

Data-Parallel Spreadsheet Programming

Florian Biermann

Advisor: Peter Sestoft
Submitted: June 2018

IT UNIVERSITY OF COPENHAGEN

Abstract

Spreadsheets are used in industry and academia across all domains and their application ranges from simple accounting to sequencing of amino acids. Some businesses re-use and extend spreadsheets over years and not only complexity but also performance becomes a problem.

In the last two decades, parallel shared-memory multiprocessor computers have become abundant, but they remain notoriously difficult to program correctly. Therefore, it is often not possible for end-users to make full use of their parallel hardware.

Declarative, functional programming languages are an alternative to imperative programming languages that makes programming shared-memory multiprocessors much easier: due to the absence of side effects in these languages, compilers and libraries can parallelize programs automatically. The formula language of spreadsheets is a declarative, first-order functional language and therefore potentially allows for an automatic parallelization of spreadsheet calculations.

This thesis explores the design space of automatically parallelizing spreadsheet recalculations. Often times, computations in spreadsheets are structured in a way that corresponds to declarative high-level programming on arrays. Therefore, the focus of this thesis lies on practical data-parallelism in a spreadsheet model of computations. This thesis makes the following contributions:

- we show how to automatically re-write high-level spreadsheet structures into higher-order functional programs for parallel execution;
- we contrast and combine our re-writing technique with a recalculation algorithm that dynamically exploits dataflow parallelism in spreadsheets;
- we describe a representation of two-dimensional arrays that pragmatically caters to the needs of high-level array programming; and
- we explore a hypothetical approach to array fusion and laziness in a purely functional, strict spreadsheet language.

Resumé

Regneark bruges i forretning og forskning bland alle domæner og deres anvendelse rækker fra enkle regnskaber til sekventering af aminosyrer. Nogle virksomheder genbruger og udvider regneark over mange år, og ikke blot kompleksiteten, men også hastigheden af beregningerne kan blive til et problem.

I de sidste tyve år er processorer med mange regnekerner og delt hukommelse blevet udbredt overalt. Det er dog fortsat svært at programmere disse maskiner korrekt. Det er derfor ofte ikke muligt for slutbrugere at udnytte deres parallelle udstyr.

Deklarativ funktionsprogrammering er et alternativ til imperative programmeringssprog og gør programmering af regnekerner med delt hukommelse meget nemmere: takket være fraværet af sideeffekter i disse sprog kan oversættere og biblioteker automatisk parallelisere programmer. Formelsproget i regneark er et deklarativt første-ordens funktionssprog og har derfor potentiale for automatisk parallelisering af beregninger i en regneark.

Denne afhandling udforsker designrummet for automatisk parallelisering af regnearksberegninger. Det sker ofte at beregningerne i en regneark er struktureret på en måde, som svarer til deklarativ højere-ordens programmering på matricer. Derfor ligger vores fokus i denne afhandling på praktisk anvendelig data-parallelisme i regneark. Denne afhandling indeholder følgende bidrag:

- vi viser hvordan visse strukturer i regneark kan omskrives til højere-ordens funktionsprogrammer som kan eksekveres i parallelt;
- vi sammenligner og derefter kombinerer vi vores omskrivningsteknik med en genberegningssalgoritme som dynamisk udnytter datastrømparallelisme i regneark;
- vi beskriver en repræsentation af matricer af to dimensioner, som ganske pragmatisk imødekommer matrixprogrammering på et abstrakt niveau; og til sidst
- udforsker vi en hypotetisk tilgang til en teknik til elimination af mellemresultater på matricer og dovenskab i et funktionsprogrammeringssprog i regneark uden sideeffekter.

Zusammenfassung

Kalkulationstabellen, im allgemeinen Sprachgebrauch oft "Excel Tabellen", werden in allen Disziplinen, sowohl in der Industrie als auch an der Akademie, genutzt, und ihre Nutzung reicht von einfacher Buchhaltung bis zur Sequenzierung von Aminosäuren. Manche Firmen benutzen und erweitern Kalkulationstabellen über Jahre hinweg und nicht nur deren Komplexität, sondern auch die Geschwindigkeit der Berechnungen wird problematisch.

In den letzten beiden Jahrzehnten sind Multiprozessoren mit gemeinsam genutztem Speicher allgegenwärtig geworden, aber ihre korrekte Programmierung ist weiterhin notorisch schwierig. Deswegen ist es für Endnutzer oft nicht möglich ihre parallele Computerhardware vollständig auszunutzen.

Deklarative, funktionale Programmiersprachen bilden eine Alternative zu imperativen Programmiersprachen und machen es einfacher, Multiprozessoren mit gemeinsam genutztem Speicher zu programmieren: dank der Abwesenheit von Nebeneffekten können Compiler und Bibliotheken Programme automatisch parallelisieren. Die Formelsprache in Kalkulationstabellen ist eine deklarative, funktionale Programmiersprache erster Ordnung und erlaubt deswegen potentiell, Tabellenkalkulationen zu parallelisieren.

Diese Dissertation erforscht den Gestaltungsraum der automatischen Parallelisierung von Tabellenkalkulationen. Die Struktur von Kalkulationstabellen korreliert oft mit deklarativen, höheren Programmkonstrukten auf Matrizen. Deswegen konzentrieren wir uns auf praktisch anwendbarer Datenparallelität in Tabellenkalkulationen. Diese Dissertation macht die folgenden Beiträge:

- wir zeigen, wie höhere Strukturen in Kalkulationstabellen automatisch in höhere funktionale Programme umgeschrieben werden können um deren Berechnung zu parallelisieren;
- wir stellen unserer Umschreibetechnik einen Algorithmus zur Neuberechnung von Kalkulationstabellen gegenüber, der dynamisch deren parallelen Datenfluss ausnutzt;
- wir beschreiben eine Darstellung von Matrizen die pragmatisch die Anforderungen von höherem Matrizenprogrammieren erfüllt; und schließlich
- erkunden wir eine hypothetische Herangehensweise zur Eliminierung von Zwischenergebnissen auf Matrizen und deren Bedarfsauswertung in einer rein funktionalen, strikten Tabellenkalkulationssprache.

Acknowledgements

First of all, I would like to thank my supervisor Peter Sestoft: for giving me the chance to pursue a PhD in the first place and for his continuous support and guidance. Peter has greatly influenced how I think about technical and scientific problems.

Thanks to Peter Eklund and John Paulin Hansen for keeping me at the IT University long enough until the opportunity for this PhD arose.

Thanks to my co-authors Alexander Asp Bock, Wensheng Dou and Peter Sestoft for interesting technical discussions and a fruitful collaboration.

Thanks to Wensheng for hosting me during my second stay at the Chinese Academy of Sciences in 2017. Thanks to Zhang Yu and Prof. Huimin Lin for hosting me during my first stay at CAS in 2016.

Thanks to our student programmers Holger Stadel Borum and Malthe Ettrup Kirkbro for continuously improving Funcalc over the last two years and thereby making some of my work possible in the first place. Also thanks to the former and current members of the SQUARE group for providing a platform for discussions and feedback.

Thanks to Paolo Tell for continuously mentoring me on academic matters over the last four years while being the only reliable source of high-quality coffee in the 4D wing.

Thanks to Peter Sestoft, Dino Ott and Johan von Tangen Sivertsen, who all gave valuable feedback that greatly improved the quality of this dissertation.

Thanks the Sino-Danish Center for Education and Research for supporting my PhD financially.

Thanks to my friends and family and thanks to everyone who has supported me and in some way or another has contributed to me finishing this project.

Line: danke!

Contents

Contents	vii
1 Introduction	1
1.1 Contributions	2
1.2 Thesis Outline	3
1.3 Experiments	5
I Background	7
2 Spreadsheets	9
2.1 Basic Concepts	9
2.2 Formal Semantics of Spreadsheets	12
2.3 Funcalc and Sheet-Defined Functions	18
2.4 Related Work	20
3 Declarative Array Programming	25
3.1 Array Combinators	26
3.2 Array Programming in Funcalc	30
3.3 Related Work	37
II Contributions	43
4 Rewriting Cell Arrays	45
4.1 Introduction	45
4.2 From Spreadsheets to Higher-Order Functional Programs	46
4.3 Systematically Rewriting Cell Arrays	51
4.4 Implementation	60

4.5	Performance	62
4.6	Alternative Usages and Related Work	68
4.7	Conclusion	69
5	Spreadsheet Dataflow Parallelism	71
5.1	Introduction	71
5.2	Funcalc: Sequential Implementation	72
5.3	Puncalc: Parallel Implementation	74
5.4	Results and Validation	87
5.5	Related Work	92
5.6	Combining Dataflow with Cell Array Rewriting	93
5.7	Conclusion	96
6	End-User Array Programming	97
6.1	Introduction	97
6.2	Quad Rope Semantics	100
6.3	Block-Sparseness	107
6.4	Two-Way Nodes vs. Four-Way Nodes	108
6.5	A Reference Implementation	110
6.6	Performance	117
6.7	Quad Ropes in Funcalc	121
6.8	Conclusion	125
7	Laziness and Deforestation in Spreadsheets	129
7.1	Introduction	129
7.2	An Algebra of Array Combinators	130
7.3	Funky Quad Ropes	134
7.4	Initial Performance Evaluation	139
7.5	Limitations	140
7.6	Related Work	142
7.7	Conclusion	143
8	Conclusion	145
	Bibliography	149
A	Funcalc Source Code	167
B	PLT Redex Model of Cell Array Rewriting Semantics	169

Contents	ix
C Quad Rope Reference Implementation	177
D Source Code for Quad Rope Fusion	179

Chapter 1

Introduction

Among programmers and other IT professionals, it is well-known that “real programmers don’t use spreadsheets” [26]. But everybody else does! Scaffidi [91] estimates that 72 million American knowledge workers use spreadsheets at least monthly.

The two-dimensional layout and the formula language of spreadsheets seem to be easily understood by non-programmers and professional software developers alike. Spreadsheets are versatile: they pose as data storage, are used for ad-hoc computations and experiments and organizations often rely on spreadsheets to perform computations on which they base business-critical decisions—for example to assess the risk of giving a bank loan to a particular customer.

Spreadsheet programs can be slow. Swidan et al. [103] report on a case study of refactoring a large spreadsheet that originally would have taken ten hours or more to recompute when edited. The refactored spreadsheet is highly parallel and runs on a high-performance cluster. However, refactoring business critical spreadsheets comes at a high risk: often, lack of documentation and absence of testing tools make it simply unfeasible and it is expensive to hire experts to perform manual refactoring and parallelization.

