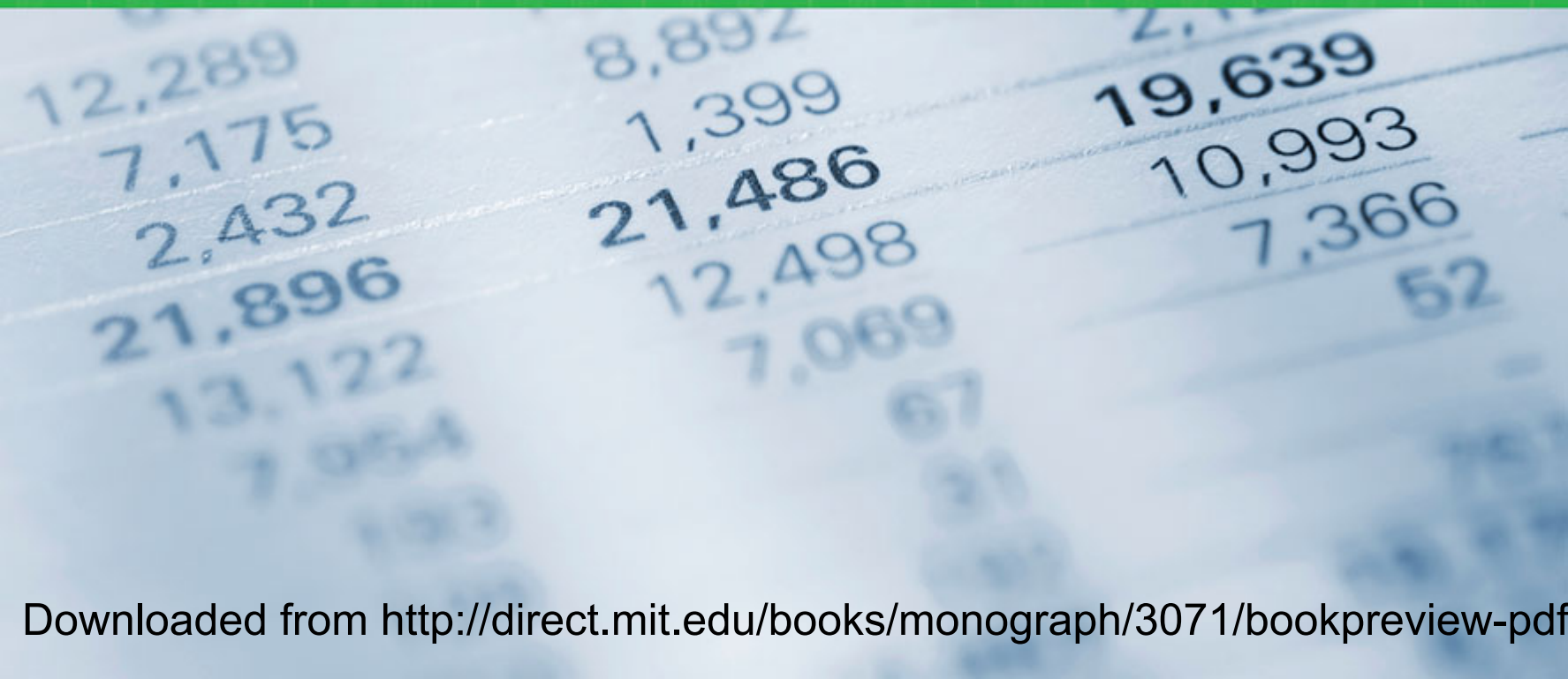
Spreadsheet Implementation Technology

Basics and Extensions

Peter Sestoft



Spreadsheet Implementation Technology

Peter Sestoft

Spreadsheet Implementation Technology

Basics and Extensions

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in New Century Schoolbook by the author using LaTeX.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Sestoft, Peter.
Spreadsheet implementation technology : basics and extensions / Peter Sestoft.
p. cm.
Includes bibliographic references and index.
ISBN 978-0-262-52664-7 (pbk. : alk. paper)
1. Electronic spreadsheets—Computer programs. 2. Functional programming (Computer science) 3. Object-oriented methods. I. Title.
HF5548.2.S43724 2014
005.54—dc23
2014006940

10 9 8 7 6 5 4 3 2 1

Contents

Preface

Spreadsheet programs are used daily by millions of people for tasks ranging from neatly organizing a list of addresses, to analysis of biological data sets, to complex economical simulations. Spreadsheet programs are easy to learn and convenient to use because they have a clear visual data model (tables) and a simple, efficient underlying computation model (functional and side effect free). It has been estimated that by 2012 there would be at least 13 million "end-user programmers" in the United States, chiefly using spreadsheets to build non-trivial computational models [135, section 4.3].

Spreadsheet programs are usually not held in high regard by professional software developers [30]. However, their implementation involves a large number of non-trivial design considerations and space-time 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 be helpful to a very 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 Calc are rather complex, they are usually written in unmanaged languages such as C and C++, and the documentation of their internals is sparse. Spreadsheet implementations written for pedagogical reasons abound but often fail to scale to realistic problem sizes. 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 book The purpose of this book is to enable others to make experiments with innovative spreadsheet functionality and with new ways to implement such functionality. 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. Most patents do not describe design alternatives, and many also neglect to describe relevant academic work on which they are based.

This book is an attempt to provide a more coherent picture by gleaning information from experience with existing spreadsheet implementations, from 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.

Moreover, we present Funcalc, which extends Corecalc with an implementation of user-defined functions, expressed solely using standard spreadsheet concepts such as cells, formulas, and references, requiring no external languages such as VBA, Python, or Java. This work is inspired by the ideas of Peyton-Jones and others [120], but here we emphasize performance because libraries of user-defined functions will not replace built-in functions in practice unless they are equally fast.

Technologically, Funcalc uses run-time code generation to obtain good performance while preserving full spreadsheet interactivity. This approach exploits the highly optimizing just-in-time compilers of modern managed software platforms while minimizing the amount of engineering and software maintenance required on our part.

Contents The book comprises the following parts:

- Chapter 1 summarizes the spreadsheet computation model and the most important challenges for efficient recalculation. This includes a survey of scholarly works, spreadsheet implementations, and some patents.
- Chapter 2 describes Corecalc, a core implementation of essential spreadsheet functionality for making practical experiments, chapter 3 discusses alternatives to some of the design decisions made in Corecalc, and chapters 4 and 5 investigate data structures to support minimal recalculation.
- Chapter 6 introduces and motivates Funcalc, an extension of Corecalc with compiled sheet-defined functions, which permits users to define their own functions without extraneous programming languages and without any loss of efficiency compared to built-in functions. Chapters 7 through 9 describe the Funcalc implementation and possible design variations and extensions.
- Chapter 10 shows how partial evaluation, or automatic function specialization, fits particularly well with the interactive and side effect-free spreadsheet programming model.
- Chapter 11 contains a user manual for the Funcalc implementation.

The implementations of Corecalc and Funcalc are available in source form under a liberal license and are written in C# using only managed code.

Goals of the Corecalc implementation The purpose of the Corecalc implementation described in chapters 2 through 4 of this book is to provide a source code platform for experiments with spreadsheet implementation, presenting also the underlying design ideas and considerations. 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.

Goals of the Funcalc implementation The purpose of the Funcalc implementation described in chapters 6 through 9 is to demonstrate that sheet-defined functions can be both convenient and fast, while retaining the highly interactive spreadsheet work mode, and hence empower spreadsheet end users. The Funcalc implementation is an extension of Corecalc.

Pedagogical side benefits A realistic spreadsheet implementation draws on and illustrates a range of software technologies and computer science subjects. In this book we will encounter "asides"—sections that briefly introduce the following subjects:

- The formal description of the semantics, or meaning, of formula evaluation and spreadsheet recalculation (section 1.8).
- The representation of formulas as abstract syntax, and the conversion of concrete syntax (text) into abstract syntax, using a simple lexer and parser generator (section 2.2.1).
- Interpretation, or evaluation, of formula abstract syntax (section 2.7).
- Concepts of object-oriented programming (section 2.1.2) and functional programming (sections 2.3 and 4.2.3).
- The representation of binary floating-point numbers according to the IEEE-754 standard [81] (section 2.8.1).
- A space- and time-efficient data structure for representing large sparse arrays (section 2.16), and the features of current computer hardware that contribute to the efficiency of this data structure.
- The Common Intermediate Language (CIL) bytecode used in the Microsoft .NET platform (sections 7.5 and 7.7).
- Compiler technology, how to get from formula abstract syntax to a list of bytecode instructions (chapters 7 and 8).
- The notion of tail call for executing certain recursive function calls in constant space (section 8.3.2).
- The concept of program specialization or partial evaluation [85], as applied to sheet-defined functions (section 10.1).

When necessary, we will briefly introduce such prerequisites and also provide references to further reading on these subjects.

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/funcalc/`

The Corecalc implementation is copyrighted by the authors and distributed under an MIT-style license:

This means that you can use and modify the Corecalc and Funcalc 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 in case the commercial exploitation of an idea presented in this book is construed to violate one or more patents.

Also, all trademarks belong to their owners.

Acknowledgments This text began to take shape, and much new work on Funcalc was done, during a visit to Greg Morrisett's group at Harvard University in March–July 2009 in a splendid corner office across from the Museum of Natural History. Some of the chapters describing Corecalc are based on a previous technical report [137], but we have revised them considerably to reflect the development of Funcalc.

