

Rewriting High-Level Spreadsheet Structures into Higher-Order Functional Programs

Florian Biermann¹(✉) , Wensheng Dou² , and Peter Sestoft¹ 

¹ Computer Science Department, IT University of Copenhagen,
Copenhagen, Denmark
{fbie,sestoft}@itu.dk

² State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing, China
wsdou@otcaix.iscas.ac.cn

Abstract. Spreadsheets are used heavily in industry and academia. Often, spreadsheet models are developed for years and their complexity grows vastly beyond what the paradigm was originally conceived for. Such complexity often comes at the cost of recalculation performance. However, spreadsheet models usually have some high-level structure that can be used to improve performance by performing independent computation in parallel. In this paper, we devise rules for rewriting high-level spreadsheet structure in the form of so-called cell arrays into higher-order functional programs that can be easily parallelized on multicore processors. We implement our rule set for the experimental Funcalc spreadsheet engine which already implements parallelizable higher-order array functions as well as user-defined higher-order functions. Benchmarks show that our rewriting approach improves recalculation performance for spreadsheets that are dominated by cell arrays.

1 Introduction

Spreadsheets are abundant in research and industry and used heavily by professionals who are not educated as programmers. Spreadsheets often become highly complex over time. Not only is it hard to maintain an understanding of the underlying model, but this complexity can also lead to slow recalculation of the entire spreadsheet. For large and complex spreadsheet models, recalculation performance may be critical.

Dou et al. [6] report that 69% of all spreadsheets with formulas in the Enron [10] and EUSES [8] spreadsheet corpora contain *cell arrays*. A cell array is a rectangular block of copy equivalent formulas [15], like the cell areas B2:F6 and B8:F12 in Fig. 2. Such a cell array is created when the spreadsheet user writes a formula, typically with a carefully crafted mix of absolute and relative references, and copies it to a rectangular cell range.

F. Biermann—Supported by the Sino-Danish Center for Education and Research.

W. Dou—Supported by the National Natural Science Foundation of China (61702490) and Beijing Natural Science Foundation (4164104).

Spreadsheets are first-order purely functional programs [2]. In purely functional programs, all values are immutable. Immutability guarantees data-race freedom and therefore allows for easy parallelization and hence a speedup of disjoint computations. If we can detect formula cells on a spreadsheet that do not depend on each other, we can safely compute these in parallel.

In functional languages, disjoint computations on values of an array can be expressed explicitly by means of higher-order functions. For instance, the higher-order function *map* explicitly applies a pure function to each element of an array individually. Hence, *map* can easily be parallelized.

In this paper, we design a source-to-source rewriting semantics for converting cell array computations into parallel higher-order functional programs to improve recalculation performance. We do this by correlating cell array structure with higher-order array functions.

Our rewriting semantics uses a common feature of spreadsheet software, called *array formulas*. An array formula must evaluate to an array of the same size and shape as the spreadsheet cell range that contains the formula. The array is then unpacked and its scalar values are placed directly in the cells according to their position in the array, such that the containing array disappears.

We target the experimental spreadsheet engine Funcalc [17]. Funcalc provides higher-order functions on immutable two-dimensional arrays, which correspond to cell ranges, as well as efficient sheet-defined higher-order functions. For our purpose, we extend Funcalc with additional functions on arrays.

To our knowledge, there is no literature on exploiting parallelism in cell arrays to improve recalculation performance. Some researchers have investigated whole-sheet graph parallelism on spreadsheets [19–21]. Prior work on high-level spreadsheet array structure has either focused on making the user aware of high-level models [11, 15, 16]; on correcting errors in cell formulas by analyzing the structure around given cells [3, 4, 6, 12]; or on synthesizing templates from spreadsheets to allow for reuse of the high-level structure [1, 13].

With our rewriting semantics, Funcalc can exploit implicit parallelism in spreadsheets dominated by large or computation-heavy cell arrays. We compare the performance of our approach on two idealized and six synthetic spreadsheets as well as twelve real-world spreadsheets from the EUSES [8] corpus. Our results show that we can indeed improve spreadsheet recalculation by parallelizing cell array computations. However, our results also show that the achievable speedup is limited by the sequential dependencies of the spreadsheet models.

2 From Cell Arrays to Higher-Order Functions

Our idea is based on the observation that the references in cell arrays often form a pattern that corresponds to one of two higher-order functions on 2D arrays [20, 21]. We define our variants of these two functions with variadic arity k to make them as general as possible.

The first function is commonly known as *map*. It takes as arguments a k -ary function f and k arrays x_1 through x_k of n rows and m columns each. We say

they are of shape $n \times m$. We require at least one argument array, i.e. $k \geq 1$. The result of the function is a new $n \times m$ array containing the results of applying f to the k elements of the input arrays at the same index:

$$\text{MAP}(f, x_1, \dots, x_k) = X, \text{ where } X[i, j] = f(x_1[i, j], \dots, x_k[i, j])$$

The other function is commonly known as *scan* or *generalized prefix sum*. We use a variant on 2D arrays that computes a wavefront prefix sum for arbitrary functions. It takes a $(k+3)$ -ary function f , a $n \times 1$ single-column array γ , a scalar value δ and a $1 \times m$ single-row array ρ as well as, again, k arrays x_1, \dots, x_k of shape $n \times m$. Its result is a new $n \times m$ array $\text{PREFIX}(f, \gamma, \delta, \rho, x_1, \dots, x_k) = X$, where:

$$X[1, 1] = f(\gamma[1], \delta, \rho[1], x_1[1, 1], \dots, x_k[1, 1]) \quad (1)$$

$$X[1, j] = f(X[1, j-1], \rho[j], x_1[1, j], \dots, x_k[1, j]) \quad (2)$$

$$X[i, 1] = f(\gamma[i], \gamma[i-1], X[j-1, 1], x_1[i, 1], \dots, x_k[i, 1]) \quad (3)$$

$$X[i, j] = f(X[i-1, j], X[i-1, j-1], X[i, j-1], x_1[i, j], \dots, x_k[i, j]) \quad (4)$$

Here we allow $k = 0$, meaning only the input arrays γ and ρ as well as the scalar δ are required. We use the values from γ , ρ and δ as if they were positioned around the upper and left fringes of the original arrays $x_1 \dots x_k$, see also Fig. 6. Equation (1) defines the first element of X at $(i, j) = (1, 1)$, on which all other values of X depend. Since no values precede it, we must refer to values from γ , δ and ρ instead. Equation (2) defines the first row and hence refers to ρ ; Eq. (3) defines the first column and therefore refers to γ . Finally, Eq. (4) is the general case for all remaining index pairs (i, j) .