Is it possible to enable spreadsheet end-users to run their spreadsheets on parallel hardware without adding this engineering overhead?

Nowadays, multi-core processors are abundant. Commodity computers are equipped with shared-memory multiprocessors, but programming them correctly remains notoriously difficult. One model of computation that relieves the burden of shared-memory multicore programming is *declarative functional programming*, where programmers describe *what* the

algorithm should compute instead of specifying *how* it should work [4]. Spreadsheets *are* declarative, first-order functional programs [26] and concepts from the array-oriented programming language APL [65] directly map to the spreadsheet paradigm [112]. These concepts allow for an *implicit* parallelization of operations that requires no effort from the programmer.

This thesis explores how well concepts from parallel functional programming apply to the spreadsheet paradigm in practice to allow end-users to write fast, parallel programs without any additional engineering overhead. Our overall conceptual contribution is the systematic application of well-known techniques from declarative data-parallel array programming to a spreadsheet model of computation and the experimental evaluation of their performance in spreadsheet programs.

Our focus is performance: we target large spreadsheets containing lots of data and complex models. We validate our approach experimentally, using both artificial and real-world spreadsheets. The techniques described in this thesis are all implemented in Funcalc [93], an experimental spreadsheet engine for end-user programming.

1.1 Contributions

1.1.1 Published Work

This thesis builds on the work presented in the following peer-reviewed articles (in order of publication date):

Quad Ropes: Immutable, Declarative Arrays With Parallelizable

Operations. Joint work with Peter Sestoft [13], presented at ARRAY'17. This paper describes the quad rope data structure, a high-level representation of two-dimensional arrays that pragmatically caters to the needs of end-user array programming. Chapter 6.

Rewriting High-Level Spreadsheet Structures into Higher-Order

Functional Programs. Joint work with Wensheng Dou and Peter Sestoft [14], presented at PADL'18. This paper describes a rewriting semantics for extracting higher-order array expressions from high-level spreadsheet structures for parallel execution. Chapter 4.

Puncalc: Task-Based Parallelism and Speculative Reevaluation

in Spreadsheets. Joint work with Alexander Asp Bock [12], to be

presented at HLPP'18. This paper describes a parallel evaluation strategy for spreadsheets that dynamically extracts parallelism without prior analysis. Chapter 5.

Additionally, this thesis uses material from the following informally published text:

Declarative Parallel Programming in Spreadsheet End-User

Development: A Literature Review. Technical report [11]. A literature review that surveys literature on array programming techniques and spreadsheet research. Section 2.4 and 3.3.

1.1.2 New Contributions

This thesis makes the following new contributions and updates to published work:

- An extension to the cell array rewriting semantics to handle cell range expressions; Sec. 4.3.5.
- A clear distinction between the formalism and the actual Funcalc implementation of cell array rewriting; Sec. 4.4.1.
- An extension of the performance evaluation of cell array rewriting; Sec. 4.5.
- A combination of cell array rewriting with dynamic dataflow parallelism and performance evaluation; Sec. 5.6.
- A discussion of the implementation and performance of quad ropes in Funcalc; Sec. 6.7
- A method for eliminating intermediate arrays in a spreadsheet model of computations, combining laziness and deforestation; Chapter 7.

1.2 Thesis Outline

This section gives a high-level overview over the structure and contributions of the dissertation.

In Part I, we provide the reader with relevant background information and define appropriate terminology that will be used throughout the remainder of the dissertation.

In Chapter 2, we give background on spreadsheet technology, “standard” spreadsheet features, a “standard” formula language and its semantics as well as an overview over related work. In particular, we introduce the Funcalc spreadsheet engine [93] in which most of the techniques presented in this thesis are implemented. We do not discuss user interfaces or any other visual features of spreadsheet systems.

Chapter 3 provides background information and related work on declarative array programming. That is, we discuss the most commonly used higher-order functions on arrays, how they can be parallelized and how they fit into a spreadsheet model of computations.

In Part II of the dissertation, we present and discuss the contributions made over the course of the PhD project.

In Chapter 4, we describe how we can directly apply techniques from array programming to structured formulas in spreadsheets via *cell array rewriting*. Spreadsheet models usually have some high-level structure that can be exploited to improve performance by running independent computations in parallel. In this chapter, we show how to map high-level spreadsheet structure to data-parallel functions on arrays such that we can compute parts of a spreadsheet in parallel without changing the recalculation algorithm of the spreadsheet engine itself.

Chapter 5 describes the implementation of a spreadsheet recalculation engine that dynamically exploits available parallelism on the fly without prior analysis. Moreover, the algorithm that we present maintains the non-strict semantics of cyclic references in spreadsheets without prior static analysis. We combine this dynamic approach with statically rewriting cell arrays and show that there are some limitations when combining the two approaches.

In Chapter 6, we focus on more traditional forms of array programming, namely the declarative specification of algorithms on arrays. We describe the *quad rope* data structure, a representation of immutable two-dimensional arrays that avoids many of the performance pitfalls of plain C-style two-dimensional arrays. Our motivation is that, for end-user development in high-level declarative programming languages, it is impractical to let users choose between different array-like data structures. Instead, end-users should use the same, somewhat performance-robust, representation for every programming task.

Chapter 7 treats a hypothetical optimization of array programming in a spreadsheet formula language. We describe a technique to perform fusion of operations on arrays across independent spreadsheet cells, such that intermediate arrays only materialize when the user implicitly requires them to.

Finally, in Chapter 8 we summarize the results of this dissertation and give an outlook on future work and open problems.

1.3 Experiments

Throughout this thesis, we report results from running benchmark experiments to verify our approach.

For the majority of these experiments, we have used an Intel Xeon E5-2680 v3 with 24 physical cores, 48 virtualized, at 2.5GHz and 32GB memory, running 64 bit Windows 10 and .NET Framework 4.7.1. The Intel Xeon consists of two chips of twelve processors each; there is no shared cache between the two chips and they communicate via main memory. We call this machine the *P3 server*. We have used the P3 server in this configuration for all experiments throughout this dissertation unless noted otherwise.

Moreover, we have compiled all .NET code for that we report performance results to 64 bit.

Part I

Background

Chapter 2

Spreadsheets

This chapter provides background information on spreadsheet technology that is sufficient for understanding the main part of the dissertation and introduces relevant definitions and terminology.

2.1 Basic Concepts

A spreadsheet is a two-dimensional grid of *cells* whose rows are labeled numerically starting from 1 and whose columns are labeled alphabetically starting from A. The uppermost leftmost cell in the spreadsheet is denoted A1, the cell to the right of it B1 and the cell below it A2. In modern spreadsheet software, multiple spreadsheets are usually organized in so-called workbooks. Each spreadsheet in a workbook has a unique name.

A cell can contain a value constant, such as a string, e.g. "spreadsheets", a number, e.g. 42.0, or an error such as #VALUE!; or a *formula expression*, indicated by a leading equals sign, e.g. =1 + 2. A formula may evaluate either to a proper value or to an error and the user can switch between displaying either values or formulas on a spreadsheet. A formula can reference one or more cells by naming their *address*, e.g. =A1 * 2. The resulting value of the cell that contains this formula will depend on the expression in cell A1. A formula can also refer to a *cell range* or *cell area* by naming two opposing corners of a rectangle using the : operator. For example, the formula =SUM(A1:A10) computes the sum of all values in the first ten rows of column A. Note that cell areas are always rectangular. A formula may also refer to cells on other spreadsheets in the same workbook.

	A	B	C	D	E
1	Year	Revenue	Expenses	Result	Total Earnings
2	1992	45	23	22	22
3	1993	1644	412	1232	1254
4	1994	12081	8041	4040	5294

	A	B	C	D	E
1	Year	Revenue	Expenses	Result	Total Earnings
2	1992	45	23	=B2 - C2	=SUM(\$D\$2:D2)
3	1993	1644	412	=B3 - C3	=SUM(\$D\$2:D3)
4	1994	12081	8041	=B4 - C4	=SUM(\$D\$2:D4)

Figure 2.1: A spreadsheet for accounting of a small business. The top sheet shows the computed results; the lower sheet shows the underlying formulas.

A cell references may be part of a reference cycle. It is legal to define the formula for cell C2 as `=C2 * 2`. Spreadsheet software will in this case usually notify the user of the presence of a cyclic dependency and abort recalculation (see Sec. 2.2.1). Moreover, spreadsheet formulas are dynamically typed. If a function is called with the wrong argument type, e.g. `=C2 * "two"`, the cell will evaluate to an error value indicating this, e.g. `#VALUE!`.

An expression may be *volatile*, which means that it must be evaluated anew every time a recalculation is requested, whether explicitly through the GUI or by editing a cell. Volatile expressions are non-deterministic and may evaluate to different values every time they are recalculated. An expression is volatile if it calls a volatile function. An example of such a volatile function is `NOW`: it returns the current time as a single number. We discuss the semantics of recalculation in Sec. 2.2.4.

2.1.1 Absolute and Relative Cell References

A cell reference can be absolute or relative. This distinction is only relevant when copying a cell reference to another cell. An absolute cell address is written by prefixing column and row with the `$` character. If the user copies the absolute cell reference `A1` to any other cell, it will still refer to cell A1. A relative cell address is just written plainly, e.g. `A1`. When copied to a different cell, the relative cell address will be *adjusted*. Copying the reference `A1` from cell B1 to cell B2 will adjust it to `A2`. It is

	A	B	C		A	B	C
1	10	23	=A\$1 * B1	1	10	23	=R1C1 * R[0]C[-1]
2		42	=A\$1 * B2	2		42	=R1C1 * R[0]C[-1]
3		99	=A\$1 * B3	3		99	=R1C1 * R[0]C[-1]

Figure 2.2: A spreadsheet in A1 format (left) and in R1C1 format (right). In R1C1 format, it is straightforward to see that all formula expressions in column C are the same.

A1	R1C1	Description
A1	R[-1]C[-1]	Relative: one row up, one column left.
A2	R[0]C[-1]	Relative: same row, one column left.
B3	R[1]C[1]	Relative: one row down, one column right.
A\$1	R1C[-1]	Row-absolute: one column left, always row 1.
\$A\$1	R1C1	Absolute: always cell A1.

Table 2.1: Cell references in A1 and R1C1 format. We assume the formula containing the cell address is located in cell B2.

possible to make a cell reference only column- or row-absolute. The cell reference \$A1, for example, always refers to column A when copied.