The original impetus to look at spreadsheet technology 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 [120].

Thomas S. Iversen investigated the use of run-time code generation for speeding up spreadsheet calculations in his 2006 MSc thesis project [84], jointly supervised with Torben Mogensen (DIKU, University of Copenhagen). Parts of this work are summarized in [137, 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 sheet-defined functions, thus allowing spreadsheet users to define their own functions using well-known spreadsheet concepts. This work was done in their 2006 MSc thesis project [37].

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 [72].

Morten Poulsen and Poul Serek implemented and experimented with the extended support graph construction in sections 5.1 through 5.4 [122]. Subsequently, they built the first compiler implementation of sheet-defined functions, based on my early versions of the design laid out in chapters 6 to 8.

Several groups of students have investigated distributed collaborative spreadsheets based on the Corecalc platform, in particular Vincens Riber Mink and Daniel Schiermer [104]. Nader Salas furthermore considered full traceability [134].

Other IT University students, including Jacob Atzen, Claus Skoubølling Jørgensen, Jens Lind, Poul Brønnum, Jens Hamann, Hui Xu, Mainul Liton, Linas Patapavicius, Rasmus Nielsen, Jens Zeilund Sørensen, Hildur Uffe Flemberg, Martin Jeanty Larsen, Jonas Druedahl Rask, and Simon Eikeland Timmermann, investigated other parts of the spreadsheet design space and provided valuable insights, comments, and corrections to this book.

Michael Reichhardt Hansen and the publisher's anonymous reviewers provided many valuable comments and suggestions, which led to significant improvements of the presentation.

It was a great pleasure to work with Ada Brunstein, Marie Lufkin Lee, Marc Lowenthal, and Virginia Crossman at MIT Press during the book's somewhat protracted gestation.

Naming Conventions

Name	Meaning	Type	Page
`act`	`void` delegate	Action⟨T⟩	
`ae`	adjusted expression	Adjusted⟨Expr⟩	68
`arr`	array value	ArrayValue	48
`c`	column index	`int`	
`ca`	cell address, absolute	CellAddr	51
`ccar`	cell or cell area reference	CellRef, CellArea	114
`cell`	cell	Cell	39
`col`	column number, zero-based	`int`	
`cols`	column count	`int`	
`deltaCol`	column increment	`int`	
`deltaRow`	row increment	`int`	
`dlg`	non-`void` delegate	Func⟨T⟩	54
`e`	expression in formula	Expr	40
`es`	expression array	Expr[]	
`fca`	full cell address, absolute	FullCellAddr	
`fv`	function value, closure	FunctionValue	201
`lr`	lower right corner of area	RARef	49
`r`	row index	`int`	
`raref`	relative/absolute reference	RARef	49
`row`	row number, zero-based	`int`	
`rows`	row count	`int`	
`sheet`	sheet	Sheet	38
`ul`	upper left corner of area	RARef	49
`v`	value	Value	44
`vs`	value array	Value[]	
`workbook`	workbook	Workbook	36

Chapter 1

What Is a Spreadsheet

To answer the question "What is a spreadsheet?", one may point to Microsoft Excel, OpenOffice Calc, or Gnumeric. However, that would provide poor guidance for designing variations and novel extensions to the spreadsheet concept, as in part II of this book. An understanding of spreadsheet concepts should leave open the possibility of designing such variations and extensions, yet those variations and extensions should behave "as expected" by experienced spreadsheet users—an idea that is known as the principle of least astonishment.

1.1 History

The first spreadsheet program in the modern sense was VisiCalc, developed by Dan Bricklin and Bob Frankston in 1979 for the Apple II computer [21, 63, 164] and [22, chapter 12]. 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 whose first version was 85 KB [132], PlanPerfect, QuattroPro, and many more. By now the dominating spreadsheet program is Microsoft Excel [102], whose executable (version 2013) weighs in at 19,917 KB. Several open source spreadsheet programs exist, including Gnumeric [66] and OpenOffice Calc [115]. Moreover, there are multiple online collaborative spreadsheet programs running in browsers, such as Google Docs (part of Google Drive) [68]. See also the encyclopedia paper [1], the introduction to Felienne Hermans' PhD thesis [76], or Wikipedia's entry on spreadsheets [163].

1.2 Basic concepts

All spreadsheet programs have the same visual model: a two-dimensional grid of cells. Columns are labeled with letters A, B, . . . , Z, AA, . . . , rows are labeled 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 a user edits the contents of a cell, all cells that directly or transitively depend on that cell will be recalculated automatically.

Figure 1.1 shows an example spreadsheet, summarizing the grades given in an exam. Column A lists the possible grades $-3, 00, 02, 4, 7, 10, 12$; they correspond to the F, FX, E, D, C, B, and A grades on the European ECTS scale. Column B shows the number of times each grade was awarded in this particular exam, and column C computes the product of the grade and number of times awarded. Cell B9 computes the total number of grades awarded (the sum of column B), cell C9 computes the sum of the products (the sum of column C), and cell C11 computes the average grade (the ratio C9/B9). Column D computes what percentage each grade constitutes of the total number of grades. Figure 1.2 shows the formulas used in these computations.

	A	B	C	D
1	Grade	Count	Product	Count %
2	-3	1	-3	1.7
3	0	6	0	10.3
4	2	5	10	8.6
5	4	9	36	15.5
6	7	19	133	32.8
7	10	14	140	24.1
8	12	4	48	6.9
9	Sum	58	364	
10				
11		Average	6.3	
12				

Figure 1.1: Spreadsheet window summarizing grades given in an exam. Column A hold the grades, B the number of times each grade was given, C computes their product, and D computes the percentage distribution of the grades. Cell C11 computes the average grade.

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. For instance, a row-absolute and column-relative reference such as `B$5` will keep referring to the same row but will have its column adjusted when copied. 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. Interestingly, the original VisiCalc program did not distinguish between relative and absolute references

in formulas; instead one had to indicate which references to adjust (relative) and which not to adjust (absolute) at each formula copy operation.

Figure 1.2 shows the formulas behind the sheet from figure 1.1. The formulas in C3:D8 are copies of that in C2, with the row numbers automatically adjusted from 2 to 3, 4, and so on. The formula in C9 is a copy of that in B9, with the column automatically adjusted from B to C in the cell area reference. Finally, the formulas in D3:D8 are copies of the formula `=B2/$B$9*100` in D2; note how the relative row number 2 in `B2` gets adjusted whereas the absolute row number 9 in `$B$9` does not.

<>	A	B	C	D
1	Grade	Count	Product	Count %
2	-3	1	=A2*B2	=B2/$B$9*100
3	0	6	=A3*B3	=B3/$B$9*100
4	2	5	=A4*B4	=B4/$B$9*100
5	4	9	=A5*B5	=B5/$B$9*100
6	7	19	=A6*B6	=B6/$B$9*100
7	10	14	=A7*B7	=B7/$B$9*100
8	12	4	=A8*B8	=B8/$B$9*100
9	Sum	=SUM(B2:B8)	=SUM(C2:C8)	
10				
11		Average	=C9/B9	

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 [83]. It is notable how messy the graph (with explicit dependencies between cells) looks compared with the rectangular layout with its implicit cell dependencies.

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. The A1 cell reference format originates in VisiCalc [21].

Microsoft's Multiplan spreadsheet program (1982) used a different 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 was used also in Piersol's 1986 spreadsheet implementation [121].

The R1C1 format is interesting because it is used in Excel's XML Spreadsheet 2003 (XMLSS) export format, and because Excel and Gnumeric (but apparently not OpenOffice Calc) can optionally display formulas in R1C1 format. Also, it is close to the internal format of our implementation Corecalc.

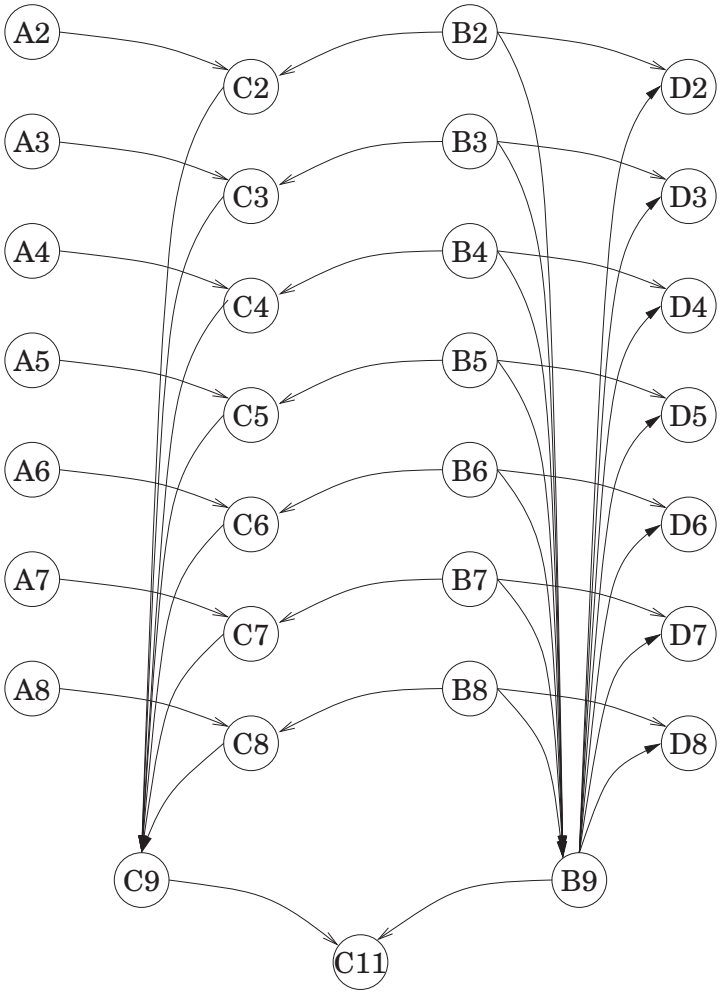


Figure 1.3: A graph-oriented view of the spreadsheet in figures 1.1 and 1.2.