2.1 A Formal Spreadsheet Language

For presentation purposes, we use a simplified formal spreadsheet language, $\lambda\text{-calc}$, as shown in Fig. 1. The e form includes lambda expressions of arbitrary arity and with named parameters. All expressions must be closed. Users are only allowed to enter expressions in u , which is a subset of e without anonymous functions and variables. References r to cells and to cell ranges are shown in the R1C1 format, but translated to the “usual” A1 format in examples. For instance, the absolute reference R6C2 in R1C1 format would in A1 format be \$B\$6, referring to column 2, row 6. Row-absolute column-relative reference R6C[2] in R1C1 format would in A1 format be G\$6 if the reference appeared anywhere in column E — that being column 5 and so the reference would be to column $5 + 2 = 7$, which is column G. See also function `lookup` in Sect. 3.1.

Function $\phi \in r \rightarrow e$ maps a cell address r to the formula $e = \phi(r)$ in that cell. When $r_1 : r_2$ is a cell array of copy equivalent formulas, we write $\phi(r_1 : r_2)$ for the common formula (see Sect. 3.1).

2.2 Example: DNA Sequence Alignment

We illustrate the rewriting of cell arrays with the spreadsheet shown in Fig. 2. It computes the optimal local alignment of two DNA sequences using the standard

$n ::= \text{Number}$		$r ::= \mathbf{R}[i]\mathbf{C}[i]$	Relative cell address.
$t ::= \text{String}$		$\mid \mathbf{R}[i]\mathbf{C} \ i$	Row-relative.
$i ::= \text{Integer}$		$\mid \mathbf{R} \ i \ \mathbf{C}[i]$	Column-relative.
$f ::= \lambda(x, \dots).e$	Anonymous function.	$\mid \mathbf{R} \ i \ \mathbf{C} \ i$	Absolute.
$\mid \mathbf{F}$	Built-in function.	$e ::= v \mid r \mid f$	
$v ::= n \mid t$		$\mid x$	Variable name.
$\mid \text{err}(t)$	Error value.	$\mid r:r$	Cell range.
$\mid [[v; \dots] \dots]$	Row-major 2D-array.	$\mid \text{IF}(e, e, e)$	Conditional.
		$\mid f(e, \dots)$	Function application.
		$\mid e \oplus e$	Short-hand for $\oplus(e, e)$.
$u ::= v \mid r \mid r:r \mid \text{IF}(u, u, u) \mid F(u, \dots) \mid u \oplus u$			

Fig. 1. The λ -calc syntax with variables and lambda expressions. Form u is a subset of e and contains “user expressions”, i.e. expressions that a user is allowed to write.

	A	B	C	D	E	F
1		A	G	C	T	A
2	T	= IF(\$A2 = B\$1, 3, -3)	= IF(\$A2 = F\$1, 3, -3)
3	G
4	T
5	T
6	T	= IF(\$A6 = B\$1, 3, -3)	= IF(\$A6 = F\$1, 3, -3)
7	0	0	0	0	0	0
8	0	= MAX(A7 + B2, A8 - 2, B7 - 2, 0)	= MAX(E7 + F2, E8 - 2, F7 - 2, 0)
9	0
10	0
11	0
12	0	= MAX(A12 + B6, A11 - 2, B11 - 2, 0)	= MAX(E12 + F6, E11 - 2, F11 - 2, 0)

Fig. 2. A spreadsheet to compute a best local DNA sequence alignment. One DNA sequence is in cells B1:F1, the other in cells A2:A6. Cells B2:F6 defines a substitution matrix. Cells B8:F12 compute the scoring matrix. Ellipses denote repeated formulas. Cell areas with light gray background have the same formula.

algorithm, based on dynamic programming (Smith-Waterman [18]). A substitution matrix s is defined in cell range B2:F6 (upper gray cell area), and the scoring matrix H in cell range B8:F12 (lower gray cell area). The substitution matrix assigns score +3 to identical nucleotides (DNA “letters”) and score -3 to distinct nucleotides.

The scoring matrix (B8:F12) computes the best score $H(i, j)$ for any alignment between the i -length prefix of one sequence with the j -length prefix of the other. This can be defined recursively as:

$$H(i, j) = \max(H(i-1, j-1) + s(i, j), H(i-1, j) - 2, H(i, j-1) - 2, 0)$$

By backtracking through the scoring matrix H from its maximal entry, one obtains the optimal local alignment of the two sequences.

2.3 Intuitive Rewriting of Cell Arrays

First consider the range B2:F6, whose formulas are copy equivalent [15]: it could be filled by copying the formula in B2 to B2:F6 with automatic adjustment of relative row and column references. (The B8:F12 formulas are copy equivalent also). In R1C1 reference format, the range B2:F6 (upper gray cell area) can be written as:

$$\phi(\text{R2C2}:\text{R6C6}) := \text{IF}(\text{R}[0]\text{C1} = \text{R1C}[0], 3, -3)$$

The row- and column-relative structure of the two references builds a cross-product of the column and the row containing the input sequences. While it is straightforward to build such an ad-hoc cell structure, this has two disadvantages. First, this implementation does not generalize to sequences with more than five elements. Second, and more important to us, the formula itself does not capture the structure of the computation. This structure is implicit in the cell references and only emerges from the context — the entire spreadsheet and the formula’s location in it — in which it is computed.