In the alternative reference format called R1C1, both rows and columns are numbered, starting from one, and their order is reversed. While the A1 format is easier to read for users, the R1C1 format makes it explicit when adjacent cells use the same relative address, as shown in Fig. 2.2. A R1C1 cell reference such as R1C1 is absolute and references the cell in row one, column one, i.e. cell A1; whereas R[1]C[1] is relative and references the cell one row below, one column to the right of the cell that contains it. Table 2.1 shows how the A1 format translates to R1C1.

2.1.2 Array Formulas

Spreadsheets allow for so-called *array formulas*: a formula expression that evaluates to an array and that is shared by multiple adjacent cells. Users can construct an array formula by first marking a rectangular cell area and then entering the array formula. The cell area should have the same *shape* as the resulting array. We say that a two-dimensional array with m rows and n columns is of shape $m \times n$. The scalar values of the resulting array are then unpacked into their respective cells. Figure 2.3 illustrates this behavior for the TRANSPOSE function. An array formula is denoted as such by the surrounding curly braces, e.g. {=TRANSPOSE(A1:C2)}.

	A	B	C
1	1	2	3
2	4	5	6
3			
4	{=TRANPOSE(A1:C2)}	{=TRANPOSE(A1:C2)}	
5	{=TRANPOSE(A1:C2)}	{=TRANPOSE(A1:C2)}	
6	{=TRANPOSE(A1:C2)}	{=TRANPOSE(A1:C2)}	

	A	B	C
1	1	2	3
2	4	5	6
3			
4	{=TRANPOSE(A1:C2)}		
5	{=TRANPOSE(A1:C2)}		
6	{=TRANPOSE(A1:C2)}		

	A	B	C
1	1	2	3
2	4	5	6
3			
4	1	4	
5	2	5	
6	3	6	

Figure 2.3: A spreadsheet containing an array formula that calls the TRANSPOSE function. **Top:** the classical visualization of an array formula in a spreadsheet. **Center:** a simplified visualization that we use throughout this thesis; it makes it more obvious that the expression is *shared* among the cells. **Bottom:** the spreadsheet after automatic unpacking of the resulting array.

Spreadsheet software usually displays the shared formula expression in each cell of the array formula. Throughout this dissertation, we simplify this visualization by removing the separating bars between the cells of an array formula and only display the shared formula once. This makes it more obvious that the formula expression is shared among a block of cells.

2.2 Formal Semantics of Spreadsheets

This section describes the formal semantics of spreadsheets and is based on Sestoft's [93] formal model of spreadsheet evaluation.

2.2.1 Dependency and Support Graph

Cell references from one cell to another can be modeled as a *dependency graph* between cells. A dependency graph is a directed graph and edges

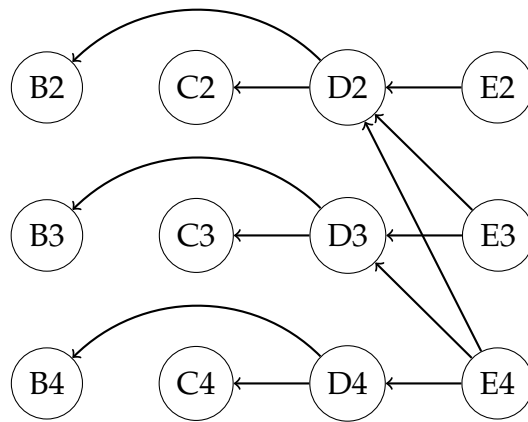


Figure 2.4: The dependency graph for the accounting spreadsheet from Fig. 2.1. Arrows go from a cell to all cells that its formula expression explicitly refers to.

go from a cell to all the cells that its formula expression refers to. If cell D2 contains the formula expression $\text{=B2} - \text{C2}$, as in Fig. 2.1, then there is an edge from D2 to B2 and C2, as illustrated in Fig. 2.4. We can use the dependency graph to construct its inverse, the *support graph* by inverting the direction of the edges. The support graph is useful for determining which cells to recompute after the user has made changes to one or more cells in a spreadsheet, as we will see in Sec. 5.2.

The support graph defines a *support set* for each cell. The support set of a cell is the set of cells that are directly reachable from the cell via the support graph. For instance, the support set of cell D2 from Fig. 2.4 is $\{E2, E3, E4\}$; the support set of cell B2 is $\{D2\}$.

The dependency and support graphs may be cyclic. Detection of cyclic dependencies in spreadsheet software is usually *non-strict*. That is, the presence of a cycle in the dependency graph does not necessarily evaluate to a cyclic dependency at run time, as illustrated in Fig. 2.5, where there is a cyclic dependency between cells A2 and A3. In this example, even though there is a statically cyclic dependency, only when the call to the random-number function RAND in cell A3 evaluates to a value greater than or equal to 0.5 and the else-branch of the conditional is evaluated, recalculation will detect a cyclic dependency. Otherwise, recalculation will evaluate the then-branch and read the current time from cell A1.

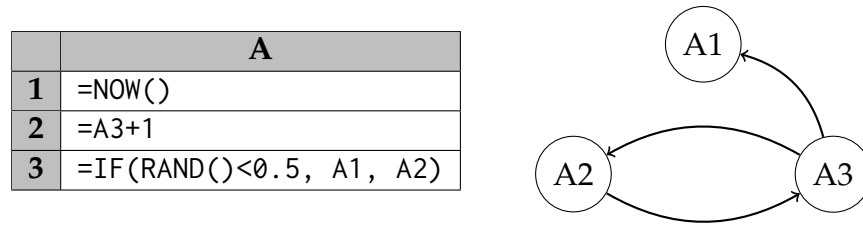


Figure 2.5: A spreadsheet (left) with a statically cyclic reference that is guarded by a non-deterministic conditional and its corresponding dependency graph (right). A dynamic cycle only occurs if the call to RAND in cell A3 evaluates to a value greater than or equal to 0.5.

2.2.2 Types of Recalculation

There are two types of recalculation. *Full recalculation* unconditionally reevaluates all formula cells. *Minimal recalculation* only reevaluates a subset of all cells. This subset consists of the cells reachable from all edited and volatile cells via the support graph. The edited cells are those modified by the user, while volatile cells must always be recalculated whenever a recalculation is triggered (see Sec. 2.1).

We call the cells that start a minimal recalculation the *recalculation roots*. In Fig. 2.1, when a user changes the value in C2 to 0, then C2 is a recalculation root and, according to the graph in Fig. 2.4, the cells D2, E2, E3 and E4 must be updated to reflect the change. If a user requests a minimal recalculation through the GUI without editing a cell (e.g. by pressing F9 in Microsoft Excel), then the recalculation starts only from volatile cells.

2.2.3 A “Standard” Formula Language

In this section, we describe the syntax for what we consider a “standard” formula language, based on the language described in [93, Sec. 1.8], but we gloss over cell ranges and array formulas. This language is only intended to inform the reader; we will not use it for technical matters in Part II of the thesis.

Our language has number values, errors, cell references, conditionals and function application, as shown in Fig. 2.6. It differs from the formula language in Microsoft Excel in that it has no boolean values. Instead, we interpret number values as booleans, where 0 is equivalent to *false* and non-zero values are equivalent to *true*. For instance, the comparison 42

$v ::= n$	Number constant, e.g. 23 or 1.34.
err	Error constant, e.g. #VALUE! or #CYCLE!.
$e ::= v$	
ca	Cell address, e.g. G5 or \$B3.
$IF(e, e, e)$	Conditional.
$F(e, \dots)$	Function application.
$RAND()$	Volatile function call.

Figure 2.6: A "standard" formula expression language.

$= 23$ returns \emptyset . We omit defining infix operators and instead use them like functions, i.e. we express what corresponds to $1 + 2$ as $+(1, 2)$. We also make no distinction between relative and absolute cell references in the A1 format. Their semantic difference is limited to user editing (see Sec. 2.1.1), which is not part of the formal semantics [93].

Moreover, we need to be able to resolve cell references. We use two functions to represent a spreadsheet [93]: σ maps from cell addresses to values; ϕ maps from cell addresses to formula expressions. Constant cells are not in the domain of ϕ . We use an *evaluation dynamics* [51, Chapter 7] for describing the operational semantics of spreadsheet formula evaluation. An evaluation has the form

$$\sigma \vdash e \Downarrow v$$

and it is read as “given the spreadsheet σ , formula expression e may evaluate to value v ”. We say “may”, because volatile functions make the language non-deterministic. Figure 2.7 shows the evaluation rules for our standard formula language [93]. The rule

$$(e2e) \frac{\sigma \vdash e_1 \Downarrow err}{\sigma \vdash IF(e_1, e_2, e_3) \Downarrow err}$$

has a premise, written above the line, and a conclusion, written below the line. It reads as “given a spreadsheet σ , if the expression e_1 evaluates to err (premise), then the expression $IF(e_1, e_2, e_3)$ evaluates to an error err (conclusion).”

The rules in Fig. 2.7 have the following meaning:

- Rule (e1e) says that if a cell address is not in the domain of σ , i.e. it is empty, the cell address expression evaluates to 0.

$$\begin{array}{c}
\text{(e1e)} \frac{ca \notin \text{dom}(\sigma)}{\sigma \vdash ca \Downarrow 0.0} \\
\\
\text{(e1)} \frac{ca \in \text{dom}(\sigma) \quad \sigma(ca) = v}{\sigma \vdash ca \Downarrow v} \\
\\
\text{(e2e)} \frac{\sigma \vdash e_1 \Downarrow \text{err}}{\sigma \vdash \text{IF}(e_1, e_2, e_3) \Downarrow \text{err}} \\
\\
\text{(e2f)} \frac{\sigma \vdash e_1 \Downarrow 0.0 \quad \sigma \vdash e_3 \Downarrow v}{\sigma \vdash \text{IF}(e_1, e_2, e_3) \Downarrow v} \\
\\
\text{(e2t)} \frac{\sigma \vdash e_1 \Downarrow v' \quad v' \neq 0.0 \quad \sigma \vdash e_2 \Downarrow v}{\sigma \vdash \text{IF}(e_1, e_2, e_3) \Downarrow v} \\
\\
\text{(e3e)} \frac{\sigma \vdash e_i \Downarrow v_i \quad \exists j. v_j = \text{err}}{\sigma \vdash \text{F}(e_1, \dots, e_n) \Downarrow \text{err}} \\
\\
\text{(e3)} \frac{\sigma \vdash e_i \Downarrow v_i \quad \forall i. v_i \neq \text{err}}{\sigma \vdash \text{F}(e_1, \dots, e_n) \Downarrow v} \\
\\
\text{(e4)} \frac{0.0 \leq v \leq 1.0}{\sigma \vdash \text{RAND}() \Downarrow v}
\end{array}$$

Figure 2.7: Evaluation semantics of a “standard” formula expression language.

