-04. By Bret Harris and Lewis Bastian; assigned to WordPerfect Corporation. See section 3.3.7.
224. Method for hiding and showing spreadsheet cells; US5255356; 1993-10-19. By Eric Michelman and Devin Ben-Hur; assigned to Microsoft. Hiding or showing cells that contribute to subtotals and schematic, according to the cells' formulas.

225. Computer-aided decision making with a symbolic spreadsheet; US5182793; 1993-01-26. By Rhonda Alexander, Michael Irrgang and John Kirchner; assigned to Texas Instruments. Using a spreadsheet program to make decisions.
226. Spreadsheet cell having multiple data fields; US5247611; 1993-09-21. By Ronald Norden-Paul and John Brimm; assigned to Emtek Health Care Systems. Display, or not, spreadsheet cells holding mandatory as well as optional information.
227. Graph-based programming system and associated method; US5255363; 1993-10-19. By Mark Seyler; assigned to Mentor Graphics. Generalization of formulas to actions and event listeners, and of cell contents to graphical components.
228. Method and system for processing formatting information in a spreadsheet; US5231577; 1993-07-27. By Michael Koss; assigned to Microsoft. Cell formatting information and how to share it among cells.
229. Method for controlling the order of editing cells in a spreadsheet by evaluating entered next cell attribute of current cell; US5121499; 1992-06-09. By Rex McCaskill and Beverly Machart; assigned to IBM. Let each cell determine which cell is "next" in editing order.
230. Graphic file directory and spreadsheet; US5093907; 1992-03-03. By Yao Hwong and Mitsuro Kaneko; assigned to Axa. Display and process (miniatures of) image files in a matrix of cells.
231. Method for assisting the operator of an interactive data processing system to enter data directly into a selected cell of a spreadsheet; US5021973; 1991-06-04. By Irene Hernandez and Beverly Machart; assigned to IBM. Type the desired contents of a cell into the cell – presumably unlike early DOS-based spreadsheets, in which the text was typed in a separate editor line above the sheet.
232. System for generating worksheet files for electronic spreadsheets; US5033009; 1991-07-16. By Steven Dubnoff. Generate new sheets by inserting variable data into a pattern sheet, in the style of word processor merge files.
233. Intermediate spreadsheet structure; US5055998; 1991-10-08. By Terrence Wright, Scott Mayo and Ray Lischner; assigned to Wang Laboratories. Describes an interchange format for multidimensional spreadsheets.

Bibliography

[1] Eero Hyvönen and Stefano de Pascale. Interval computations on the spreadsheet. In R. B. Kearfott and V. Kreinovich, editors, *Applications of Interval Computations, Applied Optimization*, pages 169–209. Kluwer, 1996.

[2] Robin Abraham and Martin Erwig. Header and unit inference for spreadsheets through spatial analyses. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'04)*, pages 165–172, 2004. At http://web.engr.oregonstate.edu/~erwig/papers/HeaderInf_VLHCC04.pdf.

[3] Robin Abraham and Martin Erwig. Inferring templates from spreadsheets. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 182–191. ACM Press, 2006.

[4] Robin Abraham and Martin Erwig. Type inference for spreadsheets. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 73–84. ACM Press, 2006.

[5] Yanif Ahmad, Tudor Antoniu, Sharon Goldwater, and Shriram Krishnamurthi. A type system for statically detecting spreadsheet errors. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 174–183, 2003.

[6] Tudor Antoniu et al. Validating the unit correctness of spreadsheet programs. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 439–448. IEEE Computer Society, 2004.

[7] Yirsaw Ayalew. *Spreadsheet Testing Using Interval Analysis*. PhD thesis, Institut für Informatik-Systeme, Universität Klagenfurt, 2001. At https://143.205.180.128/Publications/pubfiles/psfiles/2001-0125-YA.ps on 22 August 2006.

[8] S.C. Bloch *Excel for Engineers and Scientists*. Wiley, second edition, 2003.

[9] Borland. Antique software: Turbo Pascal v1.0. Webpage. At http://www.danbricklin.com/visicalc.htm.

[10] Dan Bricklin. Visicalc information. Webpage. At http://www.danbricklin.com/visicalc.htm.

[11] Chris B. Browne. Linux spreadsheets. Webpage. At http://linuxfinances.info/info/spreadsheets.html.

[12] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[13] Margaret Burnett et al. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2):155–206, March 2001.