Ideally, we would like to retain high-level information about the computation that we want to perform inside the expression, and also find the most general way to express it. Our intuition as functional programmers is to rewrite the formulas as a 2D MAP over repeated row and column values:

$$\begin{aligned} \phi(\text{R2C2}:\text{R6C6}) := \{ & \text{MAP}(\lambda(x, y). \text{IF}(x = y, 3, -3)), \\ & \text{HREP}(\text{COLS}(\text{R1C2}:\text{R1C6}), \text{R2C1}:\text{R6C1}), \\ & \text{VREP}(\text{ROWS}(\text{R2C1}:\text{R6C1}), \text{R1C2}:\text{R1C6}) \} \end{aligned}$$

The curly braces around the expression denote an *array formula*: a formula that evaluates to an array and whose values are unpacked into the individual cells of the cell array R2C2:R6C6 (B2:F6), as described in Sect. 1.

Now, this expression may look convoluted at first sight, especially to someone without a functional programming background. But indeed, it does exactly what the entire cell array B2:F6 did by replicating the formula:

- $\text{HREP}(n, x)$ creates a new two-dimensional array of size $n \times \text{COLS}(x)$ by repeating x exactly n times.
- $\text{VREP}(m, x)$ creates a new two-dimensional array of size $\text{ROWS}(x) \times m$; it works exactly like HREP but in the vertical direction.
- $\text{MAP}(f, x_1, x_2)$ combines x_1 and x_2 pointwise by applying f .

Concretely, the new expression extends the one-dimensional ranges B1:F1 and A2:A6 into two matrices of size 5×5 and combines them pointwise using the function originally written in each cell.

What have we gained from this transformation? First, we have found a generalized expression of the algorithm that was originally distributed over a number of cells, and we can use it to write a more general version of the algorithm.

Second, and more importantly, we now have an expression which describes the structure of the computation independently from its context. This is useful, as we have recovered some high-level information that we can exploit to improve performance: there is no dependency between the individual points in this combination of two matrices, or two-dimensional arrays. Hence, it is now straightforward to parallelize the computation of the result matrix.

2.4 Different Kinds of Cell Arrays

Now consider the cell array B8:F12 (lower gray cell area), which contains the following formula in R1C1 format:

$$\phi(\text{R8C2}:\text{R12C6}) := \text{MAX}(\text{R}[-1]\text{C}[-1] + \text{R}[1]\text{C}[-6], \text{R}[0]\text{C}[-1] - 2, \text{R}[-1]\text{C}[0] - 2, 0)$$

We cannot use MAP to rewrite this cell array. There is a *sequential dependency* between the cells of the cell array because the cell E10 (R10C5) depends on E9 (R9C5), D10 (R10C4) and D9 (R9C4). These cells are inside the cell array itself. We therefore call this kind of cell array *transitive*, as opposed to *intransitive* cell arrays, which can be rewritten by using MAP, as in Sect. 2.3. Hence, we need to target the second higher-order function on arrays, namely PREFIX:

$$\begin{aligned} \phi(\text{R8C2}:\text{R12C6}) := \{ & \text{PREFIX } (\lambda(x, y, z, w). \text{MAX}(y + w, x - 2, z - 2, 0), \\ & \text{R8C1}:\text{R12C1}, \\ & \text{R7C1}, \\ & \text{R8C1}:\text{R8C6}, \\ & \text{R2C2}:\text{R6C6}) \} \end{aligned}$$

Rewriting transitive cell arrays requires a bit more work: a transitive cell array could be written in either orientation (e.g. starting at the bottom right instead at the top left); and cell references in the expression might not occur in the same order as required by the semantics of PREFIX for the argument function, as we can see in our rewritten expression above. Hence, we must order the variable names correctly.

In the remainder of this paper, we formally define these properties of cell arrays and show how to rewrite them using a straightforward rewriting semantics.

3 Rewriting Cell Arrays

The overall idea of rewriting cell arrays, is to (1) rewrite the cell array's expression by systematically replacing non-absolute cell references with fresh variable names, consistently using the same variable name for multiple occurrences of the same cell reference; (2) use the fresh variable names as arguments to an anonymous function whose body is the rewritten expression; (3) infer an input range

for each replaced cell reference by looking it up at the upper left and lower right cell addresses of the array that we are rewriting; and (4) create a new expression in which we pass the anonymous function as an argument to a higher-order array function, together with the inferred input cell ranges.

For brevity, we gloss over rotated and mirrored cases of transitive cell references. Hence, we assume that all transitive references are of the form $R[-1]C[0]$, $R[-1]C[-1]$ or $R[0]C[-1]$, referring to the previous row, same column; previous row, previous column; or same row, previous column. It is straightforward to implement rules for rotated and mirrored cases via array reversal in either dimension, or both.

3.1 Cell Arrays and Transitive and Intransitive Cell References

The formal definition of intransitive and transitive cell references extends set-notation to operate on cells and cell ranges. To state that a cell reference r is inside a cell array $r_1:r_2$, we simply write $r \in r_1:r_2$. A *cell array* is a cell range $r_1:r_2$ satisfying $\forall r_i, r_j \in r_1:r_2. \phi(r_i) = \phi(r_j)$, i.e. all cells of the cell range are copy equivalent [15].