- Rule (e1) says that a cell reference whose cell is not empty evaluates to the value that has been calculated for the corresponding cell in σ .
- Rule (e2e) says that if the condition of the conditional evaluates to an error, then the entire conditional may evaluate to the same error.
- Rule (e2f) says that if the condition evaluates to 0, the conditional may evaluate to the value that the *else*-expression e_3 evaluates to.
- Rule (e2t) says that if the condition evaluates to a non-zero value, the conditional may evaluate to the value that the *then*-expression e_2 evaluates to.
- Rule (e3e) says that if any argument to a function call evaluates to an error, then this error is propagated and becomes the result of the entire function call.
- Rule (e3) says that if all of the arguments to a function call evaluates to proper values, then the function call may evaluate to a value as defined by the function (whose definition we omit).
- Rule (e4) says that a volatile expression, in this case calling the RAND function, may evaluate to any number between and including zero and one. Calling the RAND function never evaluates to an error and always allows the expression that contains it to be evaluated.

In particular, rules (e2e), (e2f) and (e2t) allow for cyclic dependencies that are *statically* present in the formula expressions but not evaluated *dynamically*, as described in Sec. 2.2.1.

2.2.4 Semantics of Recalculation

Recalculation is the process of bringing the spreadsheet to a consistent state [93]. If a user changes the formula in a cell, then the cell and all the cells that depend on it, directly or indirectly, must be updated. Using the definitions from Sec. 2.2.3, we can formulate a *consistency requirement* for successful spreadsheet recalculation as [93, Sec. 1.8.3]:

$$\text{dom}(\sigma) = \text{dom}(\phi) \quad (2.1)$$

$$\forall ca \in \text{dom}(\phi). \sigma \vdash \phi(ca) \Downarrow \sigma(ca) \quad (2.2)$$

Requirement (2.1) states that the domains of ϕ and σ must be the same. This implies that a recalculation does not evaluate constants as they involve no work. Requirement (2.2) states that for every cell ca in the domain of ϕ , and thus also in the domain of σ by way of (2.1), its formula $\phi(ca)$ must evaluate to $\sigma(ca)$. The consistency requirement does not specify how recalculation must otherwise proceed; sequentially, in parallel, in which order, or how many times a cell should be evaluated.

In this model, a cyclic dependency can be resolved by assigning the cell that contains the cyclic reference the `#CYCLE!` error as a value. The error is then propagated by rules (e2e) and (e3e) such that the spreadsheet eventually assumes a consistent state. The semantics does not specify *how* cyclic dependencies are detected.

2.3 Funcalc and Sheet-Defined Functions

Funcalc [93] is an experimental spreadsheet engine implemented in C# for the .NET platform. Funcalc features (1) a core spreadsheet implementation that recalculates spreadsheets through interpretation (see Sec. 5.2) and (2) a compiler for user-defined functions to .NET byte code. In popular spreadsheet software, as for instance Microsoft Excel, defining new functions requires the use of an external language, e.g. Visual Basic. Funcalc implements the concept of *sheet-defined functions* (SDF): users can define new functions using only familiar spreadsheet concepts [66]. Funcalc's compiler infrastructure is described in detail in [93, Part II]. Compiling SDFs to .NET byte code is not a focus of this thesis and we will largely gloss over its mechanisms.

2.3.1 Sheet-Defined Functions

To define a new SDF, the only new function that users need is the meta-function `DEFINE` [93]. It takes as arguments (1) the name of the function to define, (2) the address of the cell that will contain the result of the computation, i.e. the function's return value, and (3) $k \geq 0$ cell addresses that contain input arguments. The size of k determines the arity of the newly defined SDF and it is legal to define SDFs of arity 0. An input cell may be empty; the expression that it contains is irrelevant for the function definition. Input, output and intermediate cells are highlighted using different colors. SDFs can only be defined on special function sheets that

	A	B
1	=DEFINE("FIB", A2, B1)	
2	=IF(B1 <= 1, 1, B2)	=FIB(B1 - 1) + FIB(B1 - 2)
3		
4	Will evaluate to 34:	=FIB(8)

Figure 2.8: The recursive Fibonacci function as SDF in Funcalc. The formula expression in cell A1 defines the function and makes it available on all spreadsheets. Cell B1 is the input cell; cell A2 is the output cell; and cell B2 is an intermediate cell that is only evaluated if the condition in cell A2 evaluates to false. The function can be called like any other function, as shown in cell B4.

are indicated by a pink margin. Figure 2.8 illustrates the definition of the well known recursive Fibonacci function as an SDF.

2.3.2 Higher-Order Programming with Sheed-Defined Functions

Funcalc is a higher-order functional language and there exists a mechanism for passing a (sheet-defined) function as a value and to call it subsequently. We use the CLOSURE function to create a function value, i.e. a closure, for a function or an operator. Its arguments are (1) the name of the function that we want to create a closure for as a string constant or a function value, and (2) a list of arguments that is either empty or of length k for a k -ary function. By passing additional arguments to CLOSURE we perform partial application.¹ Moreover, we can perform out-of-order partial application by passing instead a not-available error by calling the NA function in place of a real value.

We can call a function value by passing it to the APPLY function. The caller must supply the missing arguments; APPLY returns an error value if the number of supplied arguments does not match the arity of the function value. Figure 2.9 illustrates different uses of CLOSURE and APPLY.

Funcalc has additional array-related higher-order functions built into it and we discuss these in Sec. 3.2.

¹Partial application is not the same as partial evaluation. Partial application binds a value to a variable name without executing the function; partial evaluation also performs execution. Funcalc also supports partial evaluation [93, Chapter 10]. Partial evaluation is not discussed further in this thesis.

	A	B
1	=CLOSURE("*")	=APPLY(A1, 3, 2)
2	=CLOSURE("*", NA(), NA())	=APPLY(A2, 3, 2)
3	=CLOSURE("*", 3, NA())	=APPLY(A3, 2)
4	=CLOSURE("*", NA(), 2)	=APPLY(A4, 3)
5	=CLOSURE("*", 3, 2)	=APPLY(A5)

Figure 2.9: A spreadsheet that illustrates the creation of different closures from the multiplication operator *. All formula expressions in column B evaluate to the value 6.

2.4 Related Work

The majority of this section is based on the technical report “**Declarative Parallel Programming in Spreadsheet End-User Development**” [11, Sec. 3]. Sestoft [93] gives an even more thorough review of the literature on spreadsheet technology.

There are many different topics in spreadsheet research, e.g. error detection [58] and automatic repair [39, 40, 41], visualization [59, 89] or analysis of “real-life” spreadsheet corpora [42, 56]. In this section we focus on research on end-user programming and its different incarnations in a spreadsheet model of computations.

One major topic in research on spreadsheet end-user programming is the lack of abstraction: spreadsheets bundle data and computations in a single representation [64]. Moreover, spreadsheets encourage copying of formulas across cells to replicate computations rather than using high-level expressions [7, 82]. Miller [81] claims that the lack of abstraction makes spreadsheets less powerful than general purpose programming languages.

Another major topic is general programming paradigms in a spreadsheet model of computation. Researchers have augmented spreadsheets with object orientation [7] and more declarative programming approaches [96, 99] that we will look at in greater detail in Section 2.4.2. Even though we cannot strictly separate abstraction and programming paradigms, this categorization is convenient for the discussion of how researchers have proposed to handle the complexity of spreadsheet models.

2.4.1 Abstraction

Abstraction is the separation of data and computation. It is useful to define two kinds of spreadsheet abstraction. That is (1) manual abstraction, where users have the means to use language-provided abstraction mechanisms to hide implementation details; and (2) automatic abstraction, where users construct spreadsheets in a familiar way and later use static analyzers to infer the underlying model and subsequently separate it from the data.

Manual Abstraction

Many researchers observed that spreadsheets lack the most basic abstraction of general-purpose programming languages: named functions [66]. Named functions make it possible to encapsulate and hide implementation detail that is unimportant for the overall computations of a specific model. Therefore, Jones et al. [66] proposed to allow end-users to define their own abstractions in terms of spreadsheet computations. Each newly introduced function is essentially a spreadsheet “prototype” that has one or more designated input cells and a designated output cell. Each time the user calls such a sheet-defined function, a new spreadsheet instance of this spreadsheet prototype is generated to perform the computation.

Sestoft [93] extended upon this idea by allowing sheet-defined, recursive, run-time compiled functions. This approach is more general and alleviates the need for instantiating explicit spreadsheets. This approach is implemented in the experimental spreadsheet engine Funcalc which is described in great detail in [93] and discussed in this thesis in Sec. 2.3 and 5.2.

Automatic Abstraction

Automatic analysis of spreadsheets to infer their model and to subsequently separate this model from the data allows users to build spreadsheets in a familiar manner using familiar tools. Isakowitz et al. [64] developed a system that automatically performs such a separation and manages spreadsheet logic for modular re-use. They observe that the majority of spreadsheet errors they encounter are not simple off-by-one reference errors and typos but severe errors in the model. They relate them to classic programming errors where the programmer has not chosen an adequate level of abstraction.

The visual layout of spreadsheets is often regarded as implicit documentation of the spreadsheet's logic. Mittermeir and Clermont [82] however, observe that this visual layout often leads to misconceptions if another user takes over the spreadsheet. Therefore, they developed a set of logical and semantic equivalence classes for cells. These equivalence classes help visualizing repetitions in spreadsheet grids, which are the high-level structures a user needs to understand in order to be able to maintain the spreadsheet [57].

Types are useful abstractions over spreadsheets. The literature includes different type inference systems that make the user aware of formulas where the expected type differs from the actual type [1, 31]. Researchers have proposed different solutions to handle types in spreadsheets and a common problem is efficient typing of cell areas [1, 30]. We classify types as a kind of automatic abstraction because the types often are inferred rather than annotated.

2.4.2 Programming Paradigms

The literature contains a variety of approaches to bringing different programming paradigms to the spreadsheet domain with a focus on raising the abstraction level.

Object Orientation

Functional Model Development (FMD) [7] is a domain-specific language for Microsoft Excel and exposes objects to spreadsheet users. Objects are accumulations of data with functions defined on them. FMD provides a special syntax for declaring variables that model input parameters for user-defined functions. Functions are defined inline on the same spreadsheet using prototype formulas where the cell that would yield the result actually evaluates to the newly defined function. As spreadsheets encourage copying of formulas over the same column or row, FMD introduces a high-level map construct that applies the same user-defined function across a column or row.