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	Absolute; 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 arrays

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 numbers such as RAND; functions to manipulate times and dates such as NOW and TODAY; financial functions such as "present value"; a conditional function IF; array 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.

Some built-in functions, called array functions, return an entire array (or 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.

	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.

The array result must then be distributed 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 Calc, this is achieved by entering the formula as a so-called *array formula*. First one marks the *display area*, that is, the cell area that should receive the values, then one enters the formula in the formula bar, and finally one types Ctrl+Shift+Enter instead of just Enter to complete the formula entry. This holds for Excel on Windows; for MacOS versions of Excel, use Cmd+Enter. The resulting formula is shown in curly braces, like `{=TRANSPOSE(A1:B3)}`, in every cell of the display area, although each cell contains only one component of the result. See figure 1.6 for an example.

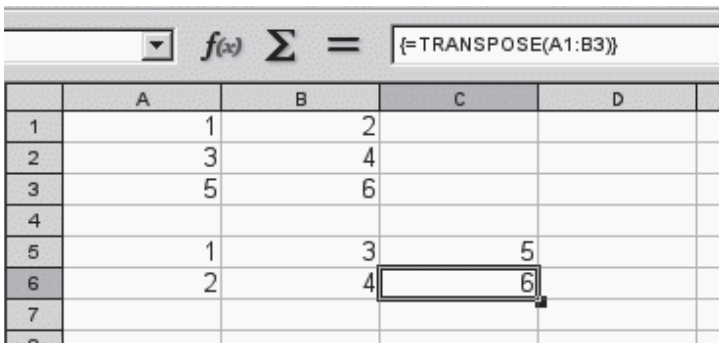


Figure 1.6: The array formula {`=TRANSPOSE(A1:B3)`} in result area A5:C6.

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 sheet in the same workbook using the notation `Sheet2!A$1` in Excel and Gnumeric and `Sheet2.A$1` in OpenOffice Calc. Similarly, cell area references can be qualified with the sheet name, 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 part I of this book supports all the functionality described above, including built-in functions and array 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 dependency of one cell on the value of another. When cell C2 contains the formula `=A2*B2`, as in figure 1.2, we say that C2 *directly depends on* cells A2 and B2, and cells A2 and B2 *directly support* cell C2. Some spreadsheet programs, notably Excel and OpenOffice Calc, can display the dependencies using a feature called *formula audit*. For instance, figure 1.7 shows which cells directly or indirectly depend on B2. This visualization simply combines part of the graphical view in figure 1.3 with the usual spreadsheet grid view.

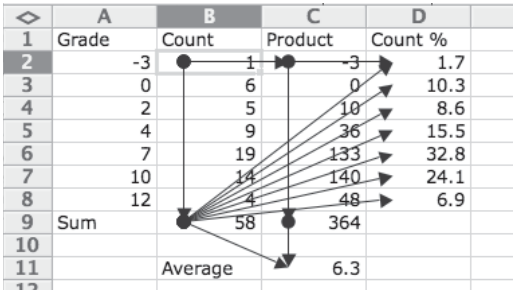


Figure 1.7: The direct and indirect dependencies on cell B2 in the sheet from figures 1.1 and 1.2.

A cell may directly depend on any number of other cells. For instance, cell C11 in figure 1.7 directly depends on B9 and C9. Similarly, a cell may directly support any number of other cells: cell B9 directly supports C11 and the cells in range D2:D8.

More precisely, C11 both *statically* and *dynamically* depends on B9 and C9. By static dependency we mean that the formula text in C11 refers to cells B9 and C9, and by dynamic dependency we mean that calculating the value of C11 requires calculating the values of those cells.

A static dependency may or may not cause a dynamic dependency; it is an approximation of dynamic dependency. For instance, a cell containing the formula `=IF(G1<>0, G2, G3)` statically depends on G1, G2, and G3, but in any given recalculation dynamically depends only on G1 and G2, or on G1 and G3, depending on G1 being 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 C11 indirectly depends on B2, through B9 (and also through C2 and C9). The notion of *transitive support* is defined similarly. For instance, cell B2 transitively supports C2, D2, B9, C9, C11, and the cells in range D2:D8—the latter because B9 supports them.

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 4.4 have more to say about cycles.

1.7 Recalculation

When the contents of a cell are changed by editing it, all cells supported by that cell, whether in the same sheet or another sheet in the workbook, must be recalculated. The purpose is to implement level 3 liveness, that is, the automatic and consistent recalculation and redisplay of results whenever the user has edited some data [152]. Such edits happen relatively frequently, although hardly more than once a second when performed by a human.

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 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, 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 Corecalc implementation uses a mixture of top-down and bottom-up recalculation (see sections 2.12 and 4.3).

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 a user has edited a single cell, since this is the most frequent scenario:

- Recalculation should ensure *consistency* of cell values and formulas. After a recalculation, the value of a cell is consistent with the cell's formula and the

values of the cells to which the formula refers. In other words, if the formula contains no calls to volatile functions, then reevaluating (only) the formula would give the same value. Similarly, if the formula does contain calls to volatile functions, then there must be plausible values of those volatile functions that would result in the cell's value. This requirement is formalized in section 1.8.

In practice, Excel sometimes violates the consistency requirement (see discussion in example 6.26).

- Recalculation should be efficient in time and space. Preferably, it should avoid recalculating a cell formula that does not contain calls to volatile functions and whose precedent cells' values have not changed since the last recalculation. The consistency principle means that such evaluation (or not) is not observable, and hence is harmless, but it is reasonable to expect that a good recalculation mechanism takes time linear in the size of formulas in those cells supported by the cells that have changed. Also, it is preferable for supporting data structures to require space that is at most linear in the total size of formulas in the workbook (see section 1.7.3).

  For the same reasons of efficiency, recalculation should preferably avoid evaluating unused arguments of `IF(e1, e2, e3)` and other non-strict functions (see section 1.7.4). When we introduce sheet-defined functions in part II of the book, it becomes essential to avoid evaluating such unused arguments to ensure termination of recursively defined functions.

- Every recalculation should evaluate `RAND()` and other volatile functions (see section 1.7.5).

- The spreadsheet implementation can freely choose the recalculation order in two senses: the order in which cell values are calculated and updated is unspecified, and the order of subexpression evaluation is unspecified. For instance, the result of `NOW()-NOW()` may be negative, zero, or positive.

- Recalculation should accurately detect dynamic cycles (see section 1.7.6).

These requirements still leave considerable latitude for a recalculation mechanism: cells may be evaluated in parallel (provided they do not depend on each other), a cell may not be evaluated at all (provided no displayed cell depends on it, directly or indirectly), a cell may be evaluated multiple times, and so on, so long as the final result is consistent.

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 array 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

cells it depends on. This simple mechanism, described in section 2.12, 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. We show how in sections 3.3 and 4.3.

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 `e3`. For instance, the function call `IF(A2<>0, 1/A2, 1)` evaluates its second operand `1/A2` only if the value of `A2` is different from 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

Some functions are *volatile*. Although they take no arguments, different calls typically produce different values. The most common volatile functions are `NOW()`, which returns the current time, and `RAND()`, which returns a random number between 0 and 1. A formula whose result depends on a volatile function must be evaluated once in each recalculation. For instance, a cell containing `=IF(RAND()>0.5, 11, 22)` will be evaluated in every recalculation and may or may not produce a new value each time. However, in `IF(RAND()>0.5, NOW(), 10)`, if the condition happens to evaluate to false, then the call to `NOW()` may not be evaluated.

Volatile functions are easy to implement, but a spreadsheet implementation must keep track of which cells contain calls to volatile functions (see sections 3.3.1 and 3.3.2).

1.7.6 Dependency cycles

If a cell dynamically depends on itself in a recalculation, then this will be discovered and reported, regardless of whether there exists a value of the cell that would satisfy the consistency principle. For instance, if cell B1 contains the formula `=2/B1`, then even though recalculation could put B1 equal to $\sqrt{2}$ or $-\sqrt{2}$ and thereby achieve the required consistency, it should instead report a cyclic dependency. Thus, a spreadsheet implementation should not be expected to compute numerical solutions or fixed points through the use of cyclic dependencies. (This view is not universally held. It seems to be a common, although risky, practice in some financial models to define equation systems by cyclic dependencies and then recalculating the spreadsheet until the results have converged.)

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 dependency 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.

Both Excel and OpenOffice Calc take that approach and 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.

1.7.7 Spreadsheets are dynamically typed

Spreadsheet programs distinguish among several types of data, such as numbers, text strings, logical values (Booleans), and arrays. However, this distinction is made dynamically, in the style of Scheme [145], rather than statically, in the style of Haskell [74] or Standard ML [103].

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 an array-shaped cell area, but evaluates to an array of `ArgType` error values 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.

Because of dynamic typing, a spreadsheet implementation must in general wrap numbers and other values as objects so that it can distinguish them at run-time (see section 2.8).