Relative cell references (first argument) are converted into absolute cell references by adding the row- and column-offset to their own location in the sheet (second argument), as defined by the function *lookup*:

$$\begin{aligned} \text{lookup}[\llbracket R[i_{r1}]C[i_{c1}], R\ i_{r2}\ C\ i_{c2} \rrbracket] &= R\ (i_{r1} + i_{r2})\ C\ (i_{c1} + i_{c2}) \\ \text{lookup}[\llbracket R\ i_{r1}\ C[i_{c1}], R\ i_{r2}\ C\ i_{c2} \rrbracket] &= R\ i_{r1}\ C\ (i_{c1} + i_{c2}) \\ \text{lookup}[\llbracket R[i_{r1}]C\ i_{c1}, R\ i_{r2}\ C\ i_{c2} \rrbracket] &= R\ (i_{r1} + i_{r2})\ C\ i_{c1} \\ \text{lookup}[\llbracket R\ i_{r1}\ C\ i_{c1}, - \rrbracket] &= R\ i_{r1}\ C\ i_{c1} \end{aligned}$$

A cell reference is intransitive if it never refers back into the cell array, no matter the location of the containing cell. We formulate this as follows:

$$\{\text{lookup}[\llbracket r, r_0 \rrbracket \mid r_0 \in r_1:r_2\} \cap r_1:r_2 = \emptyset \Rightarrow r \text{ is intransitive in } r_1:r_2.$$

Conversely, we can define transitive cell references by inverting the equation:

$$\{\text{lookup}[\llbracket r, r_0 \rrbracket \mid r_0 \in r_1:r_2\} \cap r_1:r_2 \neq \emptyset \Rightarrow r \text{ is transitive in } r_1:r_2.$$

Absolute references $RiCi$ are neither transitive nor intransitive and we treat them like constants during rewriting.

3.2 Rewriting Semantics

We use reducible expressions and a reduction relation to formalize the rewriting process. The \rightsquigarrow relation in Fig. 4 defines rewriting cell arrays from plain spreadsheet formulas to higher-order functional programs in λ -calc. More precisely, the relation \rightsquigarrow rewrites an expression u to an expression l without relative references; see Fig. 3.

$$\begin{aligned}
l &::= v \mid x \mid l \oplus l \mid \text{IF}(l, l, l) \mid F(l, \dots) \mid \text{RiCi}:\text{RiCi} \mid \text{RiCi} \\
\mathcal{L} &::= \circ \mid \mathcal{L} \oplus u \mid l \oplus \mathcal{L} \mid F(l, \dots, \mathcal{L}, u, \dots) \\
&\quad \mid \text{IF}(\mathcal{L}, u, u) \mid \text{IF}(l, \mathcal{L}, u) \mid \text{IF}(l, l, \mathcal{L}) \\
\Gamma &::= \text{more}(\underbrace{[(r, x) \dots]}_{\text{Transitive}}; \underbrace{[(r, x) \dots]}_{\text{Intransitive}}; \phi(r:r) := \mathcal{L} \mid \text{done}(\phi(r:r) := \{e\})
\end{aligned}$$

Fig. 3. Rewriting context and transformation language for λ -calc. The form l is a subset of e , with only absolute cell references.

The form Γ describes a rewriting in progress. It is either **more** with transitive cell references and their substitutions, intransitive cell references and their substitutions, a cell range, and the expression that it contains; or it is **done** with a cell range and its rewritten expression. We use (r^T, x^T) to denote a substitution pair of a transitive cell reference and (r^I, x^I) to denote a substitution pair of an intransitive cell reference.

In plain English, the rules in Fig. 4 perform the following operations:

- Rule EXIST-I replaces a cell reference r with an already existing variable x from the list of intransitive substitutions.
- Rule EXIST-T replaces a cell reference r with an already existing variable x from the list of transitive substitutions.
- Rule SUBST-I replaces an intransitive cell reference r with a fresh variable x and stores the substitution (r, x) in the list of intransitive substitutions.
- Rule SUBST-T replaces a transitive cell reference r with a fresh variable x and stores the substitution (r, x) in the list of transitive substitutions.
- Rule SYNTH-MAP takes a rewritten expression l and wraps it in a λ -expression whose variables are the variable names from the intransitive substitutions. It places the resulting function as first argument to a call to **MAP**; the remaining arguments are the substituted cell references, converted to cell ranges by performing a lookup from r_{ul} and r_{lr} for each of them and extended to match the cell array’s size. The result is an expression that can be plugged into an array formula.
- Rule SYNTH-PFX takes a rewritten expression l and wraps it in a λ -expression whose first three parameters are the variable names from the list of “sorted” transitive substitutions. The remaining parameters are taken from the intransitive substitutions, as in rule SYNTH-MAP. The rule constructs the initial row- and column-array by combining the result of the lookup of the first and last transitive reference on r_{ul} and the row, or column, of r_{lr} . The transitive cell references are converted as in rule SYNTH-MAP. The result is an expression that can be plugged into an array formula.

Both rule SYNTH-MAP and SYNTH-PFX make use of the meta-function *extd*, short for “extend”. It returns an expression that, if necessary, replicates the intransitive input arrays to match the cell array $r_1:r_2$ being rewritten:

$$\begin{aligned}
\text{extd}\llbracket r_1^I : r_2^I, r_1 : r_2 \rrbracket &= \mathbf{VREP}(n, r_1^I : r_2^I) && \text{where } n = \text{rows}\llbracket r_1 : r_2 \rrbracket, \\
&&& \text{rows}\llbracket r_1^I : r_2^I \rrbracket = 1 \\
\text{extd}\llbracket r_1^I : r_2^I, r_1 : r_2 \rrbracket &= \mathbf{HREP}(m, r_1^I : r_2^I) && \text{where } m = \text{columns}\llbracket r_1 : r_2 \rrbracket, \\
&&& \text{columns}\llbracket r_1^I : r_2^I \rrbracket = 1 \\
\text{extd}\llbracket r_1^I : r_2^I, r_1 : r_2 \rrbracket &= r_1^I : r_2^I && \text{otherwise.}
\end{aligned}$$