[14] Margaret Burnett, Andrei Sheretov, Bing Ren, and Gregg Rothermel. Testing homogeneous spreadsheet grids with the "what you see is what you test" methodology. *IEEE Transactions on Software Engineering*, 28(6):576–594, 2002.

[15] Rommert J. Casimir Real programmers don't use spreadsheets. *ACM SIGPLAN Notices*, 27(6):10–16, June 1992.

[16] Chris Clack and Lee Braine. Object-oriented functional spreadsheets. In *Proceedings of the 10th Glasgow Workshop on Functional Programming (GlaFP'97)*, September 1997. At http://citeseer.ist.psu.edu/clack97objectoriented.html.

[17] Michael Coblenz. Using objects of measurements to detect spreadsheet errors. Technical Report CMU-CS-05-150, School of Computer Science, Carnegie Mellon University, July 2005. At http://reports-archive.adm.cs.cmu.edu/anon/2005/CMU-CS-05-150.pdf.

[18] Colt. Homepage. Webpage. At http://dsd.lbl.gov/~hoschek/colt/.

[19] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

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

[21] Tony Davie and Kevin Hammond. Functional hypersheets. In *Eighth international Workshop on Implementation of Functional Languages*, pages 39–48, 1996. At http://www-fp.dcs.st-and.ac.uk/~kh/papers/Hypersheets/Hypersheets.html 31 August 2006.

[22] Walter de Hoon. Designing a spreadsheet in a pure functional graph rewriting language. Master's thesis, University of Nijmegen, 1993.

[23] Walter de Hoon, Luc Rutten, and Marko van Eekelen. Implementing a functional spreadsheet in Clean. *Journal of Functional Programming*, 5(3):383–414, 1995.

[24] Stefano de Pascale and Eero Hyvönen. An extended interval arithmetic library for Microsoft Excel. Research report, VTT Information Technology, Espöö, Finland, 1994.

[25] Decision Models. Excel pages – calculation secrets. Website. At http://www.decisionmodels.com/calcsecrets.htm.

[26] Decision Models. Homepage. Website. At http://www.decisionmodels.com/.

[27] Weichang Du and William W. Wadge The eductive implementation of a three-dimensional spreadsheet. *Software Practice and Experience*, 20(11):1097–1114, 1990.

[28] Ecma International. Homepage. At http://www.ecma-international.org/.

[29] John English. *Ada 95: The Craft of Object-Oriented Programming*. Prentice-Hall, 1997. At http://www.it.bton.ac.uk/staff/je/adacraft/.

[30] Martin Erwig and Margaret M. Burnett. Adding apples and oranges. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages. Lecture Notes in Computer Science, vol. 2257*, pages 173–191, London, UK, 2002. Springer-Verlag.

[31] Martin Erwig et al. Gencel: A program generator for correct spreadsheets. *Journal of Functional Programming*, 16(3):293–325, 2006.

[32] European Spreadsheet Risks Interest Group. Homepage. Webpage. At http://www.eusprig.org/.

[33] Marc Fisher et al. Integrating automated test generation into the wysiwyt spreadsheet testing methodology. *ACM Transactions on Software Engineering Methodology*, 15(2):150–194, 2006.

[34] OASIS Foundation. Open document format for office applications (OpenDocument) TC. Webpage. At http://www.oasis-open.org/committees/office/ on 25 August 2006.

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

[36] Joe Francoeur. Personal communication, August 2006.

[37] Gnumeric. Homepage. Webpage. At http://www.gnome.org/projects/gnumeric/.

[38] Vincent Granet. XXL spreadsheet. Webpage. At http://www.esinsa.unice.fr/xxl.html (download site not accessible August 2006).

[39] Steve Grubb. Ploticus. Webpage. At http://ploticus.sourceforge.net/ on 6 September 2006.

[40] Phong Ha and Quan Vi Tran. Brugerdefinerede funktioner i Excel. (User-defined functions in Excel). Master's thesis, IT University of Copenhagen, June 2006. In Danish.

[41] Haskell. Homepage. Webpage. At http://www.haskell.org/.

[42] Eero Hyvönen and Stefano de Pascale. A new basis for spreadsheet computing. Interval Solver(TM) for Microsoft Excel. In *11th Conference on Innovative Applications of Artificial Intelligence (IAAI-99)*, pages 799–806. AAAI Press, 1999. At http://www.mcs.vuw.ac.nz/~elvis/db/references/NBSSC.pdf.