1.7.8 Error values must be propagated

Because spreadsheet formulas, like languages such as Lisp, Javascript, and Ruby, are dynamically typed, the evaluation of an expression may fail due to giving the wrong number of arguments to a function, due to the wrong type of argument, and for many other reasons.

Two points are worth noting. First, such failures of evaluation should be tolerated because they are likely to arise during editing of a spreadsheet model. Therefore, a failure should not crash the spreadsheet program by throwing an exception, say. Second, there may be hundreds of such failed evaluations in a single recalculation (e.g., during major edits to a spreadsheet model), and such failures should not open hundreds of warning dialogs or similar.

Therefore, spreadsheet programs simply let a failed evaluation produce a distinguished kind of value—an error value. Further computations must propagate such an error value so that it can be easily traced back to its original cause. For example, applying the mathematical logarithm function to a string as in `LOG("zwei")` should produce an ArgType error value, and further computation must propagate this error. Hence, `10+LOG("zwei")` must produce ArgType error as well, and so must comparisons such as `10+LOG("zwei") < A1` and conditional expressions

such as `IF(10+LOG("zwei") < A1, 22, 33)`. Applying the logarithm to a negative number as in `LOG(-3)` must produce a NumError error value and so must any more complex expression that depends on this function call.

Thus, if a subexpression of a formula evaluates to an error value, then this error value will be propagated as the result of the formula. If multiple subexpressions evaluate to error values, then one of them will be propagated as the result of the formula. This is similar to exception propagation in an imperative language whose evaluation order is indeterminate. In particular, if *e1* evaluates to an error in `IF(e1, e2, e3)`, then the entire *IF*-expression evaluates to that error.

1.7.9 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 [74]. 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*, whereas (re)calculation in a spreadsheet typically is driven by *availability of input*. These evaluation strategies may be characterized as "backwards flow of demand" versus "forwards flow of data".

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 detected by the recalculation mechanism.

Some researchers have proposed spreadsheet programs that are lazy also in the above sense of evaluation being driven by demand for output; see Nuñez [113] and Du and Wadge [48], who call this *eductive evaluation*.

1.8 Aside: Formal semantics of spreadsheets

Here we give a simple formal semantics of spreadsheet recalculation, relating this semantics to the informal discussion in the preceding section. None of the immediately following chapters depends on this formalization so it may be skipped without much loss of context.

Rather than showing *how* recalculation should work by giving a somewhat algorithmic description in the form of a denotational or operational semantics [112], we specify *what* a recalculation should achieve, namely, consistency between the values calculated for each formula cell.

Such consistency can be specified by giving a semantics for the evaluation of individual cell formulas only, rather than for recalculation of the whole sheet or workbook. In this way, we avoid overspecifying the recalculation process, leaving it

unspecified in which order cells are updated, whether cells are recalculated sequentially or in parallel, whether the value of a cell needs to be calculated at all (in case no cell that it depends on has changed), or whether it may be calculated more than once (say, in a speculative parallel computation) during a single recalculation.

We believe that the semantics given in this section reflects actual spreadsheet implementation, but to avoid notational and conceptual clutter, we make some simplifications:

- The only types of values are Number and Error, not strings and arrays. Numbers suffice to represent also the logical values false (zero) and true (any nonzero number).
- We consider only a single sheet, not multi-sheet workbooks.
- Infix arithmetic operations such as `B2+7` and comparisons `B2<10` will be represented as prefix function calls such as `+(B2,7)` and `<(B2,10)`.
- We consider only cell references such as `B2`, not area references such as `B2:C4`. Also, note that the distinction between relative (`B2`) and absolute (`$B$2`) references does not matter when evaluating a formula, only when copying it.
- Functions propagate errors from all their arguments, except `IF(`e_1`,`e_2`,`e_3`)`, which propagates errors only from the condition e_1 and from the single branch (e_2 or e_3) that gets evaluated.
- The only volatile function is `RAND()`.
- A constant cell, such as `7`, will be represented by a constant formula `=7`.

The semantics presented here is easily extended to other types of values, multi-sheet workbooks, cell area references (ranges), other non-strict functions than `IF`, other volatile functions than `RAND()`, and a distinction between constant cells (that never need to be evaluated) and formula cells (that may need to be evaluated). However, that would just increase notational clutter without giving much new insight.

The simplified formulas used in this section are described in figure 1.8.

e	::=	`n`	number constant
	|	`ca`	cell reference
	|	`IF`(e_1,e_2,e_3)	conditional expression
	|	`RAND`$()$	volatile function
	|	`F`$(e_1,\ldots,e_n)$	function call

Figure 1.8: Syntax of the simplified formula language.

1.8.1 Semantic sets and functions

To describe the evaluation of formulas, we use the semantic sets and functions defined in figure 1.9. These are sometimes called semantic domains, but here they are ordinary sets and partial functions. For instance, $Value = Number + Error$ is the set of values, where a value v is either a proper number such as 0.42 in set $Number$ or an error such as `#DIV/0!` in set $Error$. The set $Addr$ contains cell addresses ca such as `B2`.

To describe the formulas of a worksheet, we use a map $\phi : Addr \to Expr$ so that when $ca \in Addr$ is a cell address, $\phi(ca)$ is the formula in cell ca. If cell ca is blank, then $\phi(ca)$ is undefined. The domain of ϕ is $dom(\phi) = \{\ ca \mid \phi(ca) \text{ is defined}\ \}$, the set of cell addresses that have a formula, that is, the set of non-blank cells. The ϕ function is not affected by recalculation, only by editing the sheet.

The result of a recalculation is modeled by function $\sigma : Addr \to Value$, where $\sigma(ca)$ is the computed value in cell ca. The σ function gets updated by each recalculation (see section 1.8.3).

$$
\begin{array}{lcll}
n & \in & Number & = \{ \text{ proper numbers } \} \\
 & & Error & = \{ \texttt{\#DIV/0!}, \texttt{\#CYCLE!} \} \\
ca & \in & Addr & = \{ \text{ cell addresses } \} \\
v & \in & Value & = Number + Error \\
e & \in & Expr & = \{ \text{ formulas, see figure 1.8 } \} \\
\phi & & & \in Addr \to Expr \\
\sigma & & & \in Addr \to Value
\end{array}
$$

Figure 1.9: Sets and maps used in the spreadsheet semantics: $Number$ is the set of proper floating-point numbers, excluding NaNs and infinities (section 2.8.1); $Error$ is the set of error values; $Addr$ the set of cell addresses; $Value$ the set of values (either number or error); and $Expr$ the set of formulas.

1.8.2 Semantics of formula evaluation

The semantics for formulas is given as a natural semantics [87], a variant of operational semantics [112], using inference rules that involve big-step evaluation judgments. An evaluation judgment has the form $\sigma \vdash e \Downarrow v$, which says: When σ describes the calculated values of all cells, then formula e may evaluate to value v. Note that v may be a number value or an error value.

To understand inference rules, consider this rule:

$$\frac{\sigma \vdash e_i \Downarrow v_i \in Error}{\sigma \vdash \mathtt{F}(e_1, \ldots, e_n) \Downarrow v_i}\ (e5e)$$

This inference rule consists of a premise above the line and a conclusion below the line. The conclusion concerns the value of a function call expression $\mathtt{F}(e_1, \ldots, e_n)$,

and the premise concerns the value of one of the call's argument expressions e_i. The rule can be read as follows: If there is some argument expression e_i that may evaluate to an error value v_i, then the function call may evaluate to the error value v_i also. That is, the rule describes the propagation of errors from argument to result in a function call. If multiple arguments e_i and e_j may evaluate to different error values v_i and v_j, then the rule does not specify which error will be propagated to the call's result.

For another example, consider this rule, also for a function call $\texttt{F}(e_1,\ldots,e_n)$ with n arguments:

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \notin \mathit{Error} \qquad \ldots \qquad \sigma \vdash e_n \Downarrow v_n \notin \mathit{Error}}{\sigma \vdash \texttt{F}(e_1,\ldots,e_n) \Downarrow f(v_1,\ldots,v_n)} \; (e5v)$$

This rule has n premises and can be read as follows: If all argument expressions $e_1,\ldots,e_n$ may evaluate to non-error values $v_1,\ldots,v_n$, then the value of the function call is obtained by applying the actual function f to these values, as in $f(v_1,\ldots,v_n)$.

The "may" is important because, in general, an expression may evaluate to multiple different values. For instance, `RAND()` may evaluate to any number between 0.0 (included) and 1.0 (excluded). Hence, `7+1/RAND()` may evaluate to some number greater than $7+1$ or to the error `#DIV/0!` in case `RAND()` produces 0.0.

The complete set of inference rules that describe when a formula evaluation judgment $\sigma \vdash e \Downarrow v$ holds are given in figure 1.10. Note that there are five groups of rules (e1), (e2x), (e3x), (4), (e5x), each corresponding to one of the five kinds of formulas in figure 1.8. Also, the formula fragments that appear in the premises are always smaller than the formula that appears in the conclusion. Hence, one can make a conclusion about a given formula through a finite number of rule applications.