Finally, rule **SYNTH-PFX** uses the meta-functions *fill* and *sort*. We require that there are three transitive substitutions in the order $\mathbf{R}[0]\mathbf{C}[-1]$, $\mathbf{R}[-1]\mathbf{C}[-1]$ and $\mathbf{R}[-1]\mathbf{C}[0]$. Therefore, *fill* generates placeholder substitutions for each not

$$\begin{aligned}
&\mathbf{more}([(r^T, x^T) \dots]; [(r_1^I, x_1^I) \dots (r, x)(r_2^I, x_2^I) \dots]; \phi(r_{ul} : r_{lr}) := \mathcal{L}[r] \rightsquigarrow && [\text{EXIST-I}] \\
&\mathbf{more}([(r^T, x^T) \dots]; [(r_1^I, x_1^I) \dots (r, x)(r_2^I, x_2^I) \dots]; \phi(r_{ul} : r_{lr}) := \mathcal{L}[x]) \\
\\
&\mathbf{more}([(r_1^T, x_1^T) \dots (r, x)(r_2^T, x_2^T) \dots]; [(r^I, x^I) \dots]; \phi(r_{ul} : r_{lr}) := \mathcal{L}[r] \rightsquigarrow && [\text{EXIST-T}] \\
&\mathbf{more}([(r_1^T, x_1^T) \dots (r, x)(r_2^T, x_2^T) \dots]; [(r^I, x^I) \dots]; \phi(r_{ul} : r_{lr}) := \mathcal{L}[x]) \\
\\
&\mathbf{more}([(r^T, x^T) \dots]; [(r^I, x^I) \dots]; \phi(r_{ul} : r_{lr}) := \mathcal{L}[r] \rightsquigarrow && [\text{SUBST-I}] \\
&\mathbf{more}([(r^T, x^T) \dots]; [(r^I, x^I) \dots (r, x)]; \phi(r_{ul} : r_{lr}) := \mathcal{L}[x]) \\
&\quad \mathbf{where } r \text{ is intransitive in } r_{ul} : r_{lr} \\
&\quad x \text{ fresh} \\
\\
&\mathbf{more}([(r^T, x^T) \dots]; [(r^I, x^I) \dots]; \phi(r_{ul} : r_{lr}) := \mathcal{L}[r] \rightsquigarrow && [\text{SUBST-T}] \\
&\mathbf{more}([(r^T, x^T) \dots (r, x)]; [(r^I, x^I) \dots]; \phi(r_{ul} : r_{lr}) := \mathcal{L}[x]) \\
&\quad \mathbf{where } r \text{ is transitive in } r_{ul} : r_{lr} \\
&\quad x \text{ fresh} \\
\\
&\mathbf{more}([(r^I, x^I) \dots]; \phi(r_{ul} : r_{lr}) := l) \rightsquigarrow && [\text{SYNTH-MAP}] \\
&\mathbf{done}(\phi(r_{ul} : r_{lr}) := \{\mathbf{MAP}(\lambda(x^I, \dots).l, r_{ul}^{I+} : r_{lr}^{I+}, \dots)\}) \\
&\quad \mathbf{where } [(r^I, x^I) \dots] \text{ is non-empty} \\
&\quad r_{ul}^I \dots = \text{lookup}\llbracket r^I, r_{ul} \rrbracket \dots \\
&\quad r_{lr}^I \dots = \text{lookup}\llbracket r^I, r_{lr} \rrbracket \dots \\
&\quad r_{ul}^{I+} : r_{lr}^{I+} \dots = \text{extd}\llbracket r_{ul}^I : r_{lr}^I, r_{ul} : r_{lr} \rrbracket \dots \\
\\
&\mathbf{more}([(r^T, x^T) \dots]; [(r^I, x^I) \dots]; \phi(r_{ul} : r_{lr}) := l) \rightsquigarrow && [\text{SYNTH-PFX}] \\
&\mathbf{done}(\phi(r_{ul} : r_{lr}) := \{\mathbf{PREFIX}(\lambda(x_1^T, x_2^T, x_3^T, x^I, \dots).l, \\
&\quad r_{c0} : r_{c1}, r_d, r_{r0} : r_{r1}, r_{ul}^{I+} : r_{lr}^{I+}, \dots)\}) \\
&\quad \mathbf{where } [(r^T, x^T) \dots] \text{ is non-empty} \\
&\quad r_{ul}^I \dots = \text{lookup}\llbracket r^I, r_{ul} \rrbracket \dots \\
&\quad r_{lr}^I \dots = \text{lookup}\llbracket r^I, r_{lr} \rrbracket \dots \\
&\quad r_{ul}^{I+} : r_{lr}^{I+} \dots = \text{extd}\llbracket r_{ul}^I : r_{lr}^I, r_{ul} : r_{lr} \rrbracket \dots \\
&\quad (r_1^T, x_1^T), (r_2^T, x_2^T), (r_3^T, x_3^T) = \text{sort}\llbracket \text{fill}\llbracket [(r^T, x^T) \dots] \rrbracket \rrbracket \\
&\quad r_d = \text{lookup}\llbracket r_2^T, r_{ul} \rrbracket \\
&\quad r_{c0} = \text{lookup}\llbracket r_1^T, r_{ul} \rrbracket \\
&\quad r_{r0} = \text{lookup}\llbracket r_3^T, r_{ul} \rrbracket \\
&\quad r_{c1} = \mathbf{R}(\text{row}\llbracket r_{lr} \rrbracket)\mathbf{C}(\text{column}\llbracket r_{c0} \rrbracket) \\
&\quad r_{r1} = \mathbf{R}(\text{row}\llbracket r_{r0} \rrbracket)\mathbf{C}(\text{column}\llbracket r_{lr} \rrbracket)
\end{aligned}$$

Fig. 4. The \rightsquigarrow relation for rewriting cell array formulas into λ -calc. The rules are explained in detail in Sect. 3.2.

encountered transitive reference; *sort* sorts the three substitutions after their respective references, as described above.