[43] Tomás Isakowitz, Shimon Schocken, and Henry C. Lucas Toward a logical/physical theory of spreadsheet modeling. *ACM Transactions on Information Systems*, 13(1):1–37, 1995.

[44] Thomas S. Iversen Personal communication, September 2006.

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

[46] Neil D. Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993. At http://www.dina.kvl.dk/~sestoft/pebook/pebook.html.

[47] U. Jørring and W.L. Scherlis. Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 86–96, 1986.

[48] Brian Kahin. The software patent crisis. *Technology Review*, April 1990. At http://antipatents.8m.com/software-patents.html.

[49] R. Kelsey, W. Clinger, and J. Rees (editors). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[50] Loreen La Penna. Recalculation in Microsoft Excel 2002. Web page, October 2001. At http://msdn.microsoft.com/library/en-us/dnexcl2k2/html/odc_xlrecalc.asp.

[51] A. Lew and R. Halverson A FCCM for dataflow (spreadsheet) programs. In *FCCM '95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, pages 2–10. IEEE Computer Society, 1995.

[52] Serge Lidin. *Inside Microsoft .Net IL Assembler*. Microsoft Press, 2002.

[53] Björn Lisper and Johan Malmström. Haxcel: A spreadsheet interface to haskell. In *14th International Workshop on the Implementation of Functional Languages*, pages 206–222, 2002. At http://www.mrtc.mdh.se/publications/0435.pdf on 31 August 2006.

[54] Chuck Martin. sc. Webpage. At http://freshmeat.net/projects/sc/.

[55] B. D. McCullough Fixing statistical errors in spreadsheet software: The cases of Gnumeric and Excel. CSDA Statistical Software Newsletter, 2003. At http://www.csdassn.org/software_reports.cfm.

[56] Michael Meeks and Jody Goldberg. A discussion of the new dependency code, version 0.3. Code documentation, October 2003. File doc/developer/Dependencies.txt in Gnumeric source distribution, at http://www.gnome.org/projects/gnumeric/.

[57] Microsoft. Office online. Webpage. At http://office.microsoft.com/.

[58] R. Milner, M. Tofte, R. Harper, and D.B. MacQueen *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[59] Roland Mittermeir and Markus Clermont. Finding high-level structures in spreadsheet programs. In Arie van Deursen and Elizabeth Burd, editors, *Proceedings of the 9th Working Conference in Reverse Engineering, Richmond, VA, USA*, pages 221–232. IEEE Computer Society, 2002. At https://143.205.180.128/Publications/pubfiles/pdffiles/2002-0190-RMAM.pdf on 24 August 2006.

[60] Markus Montigel. Portability and reuse of components for spreadsheet languages. In *IEEE Symposia on Human Centric Computing Languages and Environments*, pages 77–79, 2002.

[61] Hanspeter Mössenböck, Albrecht Wöß, and Markus Löberbauer. The compiler generator Coco/R. Webpage. At http://www.ssw.uni-linz.ac.at/Coco/.

[62] Netlib. Homepage. Webpage. At http://www.netlib.org/.

[63] Microsoft Developer Network. Excel primary interop assembly reference. Class ApplicationClass. Webpage. At http://msdn2.microsoft.com/en-us/library/microsoft.office.interop.excel.applicationclass.aspx.

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

[65] National Institute of Standards. Javanumerics. Webpage. At http://math.nist.gov/javanumerics/.

[66] European Patent Office. Espacenet. Webpage. At http://ep.espacenet.com/.

[67] OpenOffice. Calc – the all-purpose spreadsheet. Webpage. At http://www.openoffice.org/product/calc.html.

[68] Ray Panko. Spreadsheet research. Website. At http://panko.cba.hawaii.edu/ssr/.

[69] Einar Pehrson. Cleansheets. Webpage. At http://freshmeat.net/projects/csheets/.

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

[71] Kurt W. Piersol Object-oriented spreadsheets: the analytic spreadsheet package. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA'86), Portland, Oregon*, pages 385–390. ACM Press, 1986.

[72] ReportingEngines. Formula One for Java. Webpage. At http://www.reportingengines.com/ on 19 September 2006.

[73] Lutz Roeder. Mapack for .NET. Webpage. At http://www.aisto.com/roeder/dotnet/.

[74] Boaz Ronen, Michael A. Palley, and Henry C. Lucas Spreadsheet analysis and design. *Communications of the ACM*, 32(1):84–93, 1989.

[75] Gregg Rothermel et al. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering Methodology*, 10(1):110–147, 2001.