To apply stronger separation of implementation and instantiation, Mendes [80] developed ClassSheet. The logic of a computation is defined in a model-spreadsheet, while each common spreadsheet that performs the computations is an instance of this model. This approach resembles a manual version of the abstraction model by Isakowitz et al. [64].

Constraint Programming

Stadelmann [99] proposed to let spreadsheet users express their models by providing the system with constraints to solve. This approach considerably reduces the amount of code required to model complicated logic [99]. However, due to the requirement of being able to name a cell multiple times in a particular constraint, it is infeasible to let constraints directly replace cell formulas. Instead, the system provides a second window that contains constraints. This side-steps the spreadsheet model slightly.

By Example

Programming by example allows users to explain how to transform data by performing a few transformations manually, from which the system can infer general transformation rules. This is useful for bulk-processing similar items. Barowy et al. [5] and Singh and Gulwani [96] extended Microsoft Excel with a domain-specific language (DSL) that allows users to provide such example transformations such that Excel then can automatically transform additional items. They combine a probabilistic approach of parsing with learning of transformation rules. Their approach is related to formula copying but more general, because the data in the input cells is not required to be uniform; e.g. a column of dates can be transformed by giving more than one example if the format of the date varies from cell to cell. The transformation function, however, is not reified in the spreadsheet formula itself.

Chapter 3

Declarative Array Programming

Array programming, sometimes referred to as array-oriented programming, focuses on operations on arrays. When we talk about array-oriented languages, what we often really mean are *collection-oriented languages* [97]; however, arrays are abundant in nearly all programming languages and a well understood concept among programmers, hence we stick to the name.

In array-oriented languages, we distinguish between scalar values, such as a single floating point number, and non-scalar values, such as arrays of floating point numbers. Programs written in array-oriented languages are often highly data-parallel and parallelism is extracted via a single-instruction-multiple-data (SIMD) model: operations are applied to all elements of an array and therefore may be executed in parallel.

There are two major strands of array programming. One is imperative and loop-based where each iteration of a loop can run in parallel if there are no dependencies between any two iterations. Examples of such languages are Single-assignment-C (SAC) [49] and Fortran.

The other strand, which this thesis focuses on, is declarative array programming, where scalar operations are *lifted* to work on arrays. Lifting happens either implicitly, as for instance in APL [65], where it is legal to apply the binary addition operator either to two scalar values or to two arrays and where the latter sums the two arrays element-wise and returns a new array; or explicitly by using higher-order functions over arrays, such as map or reduce.

It is arguably easier to parallelize programs written in a declarative language without side-effects [20]. In the remainder of this dissertation,

we will only concern ourselves with declarative array-oriented programming.

3.1 Array Combinators

The array programming model that we focus on uses higher-order functions to express bulk-operations on arrays. Throughout the remainder of this thesis, we call a higher-order function that lifts a function from operating on scalar values to operate on arrays an *array combinator*. In this section, we briefly introduce three of the most important array combinators: map, reduce and scan. Note that we write function application without parenthesis. The expression $f\ x\ y$ means function f is applied to arguments x and y .

3.1.1 Map

The $\text{map } f\ xs$ combinator has type $(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$ which means that it accepts as arguments (1) a function f that takes values from type α to type β (written $\alpha \rightarrow \beta$) and (2) an array xs containing values of type α (written $[\alpha]$) and applies f to each element of the array; the result is a new array that contains values of type β :

$$\text{map } f\ [x_1, x_2, x_3, \dots, x_n] \equiv [f\ x_1, f\ x_2, f\ x_3, \dots, f\ x_n]$$

Since all applications of f are independent from each other, map is allowed to perform each application in parallel. Moreover, map can be naturally extended to take more than one array as argument, such that we can use a binary (2-ary) function, or operator, \oplus to combine two arrays element-wise. We call this combinator zipWith , or sometimes map2 and it has type $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow [\gamma]$. It behaves as follows:

$$\text{zipWith } \oplus\ [x_1, x_2, \dots, x_n]\ [y_1, y_2, \dots, y_n] \equiv [x_1 \oplus y_1, x_2 \oplus y_2, \dots, x_n \oplus y_n]$$

In principle, we can define a map-like combinator for any number of arrays, e.g. map3 , map4 and so on. The map_k combinator would take a k -ary function as an argument and k arrays of the same *shape*. Two one-dimensional arrays are of the same shape if they have the same length; two two-dimensional arrays are of the same shape if they have the same number of rows and the same number of columns; and so on.

3.1.2 Reduce

Often, it is useful to compute a single scalar value from an array. In array programming, this is usually done by using the reduce $\oplus \varepsilon xs$ combinator of type $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha] \rightarrow \alpha$. It takes (1) a binary operator \oplus that combines two scalar values of type α and returns a scalar result of type α , (2) a “default” value ε of type α from where the reduction begins and (3) an array to reduce:

$$\text{reduce } \oplus \varepsilon [x_1, x_2, \dots, x_n] \equiv \varepsilon \oplus x_1 \oplus x_2 \oplus \dots \oplus x_n$$

We can use reduce to implement summation of an array of numbers in a straightforward fashion as $\text{reduce } (+) 0$ using the addition operator $+$ and its identity, which is 0.

The reduce combinator does not specify an order of application. Assuming that the operator \oplus is associative and that ε is the identity element for \oplus such that $\varepsilon \oplus x \equiv x \oplus \varepsilon \equiv x$ then the order of the \oplus applications does not matter and reduce can run in parallel:

$$\text{reduce } \oplus \varepsilon [x_1, x_2, \dots, x_{n-1}, x_n] \equiv \varepsilon \oplus (x_1 \oplus x_2) \oplus \dots \oplus (x_{n-1} \oplus x_n)$$

The sub-expressions $x_1 \oplus x_2$ and $x_{n-1} \oplus x_n$ and all sub-expressions indicated by the ellipses can be evaluated in parallel and combined by recursive application of \oplus . If the argument array is empty, reduce just returns ε .

There exists a popular generalization of reduce which is called fold and it has type $(\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$. Its type is more general, but this increased generality comes at a cost. An implicit parallelization of fold is not possible, because here the binary function requires one argument of type β and one of type α : it has no way of combining two values of type β and therefore cannot combine partial results. Hence, fold cannot be parallelized [101].

One way to achieve greater generality of reduce is to combine it with map to a single combinator mapReduce of type $(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\beta]$:

$$\text{mapReduce } f \oplus \varepsilon [x_1, x_2, \dots, x_{n-1}, x_n] \equiv \varepsilon \oplus (f x_1 \oplus f x_2) \oplus \dots \oplus (f x_{n-1} \oplus f x_n)$$

It can be curried with the identity function id to elegantly implement reduce as $\text{mapReduce } id$.

3.1.3 Scan

The scan combinator is a generalization of computing the prefix sums [16] for a list of numbers [17]. For instance, the prefix sums of the list $[1, 2, 3, 4]$ are $[1, 3, 6, 10]$. The idea is to reduce a collection and to also return the intermediate reduction results. There are variations in naming and definition in literature and popular use. We define $\text{scan } \oplus \ \varepsilon \ xs$ as follows. The combinator has type $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha] \rightarrow [\alpha]$:

$$\text{scan } \oplus \ \varepsilon \ [x_1, x_2, x_3, \dots] \equiv [\varepsilon \oplus x_1, \varepsilon \oplus x_1 \oplus x_2, \varepsilon \oplus x_1 \oplus x_2 \oplus x_3, \dots]$$

Even though scan seems inherently sequential, Blelloch [17] showed how to parallelize this combinator for associative operators. If $(x \oplus y) \oplus z \equiv x \oplus (y \oplus z)$, it is possible to use a divide-and-conquer approach to parallelization by recursively applying \oplus to partial results. The parallel scan combinator is also used in hardware: Hinze [60] and Sheeran [95] have used the principles behind parallel scan to design optimal parallel prefix-sum circuits.

3.1.4 Scanning Two-Dimensional Arrays

It is trivial to generalize map and reduce to two-dimensional arrays. Generalizing scan takes more effort. We could choose to only scan in vertical or horizontal direction, but this seems not sufficiently general. Instead, we take some inspiration from computer graphics, where it is common to work with two-dimensional arrays. In computer graphics, prefix sums in two dimensions are called *summed-area tables* [36] and are used to compute the integral image of an input image. We define the combinator scan2d as a generalized summed-area table: a wavefront computation for arbitrary 4-ary functions over a two-dimensional array, starting at the upper-left of the array and progressing towards the lower-right. A basic definition of this wavefront computation could be

$$\text{scan2d}' \ f \ xs \equiv ys$$

where

$$ys[i, j] = f \ \ ys[i, j-1] \ \ ys[i-1, j-1] \ \ ys[i-1, j] \ \ xs[i, j].$$

This definition, however, does not define what to do in the case of $i = 0$, i.e. the first row, and $j = 0$, i.e. the first column. We could just skip the

first row and column, but this seems rather limiting. Instead, we use multiple ε values to initialize the fringes of the two-dimensional scan. A more general definition $\text{scan2d } f \ \gamma \ \delta \ \rho \ xs$ takes a 4-ary function f , an $m \times 1$ single-column array γ , a scalar value δ and a $1 \times n$ single-row array ρ as well as an input array xs of shape $m \times n$. Its result is a new $m \times n$ array

$$\text{scan2d } f \ \gamma \ \delta \ \rho \ xs \equiv ys$$

where:

$$ys[0,0] = f \ \gamma[0] \ \delta \ \rho[0] \ xs[0,0] \quad (3.1)$$

$$ys[0,j] = f \ ys[0,j-1] \ \rho[j-1] \ \rho[j] \ xs[0,j] \quad (3.2)$$

$$ys[i,0] = f \ \gamma[i] \ \gamma[i-1] \ ys[i-1,0] \ xs[i,0] \quad (3.3)$$

$$ys[i,j] = f \ ys[i,j-1] \ ys[i-1,j-1] \ ys[i-1,j] \ xs[i,j] \quad (3.4)$$

We use the values from γ , ρ and δ as if they were positioned around the upper and left fringes of xs (see Fig. 3.1). Equation (3.1) defines the first element of ys at $(i,j) = (0,0)$, on which all other values of ys depend. Since no values precede it, we must refer to values from γ , δ and ρ instead. Equation (3.2) defines the first row and hence refers to ρ ; Eq. (3.3) defines the first column and therefore refers to γ . Finally, Eq. (3.4) is the general case for all remaining index pairs (i,j) .