3.3 Preemptive Cycle Detection

Rewriting cell arrays to array formulas changes the dependency structure of the spreadsheet: where before a cell of the cell array may only have depended on a single cell the of input range, it now depends on the entire range. The rewritten cell has become part of an unpacked array, whose formula explicitly references the aforementioned range. It is easy to come up with an example that would lead to the creation of cyclic dependencies if rewritten. We require two or more cell arrays that refer to cells of each other. Rewriting the contrived spreadsheet shown in Fig. 5 leads to the creation of cyclic dependencies.

	A	B	
1	=B1	1	
2	=B2	=A1+B1	↔
3	=B3	=A2+B2	

	A	B
1	= $\{\text{MAP}(\lambda(x).x, B1:B3)\}$	1
2	= $\{\text{MAP}(\lambda(x).x, B1:B3)\}$	= $\{\text{PREFIX}(\lambda(x,y,z).x+y, A2:A3, A1, B1:B1)\}$
3	= $\{\text{MAP}(\lambda(x).x, B1:B3)\}$	= $\{\text{PREFIX}(\lambda(x,y,z).x+y, A2:A3, A1, B1:B1)\}$

Fig. 5. A spreadsheet (top) whose rewritten variant (bottom) contains cyclic dependencies. The cell arrays A1:A3 and B1:B3 are not copy equivalent. Rewriting both results in an explicit cyclic dependency between the array formulas: $\phi(A1:A3)$ refers to B1:B3 and $\phi(B2:B3)$ refers to A2:A3.

To avoid this, we perform a preemptive detection of cyclic references. We walk the reference graph from each intransitive cell reference and each cell from the initial row and column, and check that we never arrive at a cell that is part of the cell array. We use a depth-first search without repetition to detect possible cyclic references. If we detect one, we do not rewrite the cell array.

3.4 Correctness

We do not currently have a formal proof of correctness for our rewriting semantics. However, the slightly informal semantics in Sect. 2 for **MAP** and **PREFIX** are carefully chosen to capture the semantics of the original cell array structure, so we believe that our rewriting semantics are correct. The proof would require a formal semantics for spreadsheet recalculation and functions on arrays, which is beyond the scope of this paper.

With a formal semantics, we believe that one can show that rewritten cell arrays are observationally equivalent to the original formulas for cell arrays with and without transitive cell references and hence prove that the rewriting semantics is correct. More formally, if

$$\text{more}([\]; [\]; \phi(r_1:r_2) := u) \rightsquigarrow \text{done}(\phi(r_1:r_2) := \{e\})$$

and $\phi(r_1:r_2) := u$ evaluates to v , then we want to prove that $\phi(r_1:r_2) := \{e\}$ also evaluates to v .

4 Implementation

We have implemented the rewriting semantics from Sect. 3 in Funcalc [17], a prototype spreadsheet engine with efficient sheet-defined functions. The formula language in Funcalc is higher-order. We use a modified variant of Funcalc, where bulk operations on arrays are executed in parallel.

Instead of writing our own detection of cell arrays, we piggyback on Funcalc’s algorithm for rebuilding the support graph [17, Sect. 4.2.9], which runs in linear time in the number of cells in the cell array.

4.1 Parallelization Strategies

Since Funcalc runs on the .Net platform, we use the parallelization mechanisms from the Task Parallel Library [14]. We can parallelize MAP by iterating over either rows or columns in a parallel for-loop. Parallelizing the PREFIX function is slightly more complicated.

Recall from Sect. 2 that, in order to compute the value at $X[i, j]$ we must already have computed $X[i, j - 1]$, $X[i - 1, j]$ and $X[i - 1, j - 1]$. Hence, there exists a sequential dependency between the computations.

Figure 6 illustrates the order in which PREFIX processes parts of the argument array. Even though both q_2 and q_3 depend on q_1 , there is no sequential dependency between q_2 and q_3 . We can therefore compute the prefix of q_2 and q_3 in parallel. When both are computed, we can proceed to compute q_4 . We use this parallelization scheme recursively on each sub-array and stop recursing as soon as either a minimum size is reached or if we have spawned as many parallel tasks as there are processors.

4.2 Handling Over-Generalization

We can describe relative references in terms of their *stride*:

$$\text{stride}[\mathbb{R}[i_1]\mathbb{C}[i_2]] = \max(|i_1|, |i_2|)$$

In real-world spreadsheets, it may happen that a transitive reference has a stride larger than one, but the PREFIX function and its variants do not generalize to

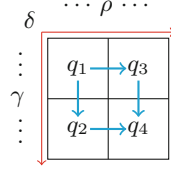


Fig. 6. Wave front scheme of the **PREFIX** function. We process from the top left to the bottom right of the 2D array, as indicated by the red arrows. The quadrants q_2 and q_3 depend on q_1 , while q_4 depends on all of these, as indicated by the blue arrows. Values γ , δ and ρ are initial values at the fringes, as described in Sect. 2

such references. Hence, we cannot directly rewrite cell arrays with transitive cell references of a stride larger than one.

Strides larger than one seem to be artifacts of the generality of the support graph rebuilding algorithm (see Sect. 4). Our key observation here is that one can turn transitive cell references into intransitive cell references by splitting up the cell array into two sub-arrays. Consider the cell array $R5C1:R15C5$ whose expression contains the transitive cell reference $R[-5]C[0]$. We can split it up into the two sub-arrays $R5C1:R10C5$ and $R11C1:R15C5$, in both of which the reference $R[-5]C[0]$ is intransitive.

We call the rewriting algorithm recursively on each of the sub-arrays until we either end up with a cell array that has transitive cell references with stride at most one, or until there is only a single cell left, in which case we abort.

5 Performance Evaluation

To demonstrate the feasibility of our technique, we have conducted performance benchmarks on synthetic and real-world spreadsheets. To avoid the overhead of excess parallelism, we impose a minimum of 64 cells per cell array on the rewriting algorithm, such that smaller cell arrays will not be rewritten. Times for rewriting are not included in the measurements, since we consider this a one-time operation. For comparison, we also benchmark performance for naively launching a parallel task per cell.