The formula evaluation rules in figure 1.10 may be explained as follows:

- Rule (e1) says that a number constant `n` evaluates to that constant's value.
- Rule (e2b) says that a reference ca to a blank cell, that is, one for which $\sigma(ca)$ is not defined, gives value 0.0.
- Rule (e2v) says that a reference ca to a non-blank cell evaluates to the value $\sigma(ca)$ calculated for that cell. This value may be a number or an error.
- Rule (e3e) says that the expression $\texttt{IF}(e_1,e_2,e_3)$ may evaluate to error v_1 if the condition e_1 may evaluate to error v_1.
- Rule (e3f) says that $\texttt{IF}(e_1,e_2,e_3)$ may evaluate to value v provided the condition e_1 may evaluate to the non-error number zero and the "false branch" e_3 may evaluate to v.
- Rule (e3t) says that $\texttt{IF}(e_1,e_2,e_3)$ may evaluate to value v provided the condition e_1 may evaluate to some non-error non-zero number v_1 and the "true branch" e_2 may evaluate to v.

$$\frac{}{\sigma \vdash \mathtt{n} \Downarrow n}\ (e1)$$

$$\frac{ca \notin dom(\sigma)}{\sigma \vdash \mathtt{ca} \Downarrow 0.0}\ (e2b)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma \vdash \mathtt{ca} \Downarrow v}\ (e2v)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \in Error}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v_1}\ (e3e)$$

$$\frac{\sigma \vdash e_1 \Downarrow 0.0 \in Number \qquad \sigma \vdash e_3 \Downarrow v}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v}\ (e3f)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \in Number \qquad v_1 \neq 0.0 \qquad \sigma \vdash e_2 \Downarrow v}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v}\ (e3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma \vdash \mathtt{RAND}() \Downarrow v}\ (e4)$$

$$\frac{\sigma \vdash e_i \Downarrow v_i \in Error}{\sigma \vdash \mathtt{F}(e_1,\ldots,e_n) \Downarrow v_i}\ (e5e)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \notin Error \qquad \ldots \qquad \sigma \vdash e_n \Downarrow v_n \notin Error}{\sigma \vdash \mathtt{F}(e_1,\ldots,e_n) \Downarrow f(v_1,\ldots,v_n)}\ (e5v)$$

Figure 1.10: Evaluation rules for simplified spreadsheet formulas.

- Rule (e4) says that function call `RAND()` may evaluate to any (non-error) number v greater than or equal to zero and less than one. Hence, this rule models nondeterministic choice. It allows any formula that involves a call to `RAND()` to be evaluated in a recalculation. However, it does not *require* such reevaluation as in section 1.7.5, nor does it require `RAND()` to produce a different number every time it is called. Such a requirement would not make sense; by definition, a random number generator is permitted to return whatever result it wants. So according to this operational semantics, `RAND()` might consistently return 0.42 whenever it is called, although that would be rather disappointing and useless.

- Rule (e5e) says that a call $\texttt{F}(e_1, \ldots, e_n)$ to a function `F` may evaluate to error v_i if one of its arguments e_1 may evaluate to error v_i. Note that if more than one argument may evaluate to an error, then the function call may evaluate to any of these. Hence, the semantics does not prescribe an evaluation order for arguments, such as a left to right or right to left.

- Rule (e5v) says that a call $\texttt{F}(e_1, \ldots, e_n)$ to a function `F` may evaluate to value v if each argument e_i may evaluate to non-error value v_i, and applying the actual function f to arguments $(v_1, \ldots, v_n)$ produces value v. The final result v may be a number such as 5, for instance, if the call is $+(2, 3)$; or it may be an error such as `#DIV/0!`, for instance, if the call is `/(1.0, 0.0)`.

1.8.3 Semantics of recalculation

Now that we know how to evaluate a formula, given values of all cells in the worksheet, we can describe the semantics of a recalculation. A recalculation must find a value for every non-blank cell ca in the sheet, and that value $\sigma(ca)$ must agree with the formula $\phi(ca)$ held in that cell. These are the central consistency requirements on a recalculation, informally stated in section 1.7.2 and formally described in figure 1.11. These requirements leave it completely unspecified how the recalculation works, whether it recalculates all or only some cells, whether it does so sequentially or in parallel, whether it guesses the values or computes them, and so on. This underspecification is essential to permit a range of implementation strategies and optimizations.

(1) $dom(\sigma) = dom(\phi)$
(2) $\forall ca \in dom(\phi).\sigma \vdash \phi(ca) \Downarrow \sigma(ca)$

Figure 1.11: The consistency requirements on a recalculation. Requirement (1) says that a recalculation must find a value $\sigma(ca)$, possibly an error, for every non-blank cell ca. Requirement (2) says that the computed value $\sigma(ca)$ must agree with the cell's formula $\phi(ca)$.

1.8.4 Properties of the semantics

We believe the semantics given by the formula evaluation rules (figure 1.10) and the consistency requirements on recalculation (figure 1.11) faithfully models actual spreadsheet implementations such as Microsoft Excel, OpenOffice Calc, Gnumeric, and Corecalc as described in this book. In particular:

- It prescribes that all references to a non-blank cell $ca \in dom(\sigma)$ produce the same value. This may be an error value.
- It prescribes that a reference to a blank cell $ca \notin dom(\sigma)$ produces the non-error value 0.0.
- It prescribes error propagation from arguments to results, as informally required in section 1.7.8, thanks to rules (e3e) and (e5e).
- It prescribes non-propagation of errors from the unevaluated branch of an `IF`-expressions, as informally required in section 1.7.4, thanks to rules (e3f) and (e3t).
- Even though neither the formula semantics in figure 1.10 nor the consistency requirements in figure 1.11 mention cyclic dependencies, the semantics does provide a useful model of recalculation in the presence of one or more cyclic dependencies (see section 1.8.5).
- It seems that the semantics can be extended to account for functions such as `INDIRECT`, `COUNTIF`, and `SUMIF` that interpret strings as formulas or cell references (see section 1.8.6).

1.8.5 Formal semantics of cyclic dependencies

If a sheet has a cyclic dependency, then it may be impossible for recalculation to find non-error values for the cells involved in the cycle. For instance, if cell B2 contains the formula `=B2+1`, then there is no proper (finite and non-NaN) number value $y = \sigma(\texttt{B2}) \in Number$ such that $y = y + 1$. Due to requirement (1) from figure 1.11, recalculation must find *some* value for that cell.

The recalculation process can satisfy the requirements simply by giving cell B2 an error value such as `#CYCLE!`. In that case, the formula semantics rule (e5e) will ensure that the formula `B2+1`, or `+(B2,1)`, evaluates to error `#CYCLE!` too, hence fulfilling requirement (2) on recalculation consistency. In fact, any error value will satisfy requirement (2), but `#CYCLE!` is the most sensible error to choose. This would implement the informal requirement from section 1.7.6 to discover dynamic dependency cycles.

Note that recalculation cannot gratuitously produce error values in the absence of cyclic dependencies. If cell B2 contains the formula `=41+1`, then by requirement (2) the result $\sigma(\texttt{B2})$ must be 42, not an error. More generally, if cell ca depends only on cells that have no cyclic dependencies, if no function called from those cells'

formulas produces a `#CYCLE!` error, and if the spreadsheet is finite, then cell $\sigma(ca)$ cannot be `#CYCLE!`. This can be proved by induction on the depth of dependencies.

On a more speculative note, some cyclic dependencies might conceivably have non-error solutions. For instance, if cell B2 contains the formula `=5+B2/2`, then a particularly clever recalculation mechanism may find the value $\sigma(\mathrm{B2}) = 10.0$ for B2, satisfying requirement (2). None of the standard spreadsheet implementations does that, reporting instead a cyclic dependency. However, they typically report the cycle not through an error value but through a dialog box, a status flag, or a marker in the affected cell itself. This allows spreadsheet users to perform multiple recalculations to see whether the computed value converges. However, this practice is dangerous, as the spreadsheet may never reach the consistency one usually expects, and which is embodied in our consistency requirement (2) in figure 1.11. Strange conclusions—within finance, science, or engineering—may be drawn from such circular spreadsheets.

1.8.6 The semantics of reflective functions

Spreadsheet implementations provide a few built-in functions that interpret strings as formulas or cell references, such as `INDIRECT("B2")`, which evaluates to the value of cell B2, and `COUNTIF(A5:A9, "< 10")`, which counts the number of values in A5:A9 that are less than 10. Moreover, the string argument may be computed from other data, as in `COUNTIF(A5:A9, "<" & B2)`, which counts the number of values in A5:A9 that are smaller than the value in cell B2.

Another example is `INDIRECT("B"&FLOOR(1+500*RAND(),1))`, which refers to a random cell among B1:B500 depending on the outcome of `RAND()`.

What is unusual about such "reflective" functions is that their evaluation may refer to cells and ranges that are not explicitly given as arguments.

Nevertheless, it seems that the formula semantics can be extended to account for reflective functions simply by passing σ as an additional argument to the underlying function. For instance, the meaning of $\texttt{INDIRECT}(e_1)$ can be described like this:

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \notin \mathit{Error}}{\sigma \vdash \texttt{INDIRECT}(e_1) \Downarrow \mathit{indirect}(\sigma, v_1)}$$

The underlying $\mathit{indirect}$ function must check that v_1 is a string and has the correct format for a cell reference ca and then look up and return $\sigma(ca)$, or else return an error. The same treatment should work for `COUNTIF`, `SUMIF`, and related functions.

1.9 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

[83], risks and mistakes in spreadsheet use, and techniques to avoid them [126]. The European Spreadsheet Risks Interest Group (EuSpRiG) [53] holds an annual conference on the topic. The EUSES consortium (for *End Users Shaping Effective Software*) has published many papers on how to mitigate spreadsheet risks [54] as well as the *EUSES Spreadsheet Corpus* [58] that enables empirical investigation of real-world spreadsheets. Ray Panko maintains a website on spreadsheet error research [118], and a spreadsheet analytics company maintains a comprehensive bibliography on spreadsheet errors and testing techniques [80].