The next question to ask is how to parallelize scan2d . Again, this is rather trivial for two-dimensional generalizations of map and reduce , because there is no sequential dependency between any two sub-computations.

Scanning a two-dimensional array only row- or column-wise gives us the opportunity to run each row- or column-wise scan independently from one another and hence in parallel, even without requiring an associative operator as function argument. In the general case, however, later rows and columns depend on both earlier rows and columns.

Instead of trying to parallelize row- or column-wise, we can split up the argument array xs into quadrants, as shown in Fig. 3.1. 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 can implement this parallelization scheme recursively on each sub-array until either a minimum size is reached or we have exploited all available cores. Figure 3.2 illustrates the recursive parallel computation graph

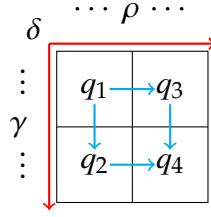


Figure 3.1: Wave front scheme of the `scan2d` combinator. We process from the top left to the bottom right of the two-dimensional array, as indicated by the red arrows. The quadrants q_2 and q_3 depend on q_1 , while q_4 depends on all of these, as indicated by the blue arrows. Values γ , δ and ρ are initial values at the fringes. Note that quadrant numbers are *not* as in plane geometry.

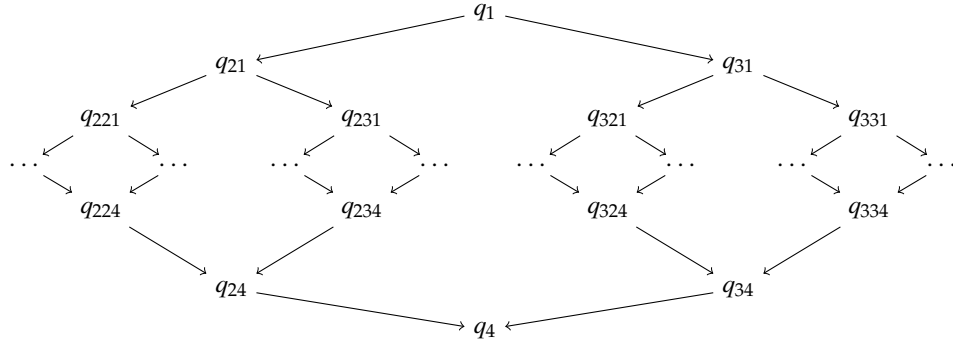


Figure 3.2: The parallel computation graph of `scan2d`. The number of nodes at each level indicates the available parallelism. The node labeled q_{21} is the first quadrant at level two of the node at the second quadrant at level one.

for `scan2d`, which looks much like the graph for parallel scan on one dimensional arrays [17].

Note that parallel `scan2d` does not require f to have any special properties for correct parallelization.

3.2 Array Programming in Funcalc

Using the taxonomy introduced by Sipelstein and Blelloch [97], arrays in Funcalc are nested, heterogeneous, (two-dimensional) grid-ordered, finite collections. Funcalc arrays are two-dimensional due to the two-dimensional cell layout of spreadsheets. Referencing a cell range such as `A1:J10` returns an array of ten rows and ten columns. Therefore, some

Type	Short	Description
<i>NumberValue</i>	<i>n</i>	A number.
<i>ArrayValue</i>	<i>arr</i>	An array containing values.
<i>FunctionValue</i>	<i>f</i>	A closure of a function.
<i>Value</i>	<i>v</i>	Subsumes all other types.

Table 3.1: Some Funcalc types and their abbreviations.

operations on arrays are duplicated and operate either on the vertical (row-wise) or the horizontal (column-wise) dimension; for instance, we can reverse an array either vertically using `VREV` or horizontally using `HREV`, as described in Sec. 3.2.1. Arrays in Funcalc are 1-based, unlike C-style arrays which are 0-based. Unlike Microsoft Excel, Funcalc allows an array value to be stored in a single cell. If this was not allowed, it would be impossible to define SDFs over arrays, as illustrated in Sec. 3.2.2 and 3.2.3.

3.2.1 Functions and Combinators on Arrays

In the following, we list all the relevant functions and combinators on arrays, their signature and give a brief description of their semantics. We use the type abbreviations listed in Table 3.1 for conciseness. A type annotated with a $^+$ symbol, e.g. arr^+ , means “one or more values of this type”. The function signature $f \times arr^+ \rightarrow arr$ means that the function accepts as arguments a function and one or more arrays and returns an array.

COLUMNS : $arr \rightarrow n$

Returns the number of columns of the array.

CONSTARRAY : $v \times n \times n \rightarrow arr$

Calling `CONSTARRAY(v, m, n)` returns a new array of shape $m \times n$ with value *v* at every index position.

HCAT : $arr \times arr \rightarrow arr$

Concatenates two arrays with an equal number of rows column-wise.

HPREFIX : $f \times arr \times arr \rightarrow arr$

Computes the row-wise generalized prefix-sum (see Sec. 3.1.3) for an array of shape $m \times n$ (last argument). The first array (second

argument) is of shape $m \times 1$ and contains initializer values for each row.

HREP : $arr \times n \rightarrow arr$

Takes an array of shape $r \times 1$ and returns a new array of size $r \times n$ by repeating the single column of arr exactly n times.

HREV : $arr \rightarrow arr$

Reverses the order of values in column-direction.

INDEX : $arr \times n \times n \rightarrow v$

Calling $\text{INDEX}(arr, r, c)$ returns the value at row r , column c or a #REF! error if r or c are less than 1 or greater than the number of rows, columns respectively, in arr .

MAP : $f \times arr^+ \rightarrow arr$

Funcalc's MAP combinator is variadic and takes a k -ary function and k arrays that must all have the same shape; see Sec. 3.1.1.

PREFIX : $f \times arr \times v \times arr \times arr^+ \rightarrow arr$

Computes a generalized two-dimensional prefix sum for a $3 + k$ -ary function and k arrays of shape $m \times n$; see Sec. 3.1.4.

REDUCE : $f \times v \times arr \rightarrow v$

Exactly as described in Sec. 3.1.2.

ROWS : $arr \rightarrow n$

Returns the number of rows of the array.

SLICE : $arr \times n \times n \times n \times n \rightarrow arr$

$\text{SLICE}(arr, i, j, m, n)$ returns a sub-array of array arr that starts at the upper-left corner (i, j) and ends at the lower-right corner (m, n) .

SUM : $arr \rightarrow n$

Computes the sum of all elements in the array.

TABULATE : $n \times n \times f \rightarrow arr$

Calling $\text{TABULATE}(m, n, f)$ returns a new array arr of shape $m \times n$ such that $\text{INDEX}(arr, r, c) \equiv f(r, c)$ where $1 \leq r \leq m$ and $1 \leq c \leq n$.

TRANPOSE : $arr \rightarrow arr$

Transposing an array arr of shape $m \times n$ returns a new array arr' of shape $n \times m$ such that $\text{INDEX}(arr', r, c) \equiv \text{INDEX}(arr, c, r)$.

VCAT : $arr \times arr \rightarrow arr$

Concatenates two arrays with an equal number of columns row-wise.

VPREFIX : $f \times arr \times arr \rightarrow arr$

Computes the column-wise generalized prefix-sum (see Sec. 3.1.3) for an array of shape $m \times n$ (last argument). The first array (second argument) is of shape $1 \times n$ and contains initializer values for each column.

VREP : $arr \times n \rightarrow arr$

Takes an array of shape $1 \times c$ and returns a new array of shape $n \times c$; it works like HREP but by repeating rows.

VREV : $arr \rightarrow arr$

Reverses the order of values in row-direction.

3.2.2 Example: Matrix Multiplication

A useful numeric function is matrix multiplication. In this section, we show how to implement matrix multiplication in Funcalc and we start with a reference implementation in F# [104] that is free of side-effects and operates on immutable, two-dimensional arrays. In F#, the type of a two-dimensional array of type α is written as 'a [,].

Our implementation uses higher-order array combinators. Function `mmult` (Fig. 3.3) takes two arguments of type `float [,]` of shape $k \times n$ (`lm`) and $m \times k$ (`rm`) and returns a new value of type `float [,]` of shape $m \times n$. To compute the value at index (i, j) , we can compute the dot-product of row i of `lm` (`r_i`) and column j of `rm` (`c_j`). The inner function `dot` implements this by (1) combining `r_i` and `c_j`, which are both of type `float [,]` and of shape $1 \times k$, point-wise using `zip_with` and the multiplication operator and (2) summing the resulting array. The types of the used functions and array combinators are listed above function `mmult` in Fig. 3.3.

The Funcalc formula language is expressive enough to follow this reference implementation closely, as shown in Fig. 3.4. There are two major differences: (1) MAP in Funcalc has variadic arity and accepts as arguments a k -ary function and k arrays of equal shape, as described in Sec. 3.2.1; and (2), Funcalc does neither have syntax for anonymous nor for inner functions, which is why the DOT function has four parameters—the two arrays from which it should slice sub-arrays and a row and a

```

1  val columns : 'a[,] → int
2  val init :
3    int → int → (int → int → 'a) → 'a[,]
4  val rows : 'a[,] → int
5  val slice :
6    'a[,] → int → int → int → int → 'a[,]
7  val sum : float[,] → float
8  val transpose : 'a[,] → 'a[,]
9  val zip_with :
10   ('a → 'b → 'c) → 'a[,] → 'b[,] → 'c[,]
11
12  let mmult lm rm =
13    let trm = transpose rm in
14    let dot = fun i j →
15      let r_i = slice lm i 0 1 (columns lm) in
16      let c_j = slice trm j 0 1 (columns trm) in
17      sum (zip_with ( * ) r_i c_j)
18    in
19    init (rows lm) (columns rm) dot

```

Figure 3.3: An implementation of functional matrix multiplication in F#.

column index. We can use partial application to generate a closure of DOT where both arrays are already provided (Fig. 3.4, cell C7). Hence, the signature of the *residual* DOT closure is $n \times n \rightarrow n$. Therefore, we can pass it as function argument to TABULATE to generate a new array.

Furthermore, note that the SLICE function in Funcalc takes absolute row and column indices for the upper-left and the lower-right corner, whereas the function slice `xs r c m n` in F# returns a sub-array of `xs` starting at the upper-left corner (r, c) with `m` rows and `n` columns.