Our benchmarks are the average of 100 full recalculations of the entire spreadsheet. Full recalculation is easier to control during automatic benchmarks, but does not reflect how rewriting cell arrays may affect the dependency structure of spreadsheets negatively for efficient minimal recalculation.

Funcalc runs on the .Net platform. To trigger JIT compilation, we run three warm-up iterations which we do not count prior to benchmarking. Our test machines are an Intel i7-6500U with four cores at 2.5 GHz and 32 GB of RAM, 64 bit Windows 7 and .Net Framework 4.7, as well as an Intel Xeon E5-2680 v3 with 48 cores at 2.5 GHz and 32 GB of RAM, 64 bit Windows 10 and .Net Framework 4.6.2. We only use 32 cores on the Xeon.

5.1 Spreadsheet Selection

We use two contrived, idealized spreadsheets to measure the isolated effect of rewriting transitive and intransitive cell arrays. Both contain one cell array of size 100×100 . The first one contains an intransitive cell array that applies the sinus function on each input cell. The second one computes a cell array’s prefix sum using transitive cell references and then calls the sinus function on the result of each cell.

Furthermore, we have chosen three spreadsheets from Filby’s [7] book from the EUSES corpus [8], as well as three Funcalc-related spreadsheets for synthetic benchmarks. All of these sheets contain large cell arrays.

Finally, we use real-world spreadsheets from the EUSES spreadsheet corpus [8]. We have selected twelve spreadsheets with relatively large and relatively many cell arrays. Selection criteria were (1) applicability of our rewriting technique and (2) effort required to make the spreadsheets compatible with Funcalc. Funcalc syntax differs from Excel in a number of ways, which requires modifications to the sheets. Additionally, we have implemented some Excel and VBA functions as sheet-defined functions¹.

5.2 Results

Table 1 shows speedup after rewriting idealized spreadsheets with only intransitive or only transitive cell references. On the i7, we achieve good parallel speedup for intransitive cell arrays; on the Xeon, parallelism doesn’t scale. The very large speedup for transitive cell arrays is likely due to (1) using a more specialized machinery to refer to values in other cells; and (2) that Funcalc compiles the functions we synthesize to byte-code, which alleviates the overhead of interpreting the expression in each cell, as during Funcalc’s “standard” recalculation.

Table 1. Average speedup and standard deviation for 100 recalculations of idealized spreadsheets that only consist of either an intransitive or transitive cell array of size 100×100 . Speedup is relative to sequential recalculation on the same machine; higher is better.

	Intel i7	Intel Xeon
Intransitive	2.77 ± 0.317	3.14 ± 0.059
Transitive	11.26 ± 0.881	10.3 ± 0.655

Figure 7 shows the speedup after rewriting the more realistic spreadsheets. On the i7, we achieve good speedups for synthetic spreadsheets. Running on the Xeon with eight times as many cores does not improve performance. On both machines,

¹ The Funcalc compatible spreadsheets from the EUSES corpus are available at <https://github.com/popular-parallel-programming/funcalc-euses/>.

the average speedup for real-world spreadsheets is lower than we would expect, given the numbers from Table 1. We have two explanations for this.

First, the achievable speedup is bound by Amdahl’s law [9, Sect. 1.5]. If a spreadsheet contains 4500 cells with formulas and a single intransitive cell array of size 500, then the maximum speedup factor we can expect to see on 32 cores is roughly 1.26. This holds for both synthetic and real-world spreadsheets. Unless rewriteable cell arrays either dominate the spreadsheet, as in `financial.xml` and `PLANCK.xml`, or contain very costly computations, as in `testsdf.xml`, the overall performance will still be determined by the sequential computations.

Secondly, real-world spreadsheets have undergone continuous development and are often cluttered with small experiments. Their design is often less streamlined towards a single large computation than that of synthetic spreadsheets. Even if there are lots of disjoint computations, our technique is unable to exploit these unless they are structured in an array-like fashion.

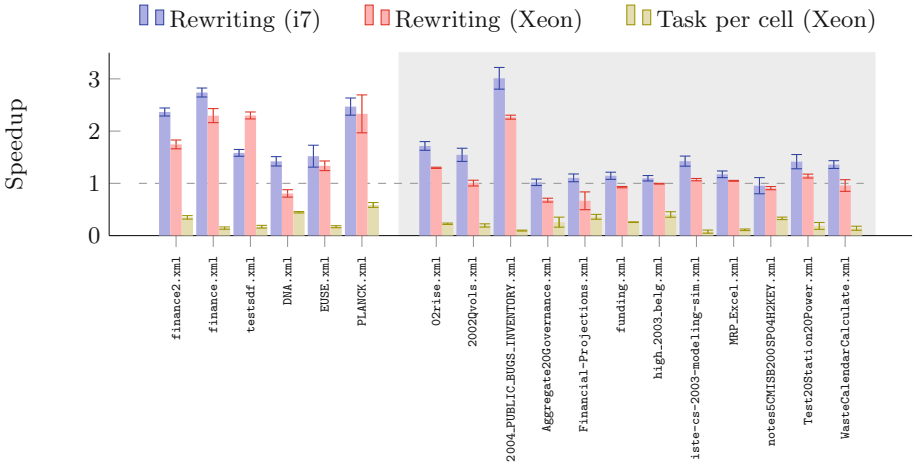


Fig. 7. Average benchmark results over 100 runs for synthetic (left part) and real-world (right part on gray background) spreadsheets. Values are speedup factors over sequential performance on the same machine; higher is better. Error bars indicate the standard deviation.

6 Alternative Usages and Related Work

Neither parallel recalculation of spreadsheets nor high-level structure analysis are new ideas. To our knowledge, however, no prior work has combined both in a practical application of functional programming.