[76] Gregg Rothermel, Lixin Li, and Margaret Burnett. Testing strategies for form-based visual programs. In *Eighth International Symposium on Software Reliability Engineering*, pages 96–107. IEEE Computer Society, 1997.

[77] Gregg Rothermel, Lixin Li, C. DuPuis, and Margaret Burnett. What you see is what you test: a methodology for testing form-based visual programs. In *20th International Conference on Software Engineering*, pages 198–207. IEEE Computer Society, 1998.

[78] Christoph Rüegg. Math.net project.

[79] Russell Schulz. comp.apps.spreadsheet FAQ. Newsgroup, June 2002. At http://www.faqs.org/faqs/spreadsheets/faq/.

[80] Peter Sestoft. Runtime code generation in JVM and CLR. Technical report, Royal Veterinary and Agricultural University, October 2002. At http://www.dina.kvl.dk/~sestoft/rtcg/.

[81] Bradford L. Smith Abykus. an object-oriented spreadsheet for windows. Website. At http://www.abykus.com/ on 7 September 2006.

[82] EUSES: End Users Shaping Effective Software. Wysiwyt: What you see is what you test. Webpage. At http://eusesconsortium.org/wysiwyt.php.

[83] Spec#. Homepage. At http://research.microsoft.com/specsharp/.

[84] SpreadsheetGear LLC. SpreadsheetGear for .NET. Webpage. At http://www.spreadsheetgear.com/ on 19 September 2006.

[85] Marc Stadelmann. A spreadsheet based on constraints. In *UIST '93: Proceedings of the 6th annual ACM symposium on User Interface Software and Technology*, pages 217–224. ACM Press, 1993.

[86] United States Court of Appeals for the Federal Circuit. Refac versus Lotus. Opinion 95-1350, April 1996. At http://www.ll.georgetown.edu/Federal/judicial/fed/opinions/95opinions/95-1350.html.

[87] United States Patent and Trademark Office. Patent full-text and full-page image databases. Webpage. At http://www.uspto.gov/patft/.

[88] Usenet. comp.apps.spreadsheet. Newsgroup.

[89] Noah Vawter. DFT multiply demo spreadsheet. Webpage, 2002. At http://www.gweep.net/~shifty/portfolio/fftmulspreadsheet/ on 29 August 2006.

[90] Andrew P. Wack *Partitioning dependency graphs for concurrent execution: a parallel spreadsheet on a realistically modelled message passing environment*. PhD thesis, Department of Computer and Information Sciences, University of Delaware, 1995.

[91] Guijun Wang and Allen Ambler. Solving display-based problems. In *IEEE Symposium on Visual Languages, Boulder, Colorado*, pages 122–129. IEEE Computer Society, 1996.

[92] Wikipedia. OpenDocument. Webpage. At http://en.wikipedia.org/wiki/OpenDocument on 25 August 2006.

[93] Wikipedia. Spreadsheet. Webpage. At http://en.wikipedia.org/wiki/Spreadsheet.

[94] Wikipedia. Visicalc. Webpage. At http://en.wikipedia.org/wiki/VisiCalc.

[95] Stephen Wolfram. *The Mathematica Book*. Cambridge University Press, 1999.

[96] Alan G. Yoder and David L. Cohn Architectural issues in spreadsheet languages. In *1994 Conference on Programming Languages and System Architectures. Lecture Notes in Computer Science, vol. 782*. Springer-Verlag, 1994. Also at http://www.cse.nd.edu/research/tech_reports/1993.html.

[97] Alan G. Yoder and David L. Cohn Observations on spreadsheet languages, intension and dataflow. Technical Report TR-94-22, Computer Science and Engineering, University of Notre Dame, 1994. At ftp://www.cse.nd.edu/pub/Reports/1994/tr-94-22.ps.

[98] Alan G. Yoder and David L. Cohn Real spreadsheets for real programmers. In *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*, pages 20–30, 1994. Also at ftp://www.cse.nd.edu/pub/Reports/1994/tr-94-9.ps.

[99] Alan G. Yoder and David L. Cohn Domain-specific and general-purpose aspects of spreadsheet languages. In Sam Kamin, editor, *DSL '97 - First ACM SIGPLAN Workshop on Domain-Specific Languages, Paris, France*, University of Illinois Computer Science Report, pages 37–47, 1997. At http://www-sal.cs.uiuc.edu/~kamin/dsl.

Index