3.2.3 Example: Batcher Sort

Batcher Sort [6] or bitonic sort is a sorting algorithm whose structure resembles that of a bitonic sorting network. Sorting networks are highly parallel but not necessarily more efficient than other standard sorting algorithms [54]. We say that a sequence $x_1, \dots, x_k, \dots, x_n$ is bitonic if

$$x_1 \leq \dots \leq x_k \geq \dots \geq x_n.$$

	A	B	C
1	lm:		=SLICE(B1, B3, 1, B3, COLUMNS(B1))
2	trm:		=SLICE(B2, B4, 1, B4, COLUMNS(B2))
3	r:		=SUM(MAP(CLOSURE("*"), C2, C3))
4	c:		=DEFINE("DOT", C3, B1, B2, B3, B4)
5			
6	lm:		=TRANPOSE(B7)
7	rm:		=CLOSURE("DOT", B6, C6, NA(), NA())
8			=TABULATE(C7, ROWS(B6), COLUMNS(B7))
9			=DEFINE("MMULT", C8, B6, B7)

Figure 3.4: Functional matrix multiplication in Funcalc as SDF. Funcalc has no syntax for defining anonymous or inner functions, so we must make the DOT function public. Moreover, the implicit parameters (lm and trm) to the dot function from Fig. 3.3 are made explicit. Here, we bind these to the closure of DOT using partial application (cell C7).

For instance, the sequences 3,6,9,5,2,1 and 4,3,2,1 are bitonic, but 3,6,4,5 is not.

We can implement a repeating network structure using recursion [34, 61]. Function `bitonic_sort` in Fig. 3.5 sorts a bitonic sequence of length 2^k recursively by (1) splitting the input array into two equally sized sub-arrays, (2) comparing the two sub-arrays point-wise such that mins contains a bitonic sequence of values that are all strictly smaller than the values in maxs, (3) sorting these bitonic sequences recursively and (4) combining the resulting arrays by concatenation.

Function `bitonic_sort` only works for bitonic sequences. To generalize to sequences that are arbitrarily ordered, function `batcher_sort` exploits that all sequences of length 2 are bitonic. If we have two sorted sequences xs and ys , we can sort the values from both by concatenating xs to ys reversed and passing the result to `bitonic_sort`. Concatenating two sorted arrays and reversing the latter array produces a new bitonic array. This is what function `batcher_sort` in Fig. 3.5 does.

Again, Funcalc's formula language is expressive enough that we can implement Batcher's sorting algorithm as an SDF, as shown in Fig. 3.6. There are two major differences.

First, there is no tuple type in Funcalc and we therefore split function `split` from Fig. 3.5 into two functions `UPPER` and `LOWER` in that return the respective upper and lower half of a vertical column array. It would be possible to implement a function `SPLIT` in Funcalc that returns an

```

1  val ( @ ) : 'a[] → 'a[] → 'a[]
2  val init : int → (int → 'a) → 'a[]
3  val len : 'a[] → int
4  val rev : 'a[] → 'a[]
5  val zip_with :
6    ('a → 'b → 'c) → 'a[] → 'b[] → 'c[]
7
8  val split : 'a[] → 'a[] * 'a[]
9
10 let rec bitonic_sort xs =
11     if len xs <= 1 then xs
12     else
13         let lhs, rhs = split xs in
14         let mins = zip_with min lhs rhs in
15         let maxs = zip_with max lhs rhs in
16         bitonic_sort mins @ bitonic_sort maxs
17
18 let rec batcher_sort xs =
19     if len xs <= 1 then xs
20     else
21         let lhs, rhs = split xs in
22         let lhs' = batcher_sort lhs in
23         let rhs' = batcher_sort rhs in
24         bitonic_sort (lhs' @ rev rhs')

```

Figure 3.5: An implementation of Batcher Sort in F#. The definition of `split` has been omitted; it splits an array `xs` at `len xs / 2` and returns the left and the right half of `xs` as a tuple. The `@` operator performs concatenation.

	A	B	C
1	xs:		=DEFINE("BITONIC",B5,B1)
2	lhs:	=UPPER(B1)	=MAP(CLOSURE("MIN"),B2,B3)
3	rhs:	=LOWER(B1)	=MAP(CLOSURE("MAX"),B2,B3)
4		=VCAT(C4, C5)	=BITONIC(C2)
5		=IF(ROWS(B1)<=1,B1,B4)	=BITONIC(C3)
6			
7	xs:		=DEFINE("BATCHER",B11,B7)
8	lhs:	=UPPER(B7)	=BATCHER(B8)
9	rhs:	=LOWER(B7)	=BATCHER(B9)
10		=BITONIC(C11)	=VCAT(C8, VREV(C9))
11		=IF(ROWS(B7)<=1,B7,B11)	

Figure 3.6: Batcher Sort in Funcalc as SDF. The definitions of functions UPPER and LOWER have been omitted; they return the respective upper and lower half of a two-dimensional array.

array containing exactly two values to simulate tuples; we have decided against this approach in the interest of clarity.

Second, all arrays in Funcalc are two-dimensional. Therefore, the Funcalc implementation of batcher sort is slightly more general than the F# implementation: there is no requirement on the number of columns of an input array, so the SDF sorts the elements of each column of a two-dimensional array in parallel.

Moreover, it is possible to implement Batcher sort in a normal spreadsheet by unrolling the recursion and calling BITONIC directly. The trick here is to use array formulas to unpack the values from the sorted sub-arrays into single cells, as shown in Fig. 3.7.

3.3 Related Work

The majority of this section is based on the technical report “**Declarative Parallel Programming in Spreadsheet End-User Development**” [11, Sec. 2].

We have identified three key topics within (parallel) array programming: nested data parallelism, array fusion and array representation. While these are obviously connected and therefore cannot be discussed in isolation, we will use them to structure the review of the relevant research body on parallel array programming.

	A	B	C	D
1	42	{=BITONIC(A1:A2)}	{=BITONIC(B1:B4)}	{=BITONIC(C1:C8)}
2	23			
3	66	{=VREV(BITONIC(A3:A4))}		
4	534			
5	123	{=BITONIC(A5:A6)}	{=VREV(BITONIC(B1:B4))}	
6	56			
7	8	{=VREV(BITONIC(A7:A8))}		
8	21			

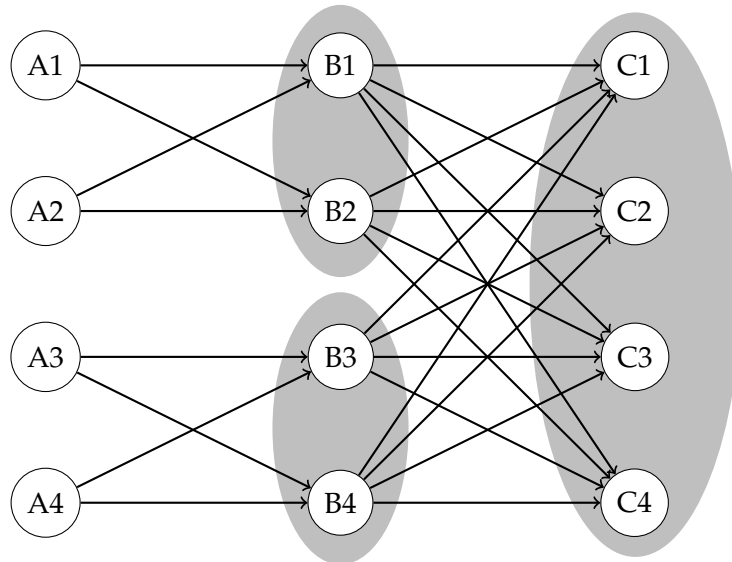


Figure 3.7: Implementing Batcher Sort in a standard spreadsheet by unrolling the recursion. **Top:** We can use array formulas to unpack the result of each call to BITONIC into single cells. **Bottom:** The support graph (or dataflow graph) of the above spreadsheet for the first two iterations; it roughly resembles a bitonic sorting network without any distinction between comparison and “flow” edges. The gray ellipses group cells that are part of the same array formula.

There are some key languages that appear in the literature. The two oldest are Fortran, which is an imperative high-performance language, and APL [65], which is a high-level declarative language. Furthermore, much research has focused on Single-assignment-C (SAC) [49], which is a C-like functional language, and Data Parallel Haskell (DPH) and REPA [68], both of which are library and compiler extensions for Haskell. Another important language is NESL [18], which introduced the idea of statically flattening irregularly nested arrays to load-balance work.

The opportunity for parallelism in these languages stems from array combinators. Where imperative languages like Fortran use iterative loops, functional languages depend on higher-order combinators such as `map` or `reduce` to express data-parallel computations. Ching [33] argues that it is easier for programmers to express parallelism declaratively than by explicitly scheduling work to different processors.

The only requirement for implicit parallelization is that the programmer writes idiomatic code which the compiler or libraries knows how to parallelize efficiently [9]. Not only is parallelism much easier to extract, but sequential programs written in such a high-level style will often perform better, too [9].

3.3.1 Nested Data Parallelism

Nested data parallelism describes the nesting of parallel operations over an array. For instance, the functional matrix multiplication algorithm described in Sec. 3.2.2 nests calls to `sum` and `zip_with` in a call to `init`, all of which can be executed in parallel.

A naive way of implementing such nested parallelism would be to start new threads from within each parallel thread which, naively done, would add an immense overhead to the program and prohibit parallel speedup. Early work on the parallel language Actus [88] recognizes the difficulty of nested data parallelism and, as a temporary solution, restricts the level of parallelism to a dimension that the programmer has to choose explicitly ahead of time. Further research on nested data parallelism has focused on eliminating this overhead in two different ways, which we discuss in the following.

Flattening

Flattening nested data parallelism statically flattens homogeneous computations on possibly irregularly nested arrays. Blelloch et al. [22] in-

roduced the idea of statically flattening irregularly nested arrays for optimal parallelization. The key to flattening is the right choice of array representation. Blelloch et al. [22] showed that there exists a representation that enables flattening of irregularly nested arrays in constant time and space. In the first-order functional language NESL, flattening requires the generation of lifted versions of all functions to the next rank. For instance, if the $+$ operator performs addition on scalars, then its lifted variant performs point-wise addition of arrays. Blelloch and Greiner [21] showed, based on NESL's cost semantics, that NESL actually can be implemented without any additional run-time overhead due to flattening.

Other researchers have since taken up the idea of flattening nested parallelism again [77] and implemented it in Haskell. In contrast to NESL, Haskell is a higher-order functional programming language with a static type system that is compiled instead of interpreted.