More recently, Felienne Hermans and others have started applying general software engineering concepts, principles, and tools to the development of spreadsheet models [76].

However, our main interest here is spreadsheet implementation and variations and extensions on the spreadsheet concept. Literature in that area includes Piersol's 1986 paper [121] 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, described in section 2.12.

De Hoon's 1995 MSc thesis [41] and related papers [42] 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 [40] and Lisper and Malmström's Haxcel interface to Haskell [93].

Nuñez's remarkable 2000 MSc thesis [113] presents an extended spreadsheet system called ViSSh (Visualization Spreadsheet). 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 (for each cell, a set of the cells that it refers to) and an explicit support graph (for each cell, a set of the cells that refer to it). Such explicit representations can be very memory-consuming when there are multiple copies of formulas with cell area arguments, as discussed in section 3.3.2. Chapters 4 and 5 describe more compact symbolic representations.

Jocelyn Paine describes how to generate a VBA (Visual Basic for Applications) function from Excel formulas [117]. This work is inspired by [120] but unfortunately seems somewhat incomplete. For instance, it appears not to handle Excel's `IF` function, which indeed requires special treatment and in particular if recursion is allowed (see chapter 9).

Wang and Ambler developed an experimental spreadsheet program called Formulate [161]. 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 [28], 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 available (for non-commercial use) in binary form for the Sun Solaris and HP-UX operating systems, but it does not appear to be available in source form.

A MITRE technical report [61] by Francoeur presents a recalculation engine, called 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 (section 1.7.5). There is no discussion of the size of the dependency graph or techniques for representing it compactly. The ExcelComp implementation is not available to the public [62].

Yoder and Cohn have written a whole series of papers on spreadsheets, data-flow computation, and parallel execution. Topics include the relation among spreadsheet computation, demand-driven (eductive, lazy) and data-driven (eager) evaluation, parallel evaluation, and generalized indexing notations [168], as well as the design of a spreadsheet language Mini-SP with array values and recursion (not unlike Corecalc) and a case study solving several non-trivial computation problems [169]. They also present 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 and chapter 4) is used to perform minimal recalculation and schedule parallel execution [167, 170].

Clack and Braine present a spreadsheet paradigm that includes features from functional programming, such as higher-order functions, as well as features from object-oriented programming, such as virtual methods and dynamic dispatch [34].

None of the investigated implementations appears to use the sharing-preserving formula representation of Corecalc.

In addition to Yoder and Cohn's papers mentioned above, there are a few other papers on parallelization of spreadsheet computations. For instance, Wack [159] investigates how the dependency graph can be used to schedule parallel computation.

Lew and Halverson [90] proposed creating field-programmable custom hardware for spreadsheet evaluation. Custom circuitry realizing a particular spreadsheet's formula would be generated at run-time by configuring an FPGA (field-programmable gate array) chip attached to a desktop computer. This can be thought of as run-time hardware generation, an extreme form of run-time code generation. In addition, it should be possible to perform computations in parallel; spreadsheets lend themselves well to parallelization because of a fairly static dependency structure.

A paper by Stadelmann [147] describes a spreadsheet paradigm that uses equational constraints (as in constraint logic programming) instead of unidirectional formulas. Some patents (numbers 11 and 16) propose a similar idea. This seriously changes the recalculation machinery needed; Stadelmann used Wolfram's Mathematica [166] tool to compute solutions.

A spreadsheet paradigm that computes with intervals, or even interval constraints, is proposed by Hyvönen and de Pascale [43, 78, 79].

The interval computation approach was used in the PhD thesis [11] of Ayalew as a tool for testing spreadsheets. A user 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 methods for spreadsheet testing, in particular the Wysiwyt or "What You See Is What You Test" approach [29, 127, 128, 129, 57] within the EUSES consortium [144]. This work is also the subject of patents 9 and 10, listed in appendix B.

The research group *SpreadSheets as a Programming Paradigm* (SSaaPP) [9] at the University of Minho investigates extensions of the spreadsheet paradigm and spreadsheet use, such as their interaction with relational databases [39] and model-driven spreadsheet engineering [38].

Gulwani and others [71] have investigated how to synthesize spreadsheet functions or formulas from a small set of examples of the transformations desired.

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 include Erwig and Burnett [51], Ahmad and others [6], Antoniu and others [8], Coblenz [35], Abraham and Erwig [2, 4, 5], Chambers [31], and Cheng and Rival [33].

1.10 Online resources and implementations

The company Decision Models sells advice on how to improve recalculation times for Excel spreadsheets, and in that connection it provides useful technical information about Excel's implementation (see section 3.3.5) on its website [45]. Charles Williams maintains an interesting blog *Excel and UDF Performance Stuff* with technical information about Excel and advice on its efficient use [165].

There are quite a few open source spreadsheet implementations in addition to the modern comprehensive implementations Gnumeric [66] and OpenOffice Calc [115], already mentioned. A Unix classic is `sc`, originally written by James Gosling and now maintained by Chuck Martin [97], 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 MS DOS era spreadsheet programs.

A comprehensive and free spreadsheet program is Abykus [143] by Brad Smith. This program is not open source and presents a number of generalizations and deviations relative to the mainstream (Excel, OpenOffice Calc, and Gnumeric).

One managed code open source spreadsheet program is Vincent Granet's XXL [70], written in STk, a version of Tk based on the Scheme programming language. Another one, currently less developed, is Einar Pehrson's CleanSheets [119], which is written in Java. Martin Manns' Pyspread uses Python code for cell formulas [95]. More spreadsheet programs—historical, commercial, or open source—are listed on Chris Browne's spreadsheet website [23], with historical notes connecting them. An-

other source of useful information is the list of frequently asked questions [136] from the Usenet newsgroup `comp.apps.spreadsheets`, although the last update was in June 2002. The newsgroup itself [155] seems to be devoted mainly to spreadsheet application and does not appear to receive much traffic.

There are a number of commercial closed source managed code implementations of Excel-compatible spreadsheet recalculation engines, graphical components, and report generators. Two such implementations are SpreadsheetGear for .NET [146] and the seemingly defunct Formula One for Java [124]. The lead developer for both is (or was) Joe Erickson. Two other implementations are KDCalc [82] from Knowledge Dynamics Inc. and SpreadsheetConverter by Framtidsforum AB [60]. Such implementations are typically used to implement spreadsheet logic on servers without the need to manually 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 spreadsheet program distributed in source form with Borland Turbo Pascal 1.0 (November 1983), still available at Borland's "Antique Software" site [20]. Another example is the spreadsheet chapter in John English's Ada95 book [50, chapter 18]; however, this is clearly not designed with efficiency in mind.

1.11 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 [114], the US Patents and Trademarks Office [154], or Google Patents [69]. As of June 2013, there were 611 granted US patents in which the word "spreadsheet" appears in the title or abstract and many more patent applications. Appendix B lists several patents and patent applications that appear to be concerned with the *implementation* rather than the *use* of spreadsheets. A separate technical report [140] lists many more spreadsheet implementation patents.

Some patents of interest are:

- Harris and Bastian at WordPerfect Corporation have a patent, number 17 in appendix B, on a method for "optimal recalculation", further discussed in section 3.3.7.

- Roger Schlafly has two patents, numbers 13 and 15 in appendix B, that describe run-time compilation of spreadsheet formulas to x86 code. A distinguishing feature is the clever use of the math coprocessor and the then relatively new IEEE 754 binary floating-point number representation, and especially its NaN (not-a-number) values, to achieve very fast formula evaluation (see section 2.8.1).

- Bruce Jones and others at Microsoft have a patent, number 3 in appendix B, on multiprocessor recalculation of spreadsheet formulas. It includes a description of the uniprocessor recalculation model that agrees with that given by La Penna [88], summarized in section 3.3.5.

In fact, in one of the very first software patent controversies, several major spreadsheet implementors were sued in 1989 for infringing on US Patent No. 4398249, filed by Rene K. Pardo and Remy Landau in 1970 and granted in 1983 [86]; it is number 18 in appendix B. The patent basically describes an application of topological sorting and appears to contain no technological innovation. The United States Court of Appeals for the Federal Circuit in 1996 upheld the District Court's ruling that the patent is unenforceable [153].

Of particular relevance to Funcalc, described in part II of this book, are the surprisingly many patents and patent applications that claim to have invented compilation of spreadsheet models to more traditional kinds of code, similar to the compiled-mode version of Francoeur's implementation [61] mentioned above:

- Schlafly's patents (numbers 13 and 15 in appendix B) describe compilation of individual formulas to x86 machine code.
- Khosrowshahi and Woloshin's patent (number 8) describes compilation of a spreadsheet model with designated input and output cells to code in a procedural programming language.
- Rank and Pampuch's patent (number 2) describes the idea, but few technical details, of cross-compilation of spreadsheet formulas for space-conserving execution on small-memory mobile devices. This involves, for instance, leaving out unused library functions.
- Rubin and Smialek have a series of patents (including number 1) that describe a spreadsheet recalculation engine, as well as the compilation of individual formulas to source code in Java and other languages. It does not seem to handle non-strict functions such as `IF` specially, and thus hardly is faithful to Excel or OpenOffice Calc semantics. Probably the system described is the commercial tool KDCalc [82], which allows Excel workbooks to be compiled to web applications and more.
- Waldau's patent application (number 7) describes cross-compilation to another platform, such as a mobile phone or web service. This is a technically substantial patent with references to relevant prior art, such as Schlafly's patents. It describes compilation to dynamically typed and statically typed languages (JavaScript and Java), and how to present the generated code as a WML service, say. Probably the technology described by this application is that used in the SpreadsheetConverter product [60].
- Tanenbaum's patent applications (number 5 and 6) describe compilation of a spreadsheet model with designated input and output cells to C source code.