Wack [19] focused on a dataflow approach to whole-spreadsheet parallelization, in contrast to our idea that harnesses local array parallelism. Yoder and Cohn [20, 21] investigate spreadsheets from a theoretical point of view, also with

data flow parallelism in mind. They observe that high-level array programming intuitively maps to spreadsheets [21]; this is the core of our technique.

Much research on high-level spreadsheet structures focuses on user understanding; either by highlighting areas with equal or similar formulas [15], whose definition is highly related to cell arrays, or by drawing dataflow diagrams [11] to illustrate relations between sheets and cell arrays. Our rewriting technique could be adapted to give such a high-level overview over operations on cell arrays by displaying the synthesized function.

Rewriting of cell arrays is related to template synthesis from spreadsheets. Isakowitz et al. [13] describe a method to synthesize either a model from a spreadsheet or instantiate a spreadsheet from a model. The notable difference to our work is that they generate a whole-sheet model. Furthermore, they use an external language to describe the model, whereas we perform source-to-source rewriting. Generating local high-level abstractions, as opposed to whole-sheet models, could be useful for expert spreadsheet developers when devising algorithms, similar to spreadsheet generation.

Abraham and Erwig [1] infer templates by analyzing references across cell arrays to prevent errors during modification, also using copy equivalence. Our technique is only concerned with single cell arrays.

Others [3, 5, 12] focus on detecting clones of cell arrays or tables on the same spreadsheet, which is again a whole-sheet analysis.

7 Conclusion

In this paper, we presented a rewriting semantics to rewrite cell arrays that consist of copy equivalent cells to higher-order functional expressions on arrays. We can easily exploit the implicit parallelism of these rewritten cell arrays and therefore improve recalculation speed of spreadsheets where cell arrays dominate on typical consumer hardware.

There are limitations to our approach. Our rewriting semantics currently does not support cell arrays that reference cell ranges. We believe that this will be easy to add. We have furthermore not yet presented a formal proof that our rewriting semantics preserves the semantics of the cell array’s expression.

Naively rewriting all detectable cell arrays can introduce cyclic references and hence change the semantics of the original spreadsheet. Detecting these before rewriting comes at the cost of an additional walk of the dependency graph. Moreover, the parallel speedup we can achieve is limited by the ratio of parallelizable cell arrays to inherently sequential dependencies in the spreadsheet.

Our experimental results show that only spreadsheets consisting of large cell arrays achieve good speedups on consumer hardware. This suggests that our rewriting approach should not be automatic but instead a manual tool for expert spreadsheet developers, and also that it makes sense to investigate how our technique can be combined with other parallelization techniques, for instance data flow parallelism.

References

1. Abraham, R., Erwig, M.: Inferring templates from spreadsheets. In: ICSE 2006
2. Casimir, R.J.: Real programmers don't use spreadsheets. ACM SIGPLAN Not. Homepage Arch. **27**(6), 10–16 (1992)
3. Cheung, S.C., Chen, W., Liu, Y., Xu, C.: CUSTODES: automatic spreadsheet cell clustering and smell detection using strong and weak features. In: ICSE 2016
4. Dou, W., Cheung, S.C., Wei, J.: Is spreadsheet ambiguity harmful? Detecting and repairing spreadsheet smells due to ambiguous computation. In: ICSE 2014
5. Dou, W., Cheung, S.C., Gao, C., Xu, C., Xu, L., Wei, J.: Detecting table clones and smells in spreadsheets. In: FSE 2016
6. Dou, W., Xu, C., Cheung, S.C., Wei, J.: CACheck: detecting and repairing cell arrays in spreadsheets. IEEE Trans. Softw. Eng. **43**(3), 226–251 (2016)
7. Filby, G. (ed.): Spreadsheets in Science and Engineering. Springer, New York (1998). <https://doi.org/10.1007/978-3-642-80249-2>
8. Fisher, M., Rothermel, G.: The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. In: WEUSE I
9. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier/Morgan Kaufmann, Amsterdam (2008). ISBN 9780123705914
10. Hermans, F., Murphy-Hill, E.: Enron's spreadsheets and related emails: a dataset and analysis. In: ICSE 2015
11. Hermans, F., Pinzger, M., van Deursen, A.: Supporting professional spreadsheet users by generating leveled dataflow diagrams. In: ICSE 2011
12. Hermans, F., Sedee, B., Pinzger, M., van Deursen, A.: Data clone detection and visualization in spreadsheets. In: ICSE 2013
13. Isakowitz, T., Schocken, S., Lucas, H.C.: Toward a logical/physical theory of spreadsheet modeling. ACM Trans. Inf. Syst. **13**(1), 1–37 (1995)
14. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: OOPSLA 2009
15. Mittermeir, R., Clermont, M.: Finding high-level structures in spreadsheet programs. In: WCRE 2002
16. Sajaniemi, J.: Modeling spreadsheet audit: a rigorous approach to automatic visualization. J. Vis. Lang. Comput. **11**(1), 49–82 (2000)
17. Sestoft, P.: Spreadsheet Implementation Technology: Basics and Extensions. The MIT Press, Cambridge (2014). ISBN 0262526646
18. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
19. Wack, A.P.: Partitioning dependency graphs for concurrent execution: a parallel spreadsheet on a realistically modeled message passing environment. Ph.D. thesis, Newark, DE, USA (1996)
20. Yoder, A., Cohn, D.L.: Observations on spreadsheet languages, intension and dataflow. Technical report
21. Yoder, A.G., Cohn, D.L.: Domain-specific and general-purpose aspects of spreadsheet language. In: DSL 1997

Practical Aspects of Declarative Languages

20th International Symposium, PADL 2018, Los Angeles,

CA, USA, January 8–9, 2018, Proceedings

Calimeri, F.; Hamlen, K.W.; Leone, N. (Eds.)

2018, XIV, 203 p. 56 illus., Softcover

ISBN: 978-3-319-73304-3