Bibliography

[1] Robin Abraham, Margaret Burnett, and Martin Erwig. Spreadsheet programming. In B.J. Wah, editor, *Encyclopedia of Computer Science and Engineering*, pages 2804–2810. Wiley, 2009.

[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 on 9 June 2013.

[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] Robin Abraham and Martin Erwig. Ucheck: A spreadsheet type checker for end users. *Journal of Visual Languages and Computing*, 18(1):71–95, February 2007.

[6] 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.

[7] A.V. Aho, M. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[8] 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.

[9] SpreadSheets as a Programming Paradigm. Project homepage. Website. At http://ssaapp.di.uminho.pt/ on 10 May 2014.

[10] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: A two-stage DSL embedded in Haskell. In *International Conference on Functional Programming (ICFP'08)*, pages 225–228. ACM, September 2008.

[11] Yirsaw Ayalew. *Spreadsheet Testing Using Interval Analysis*. PhD thesis, Institut für Informatik-Systeme, Universität Klagenfurt, 2001. At http://www.isys.uni-klu.ac.at/PDF/2001-0125-YA.pdf on 9 June 2013.

[12] Dermot Balson and Jerzy Tyszkiewicz. User defined spreadsheet functions in Excel. Presentation, EuSpRiG 2012, July 2012. At http://www.eusprig.org/presentations-2012.htm on 5 January 2014.

[13] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A new approach to incremental cycle detection and related problems. Preprint, December 2011. At http://arxiv.org/abs/1112.0784 on 15 June 2013.

[14] Lee Benfield. FMD: functional development in Excel. At Commercial Users of Functional Programming (CUFP), Edinburgh 2009 ACM SIGPLAN. Video presentation, 2009. At http://cufp.org/videos/fmd-functional-development-excel on 12 January 2014.

[15] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

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

[17] Luca Bolognese. Excel financial functions for .NET. Website, 2009. At http://archive.msdn.microsoft.com/FinancialFunctions on 9 June 2013.

[18] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[19] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[20] Borland. Antique software: Turbo Pascal v1.0. Website. At http://edn.embarcadero.com/article/20693 on 9 June 2013.

[21] Dan Bricklin. Visicalc information. Website. At http://www.danbricklin.com/visicalc.htm on 9 June 2013.

[22] Dan Bricklin. *Bricklin on technology*. John Wiley and Sons, 2009.

[23] Chris Browne. Linux spreadsheets. Website. At http://linuxfinances.info/info/spreadsheets.html on 5 January 2014.

[24] Poul Brønnum. Type analysis for sheet-defined functions. Master's thesis, IT University of Copenhagen, 2009.

[25] Mikhail A. Bulyonkov Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[26] William H. Burge *Recursive Programming Techniques*. Addison-Wesley, 1975.

[27] Margaret Burnett, John Attwood, and Zachary Welch. Implementing level 4 liveness in declarative visual programming languages. In *1998 IEEE Symposium on Visual Languages*, pages 126–133. IEEE, 1998.

[28] 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.

[29] 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.

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

[31] Chris Chambers and Martin Erwig. Dimension inference in spreadsheets. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VLHCC'08*, pages 123–130. IEEE Computer Society, 2008.

[32] Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[33] Tie Cheng and Xavier Rival. An abstract domain to infer types over zones in spreadsheets. In Antoine Miné and David Schmidt, editors, *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France. Lecture Notes in Computer Science, vol. 7460*, pages 94–110. Springer, 2012.

[34] Chris Clack and Lee Braine. Object-oriented functional spreadsheets. In *Proceedings of the 10th Glasgow Workshop on Functional Programming (GlaFP'97)*, September 1997.

[35] 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 on 9 June 2013.

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

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

[38] Jácome Cunha et al. MDSheet: A framework for model-driven spreadsheet engineering. In *International Conference of Software Engineering (ICSE)*, pages 1412–1415, 2012.

[39] Jácome Cunha, João Saraiva, and Joost Visser. From spreadsheets to relational databases and back. In *Partial Evaluation and Program Manipulation (PEPM)*, pages 179–188. ACM, 2009.

[40] 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 on 9 June 2013.

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

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

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

[44] Decision Models. Excel pages – calculation secrets. Website. At http://www.decisionmodels.com/calcsecrets.htm on 22 October 2013.

[45] Decision Models. Homepage. Website. At http://www.decisionmodels.com/ on 9 June 2013.

[46] Nachum Dershowitz and Edward M. Reingold. *Calendrical calculations*. Cambridge University Press, third edition, 2008.

[47] Jack Doweck. Inside Intel core microarchitecture and smart memory access. Whitepaper, 2006. At http://software.intel.com/sites/default/files/m/3/4/d/6/3/18374-sma.pdf on 11 June 2013.

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

[49] Ecma TC39 TG3. *Common Language Infrastructure (CLI). Standard ECMA-335*. Ecma International, sixth edition, June 2012.

[50] John English. *Ada 95: The Craft of Object-Oriented Programming*. Prentice-Hall, 1997. At http://faculty.cs.wwu.edu/reedyc/AdaResources/bookhtml/contents.htm on 9 June 2013.

[51] 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. Springer-Verlag, 2002.

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

[53] European Spreadsheet Risks Interest Group. Homepage. Website. At http://www.eusprig.org/ on 9 June 2013.

[54] EUSES Consortium. End users shaping effective software. Website. At http://eusesconsortium.org/ on 5 January 2014.

[55] Excel DNA Project. Homepage. At http://exceldna.codeplex.com/ on 28 February 2014.

[56] Raphael A. Finkel and Jon L. Bentley. Quad trees as a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[57] 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.

[58] Marc Fisher and Gregg Rothermel. The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *First workshop on end-user software engineering (WEUSE)*, pages 1–5. ACM, 2005.

[59] Marc Fisher, Gregg Rothermel, Tyler Creelan, and Margaret Burnett. Scaling a dataflow testing methodology to the multiparadigm world of commercial spreadsheets. In *17th International Symposium on Software Reliability Engineering*, pages 13–22. IEEE, 2006.

[60] Framtidsforum. SpreadsheetConverter. Website. At http://www.spreadsheetconverter.com/ on 9 June 2013.

[61] 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 on 9 June 2013.

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

[63] Bob Frankston. Implementing VisiCalc. Website, April 2003. At http://www.frankston.com/public/?name=implementingVisicalc on 9 June 2013.

[64] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[65] Michael R. Garey and David S. Johnson *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[66] Gnumeric. Homepage. Website. At http://projects.gnome.org/gnumeric/ on 9 June 2013.

[67] David Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, 23(1):5–48, March 1991.

[68] Google. Google docs and spreadsheets. Website. At http://docs.google.com/ on 9 June 2013.

[69] Google. Google patent search. Website. At http://patents.google.com/ on 9 June 2013.

[70] Vincent Granet. The XXL spreadsheet project. *Linux Journal*, April 1999. At http://www.linuxjournal.com/article/3186 on 9 June 2013.

[71] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, August 2012.

[72] 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.

[73] John F. Hart et al. *Computer Approximations*. Wiley, 1968.

[74] Haskell. Homepage. Website. At http://www.haskell.org/ on 9 June 2013.

[75] John Hatcliff, Torben Mogensen, and Peter Thiemann, editors. *Partial Evaluation: Practice and Theory: DIKU 1998 International Summer School*, volume 1706 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[76] Felienne Hermans. *Analyzing and visualizing spreadsheets*. PhD thesis, Technical University of Delft, September 2012. At http://figshare.com/articles/Analyzing_and_Visualizing_Spreadsheets/658936 on 5 January 2014.

[77] Carsten Kehler Holst. Poor man's generalization. Note, August 1988. 2 pages.

[78] 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.

[79] 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 on 9 June 2013.

[80] I-Nth. Bibliography of spreadsheet errors and testing literature. Website. At http://www.i-nth.com/resources/bibliography on 5 January 2014.

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

[82] Knowledge Dynamics Inc. Kdcalc. Website. At http://www.kdcalc.com/ on 9 June 2013.

[83] 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.

[84] 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/funcalc/Iversen2006.pdf on 9 June 2013.

[85] Neil D. Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. At http://www.itu.dk/people/sestoft/pebook/pebook.html on 9 June 2013.

[86] Brian Kahin. The software patent crisis. *Technology Review, MIT*, April 1990. At http://antipatents.8m.com/software-patents.html on 9 June 2013.

[87] Gilles Kahn. Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany (Lecture Notes in Computer Science, vol. 247)*, pages 22–39. Springer-Verlag, 1987.

[88] Loreen La Penna. Recalculation in Microsoft Excel 2002. Web page, October 2001. At http://msdn.microsoft.com/en-us/library/office/aa140058(v=office.10).aspx on 9 June 2013.

[89] Xavier Leroy. The Zinc experiment: An economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, France, 1990. At http://gallium.inria.fr/~xleroy/publi/ZINC.pdf on 9 June 2013.

[90] Art Lew and Richard 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.

[91] Serge Lidin. *.NET 2.0 IL Assembler*. Apress, 2006.

[92] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Oracle, Java SE 7 edition, 2013. At http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf on 15 June 2013.

[93] 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 9 June 2013.

[94] Haibo Luo. ILVisualizer. Homepage. At http://blogs.msdn.com/b/haibo_luo/archive/2010/04/19/9998595.aspx on 9 June 2013.

[95] Martin Manns. Pyspread. Website, 2013. At http://manns.github.io/pyspread/ on 5 January 2014.

[96] Bill Manville. Update linked cells within a workbook??? ExcelBanter online forum posting, reply 20 January, 2005. At http://www.excelbanter.com/showthread.php?t=557 on 13 June 2013.

[97] Chuck Martin. `sc`. Website. At http://freecode.com/projects/sc on 9 June 2013.

[98] 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/.

[99] Microsoft. Microsoft Composition. Website. At http://www.nuget.org/packages/microsoft.composition on 13 January 2014.

[100] Microsoft. Microsoft Extensibility Framework. Website. At http://mef.codeplex.com/ on 13 January 2014.

[101] Microsoft. .NET framework. Website. At http://msdn.microsoft.com/en-us/vstudio/aa496123 on 9 June 2013.

[102] Microsoft. Office online. Website. At http://office.microsoft.com/ on 9 June 2013.

[103] Robin Milner, Mads Tofte, Robert Harper, and David B. MacQueen *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[104] Vincens Riber Mink and Daniel Schiermer. Collaborative spreadsheet. BSc thesis, IT University of Copenhagen, May 2010.

[105] 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.

[106] Hanspeter Mössenböck, Albrecht Wöß, and Markus Löberbauer. The compiler generator Coco/R. Website. At http://www.ssw.uni-linz.ac.at/Coco/ on 9 June 2013.

[107] Bonnie A. Nardi and James R. Miller. The spreadsheet interface: A basis for end-user programming. In D. Diaper et al., editors, *IFIP TC13 Third International Conference on Human-Computer Interaction (INTERACT)*, pages 977–983. North-Holland, 1990. At http://www.miramontes.com/writing/spreadsheet-eup/ on 5 May 2014.

[108] Bonnie A. Nardi and James R. Miller. Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *International Journal of Man-Machine Studies*, 34:161–184, 1991.

[109] Microsoft Developer Network. Excel primary interop assembly reference. Class ApplicationClass. Website. At http://msdn.microsoft.com/en-us/library/microsoft.office.interop.excel.applicationclass.aspx on 9 June 2013.

[110] Microsoft Developer Network. XL97: How to obtain the Excel 97 auto recalculation patch. Website, January 2007. At http://support.microsoft.com/kb/q174868/ on 15 June 2013.

[111] Gregory Neverov and Paul Roe. Cross-stage persistence in Metaphor. In *First MetaOCaml Workshop, Vancouver, Canada*, pages 168–185, October 2004.

[112] H.R. Nielson and F. Nielson. *Semantics with Applications. An Appetizer*. Springer-Verlag, 2007.

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

[114] European Patent Office. Espacenet. Website. At http://worldwide.espacenet.com/ on 9 June 2013.

[115] OpenOffice. Calc – the all-purpose spreadsheet. Website. At http://www.openoffice.org/product/calc.html on 9 June 2013.

[116] Niek Otten. Re: Ctrl+Alt+F9 not performing full recalculation on some PCs. Excel Forum posting, 8 October 2006, 2006. At http://www.excelforum.com/excel-worksheet-functions/570413-ctrl-alt-f9-not-performing-full-recalculation-on-some-pcs.html on 9 June 2013.

[117] Jocelyn Paine. Defining Excel functions without Visual Basic: a compiler that converts Excel function definition sheets to VBA. Website. At http://www.j-paine.org/dobbs/udfs.html on 5 January 2014.

[118] Ray Panko. Spreadsheet research. Website. At http://panko.shidler.hawaii.edu/SSR/ on 5 January 2014.

[119] Einar Pehrson. Cleansheets. Website. At http://freecode.com/projects/csheets.

[120] 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. ACM, 2003.

[121] 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.

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

[123] Jonas Druedahl Rask and Simon Eikeland Timmermann. Funsheet. Integration of sheet-defined functions in Excel using C#. Master's thesis, IT University of Copenhagen, June 2014.

[124] ReportingEngines. Formula One for Java. Website. At http://www.mit.edu/~mbarker/formula1/ on 9 June 2013.

[125] Jeffrey Richter. *CLR via C#*. Microsoft Press, fourth edition, 2012.

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

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

[128] 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.

[129] 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.

[130] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, California, February 1993. Published as technical report CSL-TR-93-563.

[131] Erik Ruf and Daniel Weise. Opportunities for online partial evaluation. Technical Report CSL-TR-92-516, Computer Systems Laboratory, Stanford University, Stanford, CA, April 1992.

[132] Jonathan Sachs. Recollections: Developing Lotus 1-2-3. *IEEE Annals of the History of Computing*, 29(3):41–48, July-September 2007.

[133] Jorma Sajaniemi. Modeling spreadsheet audit: a rigorous approach to automatic visualization. *Journal of Visual Languages and Computing*, 11(1):49–82, 2000.

[134] Nader Salas. Collaborative spreadsheet with traceability. Master's thesis, IT University of Copenhagen, August 2011.

[135] Chris Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.

[136] Russell Schulz. comp.apps.spreadsheet FAQ. Newsgroup, June 2002. At http://www.faqs.org/faqs/spreadsheets/faq/ on 9 June 2013.

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

[138] Peter Sestoft. Numeric performance in C, C# and Java. Technical report, IT University of Copenhagen, February 2009. 14 pages. At http://www.itu.dk/people/sestoft/papers/numericperformance.pdf on 9 June 2013.

[139] Peter Sestoft. Online partial evaluation of sheet-defined functions. In A. Banerjee, O. Danvy, K. Doh, and J. Hatcliff, editors, *Semantics, Abstract Interpretation, and Reasoning about Programs*, volume 129 of *Electronic Proceedings in Theoretical Computer Science*, pages 136–160, 2013.

[140] Peter Sestoft. Spreadsheet patents. Technical Report ITU-TR-2014-178, IT University of Copenhagen, 2014. ISBN 978-87-7949-317-9. (To appear).

[141] Peter Sestoft and Jens Zeilund Sørensen. Sheet-defined functions: implementation and initial evaluation. In Y. Dittrich et al., editors, *International Symposium on End-User Development, June 2013*, volume 7897 of *Lecture Notes in Computer Science*, pages 88–103, 2013.

[142] Charles Severance. An interview with William Kahan. *IEEE Computer*, 31(3):114–115, March 1998.

[143] Bradford L. Smith. Abykus. An object-oriented spreadsheet for windows. Website. At http://www.abykus.com/ on 9 June 2013.

[144] EUSES: End Users Shaping Effective Software. Wysiwyt: What you see is what you test. Website. At http://eusesconsortium.org/wysiwyt.php on 9 June 2013.

[145] Michael Sperber et al., editors. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

[146] SpreadsheetGear LLC. SpreadsheetGear for .NET. Website. At http://www.spreadsheetgear.com/ on 9 June 2013.

[147] 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.

[148] Statfactory Ltd.. FCell add-in. Webpage. At http://www.statfactory.co.uk/fcell-add-in/ on 12 January 2014.

[149] Jørgen Staunstrup. *A Formal Approach to Program Design*. Kluwer, 1994.

[150] Microsoft Support. How formula calculations are performed in Excel. Website, September 2011. At http://support.microsoft.com/kb/825012 on 5 January 2014.

[151] Jens Zeilund Sørensen. An evaluation of sheet-defined financial functions in Funcalc. Master's thesis, IT University of Copenhagen, March 2012.

[152] S. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, June 1990.

[153] United States Court of Appeals for the Federal Circuit. Refac versus Lotus. Opinion 95-1350, April 1996. At http://caselaw.findlaw.com/us-federal-circuit/1339862.html on 9 June 2013.

[154] United States Patent and Trademark Office. Patent full-text and full-page image databases. Website. At http://patft.uspto.gov/ on 9 June 2013.

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

[156] Johannes G. van der Corput. Verteilungsfunktionen. *Proc. Ned. Akad. v. Wet.*, 38:813–821, 1935.

[157] Michael van Schothorst et al. Relating microbiological criteria to food safety objectives and performance objectives. *Food Control*, 20:967–979, 2009.

[158] Noah Vawter. DFT multiply demo spreadsheet. Website, 2002. At http://www.gweep.net/~shifty/portfolio/fftmulspreadsheet/ on 9 June 2013.

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

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

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

[162] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 165–191. Springer-Verlag, 1991.

[163] Wikipedia. Spreadsheet. Website. At http://en.wikipedia.org/wiki/Spreadsheet on 9 June 2013.

[164] Wikipedia. Visicalc. Website. At http://en.wikipedia.org/wiki/VisiCalc on 9 June 2013.

[165] Charles Williams. Excel and UDF performance stuff. Website. At http://fastexcel.wordpress.com/ on 5 January 2014.

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

[167] 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://www3.nd.edu/~csesoft/tech_reports/1993.html on 9 June 2013.

[168] 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 http://www3.nd.edu/~csesoft/tech_reports/1994.html on 9 June 2013.

[169] 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. At http://www3.nd.edu/~csesoft/tech_reports/1994.html on 9 June 2013.

[170] 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.