As it turns out, flattening nested data parallelism is much harder in higher-order languages [76]. Flattening in higher-order languages often introduces additional intermediate arrays that cost both time and space. Therefore, Keller et al. [69] developed a technique to avoid flattening in such cases.

Flattening nested data parallelism is tightly coupled to array fusion, or deforestation [111], which we will discuss in more detail in Sec. 3.3.2.

Dynamic Scheduling

Dynamic scheduling leaves the distribution of work to the run-time of the program. SAC [49] and the .NET Thread Parallel Library [74] both use work-stealing queues [28] to dynamically schedule work. Such scheduling schemes work well in practice, but they can result in some overhead at run-time that otherwise could have been alleviated by using compile-time transformations. To minimize the overhead of scheduling different kinds of computations, Fluett et al. [47] argue for a mix of schedulers and to let the run-time choose dynamically which scheduler to use for distributing work.

A notable scheduling heuristic is lazy binary splitting [106, 107]. In this scheme, every thread gets assigned some part of the workload. Threads communicate with other threads via a technique inspired by and as efficient as work-stealing queues. When a worker thread iterates over an array, it checks at every n -th iteration step whether any of the other

threads are idle. If yes, it splits its remaining array in half, dispatches the latter half to the idle threads and continues to the next iteration step. This heuristic shows low overhead in experimental settings [106].

3.3.2 Fusion

Fusion, or deforestation [111], refers to avoiding the materialization of intermediate data structures for consecutive applications of higher-order combinators. For instance, we can combine two succeeding applications of `map` into a single application as follows, where \circ is the function composition operator (we give a more thorough overview over fusion techniques in Chapter 7):

$$\text{map } f (\text{map } g \text{ } xs) \equiv \text{map } (f \circ g) \text{ } xs$$

This optimization is valuable for both sequential and parallel programs, and for example also in sequential Fortran 90 programs [63].

Chakravarty and Keller [27] express fusion transformations as straightforward equational rewrite rules in the Glasgow Haskell Compiler (GHC). They also observe that flattening nested arrays makes fusion a much more straightforward task, which emphasizes the connection between flattening and fusion. Some researchers have been focusing on making such optimizations visible to the programmer via types [76]. This enables programmers to reason about the performance of their declarative code.

More aggressive fusion is possible if the compiler performs a more complex analysis. Henriksen and Oancea [52] use a dataflow based graph-reduction to analyze functional programs with second-order combinators on arrays for fusion possibilities. Their analysis can detect code structures that inhibit fusion, subsequently re-writes the program in such a way that fusion becomes possible and avoids duplicate computations; this technique, among others, is implemented in the Futhark language [53]

A special variant of fusion is destructive update analysis. Since data structures in purely functional languages are immutable, writing to a single index of an array produces a new array. The update operation copies all elements from the original array with exception of the index that it updated. If the updated array is not referenced again throughout the program, we can perform the update in-place, or destructively, without the overhead of copying all data. Sastry and Clinger [90] developed an analysis for destructive array updates using live-variable dataflow analysis.

3.3.3 Array Representations

Choosing the right data structure is a key element to high-performance array programming where the performance of individual operations on arrays is the main concern. For instance, arrays should be able to perform constant-time lookup and preferably also constant-time update [84]. It is well-known that data structure designers often must make a trade-off between the asymptotic complexity of different operations, like random access, update and concatenation. Nevertheless, Stucki et al. [102] developed a general-purpose, immutable, array-like data structure, the *relaxed radix-bound* (RRB) tree and proved the asymptotic complexity for all operations and for practical sizes of RRB trees, bounded by the maximal word size of the underlying machine architecture, to be constant or amortized constant.

Arvind et al. [3] developed I-Structures that conceptually are write-once arrays. Each index can be written to exactly once during the entire run-time of the program. Reads from and writes to indices can be re-ordered according to re-write rules at the cost of sacrificing referential transparency.

Type-based run-time specialization of functions and optimization of data structures is a widely applicable technique in functional programming [50]. REPA [68] uses types in order to represent irregular shapes of arrays and to specialize higher-order functions accordingly. REPA arrays consist of unboxed values and are lazy. Accessing a particular element of an array forces the evaluation of the entire array, where the individual elements of the array can be evaluated in parallel.

Part II

Contributions

Chapter 4

Rewriting Cell Arrays

This chapter is based on the paper “**Rewriting High-Level Spreadsheet Structures into Higher-Order Functional Programs**” [14] which is joint work with Wensheng Dou and Peter Sestoft.

4.1 Introduction

The lack of abstraction in spreadsheet formula languages encourages formula copying across columns and rows [7, 82]. In fact, Dou et al. [41] report that 69% of all spreadsheets with formulas in the Enron [56] and EUSES [46] spreadsheet corpora contain *cell arrays*. A cell array is a rectangular block of copy equivalent formulas [82], like the cell areas B2:F6 and B8:F12 in Fig. 4.2. 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.

Can we exploit the presence of cell arrays to speed up spreadsheet recalculations? In functional languages, disjoint computations on values of an array can be expressed explicitly by means of higher-order functions. For instance, the array combinator `map` explicitly applies a pure function to each element of an array individually. Hence, `map` can easily be parallelized (see Sec. 3.1.1).

In this chapter, we design a source-to-source rewriting semantics for converting cell arrays into parallel higher-order functional programs to improve recalculation performance. Our idea is based on the observation that the references in cell arrays often form a pattern that corresponds to one of two combinators on 2D arrays [112, 113].

Our rewriting semantics uses a common feature of spreadsheet software, called *array formulas* (see Sec. 2.1.2). 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.

To our knowledge, there is no previous work on exploiting parallelism in cell arrays to improve recalculation performance. Some researchers have investigated whole-sheet graph parallelism on spreadsheets [110, 112, 113]. Prior work on high-level spreadsheet structures has either focused on making the user aware of high-level models [57, 82, 89]; on correcting errors in cell formulas by analyzing the structure around given cells [32, 39, 41, 59]; or on synthesizing templates from spreadsheets to allow for reuse of the high-level structure [1, 64].

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 idealized and real-world spreadsheets from the LibreOffice Calc [48] benchmark suite and the EUSES [46] corpus. Our results show that, by parallelizing cell arrays on 48 cores, spreadsheet recalculation becomes up to 25 times faster if the spreadsheets are sufficiently large and their computations sufficiently parallel. However, our results also show that the achievable speedup is limited by the sequential dependencies of the spreadsheet models and that, on small spreadsheets, our technique can even make performance worse.

4.2 From Spreadsheets to Higher-Order Functional Programs

In this section, we describe our overall idea: how to rewrite the copy-equivalent formulas of a cell array into a single call to one of two higher-order array combinators, namely MAP and PREFIX, as defined in Sec. 3.2.1. We illustrate our idea by means of a small but complete example, using a specialized formula language, λ -calc.

4.2.1 A Formal Spreadsheet Language

For presentation purposes, we use a formal spreadsheet language, λ -calc, as shown in Fig. 4.1. Terms in e include lambda-expressions of arbitrary arity with named parameters. All expressions must be closed. Users are

only allowed to write expressions in u which is a subset of e without anonymous functions and variable names. References r to cells and to cell ranges are shown in the R1C1 format, but translated to the “usual” A1 format in examples. See Sec. 2.1.1 for a detailed explanation of both address formats.

Function $\phi \in r \rightarrow e$ maps a cell address r to the formula $e = \phi(r)$ in that cell (as described in in Sec. 2.2.3). 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 Sec. 4.3.1).

4.2.2 Example: DNA Sequence Alignment

We illustrate the rewriting of cell arrays with the spreadsheet shown in Fig. 4.2. It computes the optimal local alignment of two DNA sequences using the standard algorithm by Smith and Waterman [98] that is based on dynamic programming. A substitution matrix s is defined in cell range B2:F6 (light-blue cells), and the scoring matrix H in cell range B8:F12 (green cells). 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, we obtain the optimal local alignment of the two sequences.

4.2.3 Intuitive Rewriting of Cell Arrays

The formulas in range B2:F6 are copy equivalent [82]: the range could be filled by copying the formula from B2 to all other cells in B2:F6, making use of the automatic adjustment of relative row and column references, as described in Sec. 2.1.1. The formulas in B8:F12 are copy equivalent, too. In R1C1 reference format, the range B2:F6 (light blue cells) can be written as:

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

The row- and column-relative structure of the two references builds a cross-product of the column and the row containing the input sequences.

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

Figure 4.1: 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)

Figure 4.2: 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 the same background color are copy-equivalent.

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 location of the formulas 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 two-dimensional MAP over repeated row and column values:

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

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 B2:F6 (R2C2:R6C6), as described in Sec. 2.1.2.

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 $n \times \text{COLUMNS}(x)$ 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 element-wise by applying f to pairs of their respective elements.

Concretely, the new expression extends the one-dimensional ranges B1:F1 and A2:A6 into two matrices of size 5×5 and combines them element-wise using the function originally written in each cell. All three functions are described in detail in Sec. 3.2.1.

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.

4.2.4 Different Kinds of Cell Arrays

Now consider the cell array B8:F12 (green cells), which contains the following formula in R1C1 format:

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

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 using MAP, as in Sec. 4.2.3. Hence, we need to target the other array combinator, namely PREFIX:

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

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 (see Sec. 3.2.1), as we can see in our rewritten expression above. Hence, we must order the variable names correctly.

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

4.3 Systematically 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 parameters of 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 an array combinator, 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 $R[\emptyset]C[-1]$, $R[-1]C[-1]$ or $R[-1]C[\emptyset]$, referring to the same row, previous column; previous row, previous column; or previous row, same column. It is straightforward to implement rules for rotated and mirrored cases via array reversal using HREV and VREV in either dimension, or both.

4.3.1 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 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 [82].

The meta-function *lookup* converts relative cell references (first argument) into absolute cell references by adding the row- and column-offset to their own location in the sheet (second argument):

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

A cell reference is intransitive if it never refers back into the cell array, no matter the location of the cell that contains it. 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.

4.3.2 Rewriting Semantics

We use an evaluation context \mathcal{L} (see Fig. 4.3) and a reduction relation \rightsquigarrow to formalize the rewriting process [43]. The \rightsquigarrow relation in Fig. 4.4 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. 4.3.

We introduce the meta-functions *rows* and *columns* that both take a cell range expression $r:r$ and return its number of rows or columns, respectively. This allows us to access the size of a cell array that is currently being rewritten without introducing a cyclic dependency.

$$\begin{aligned}
l &::= v \mid x \\
&\mid l \oplus l \\
&\mid \text{IF}(l, l, l) \\
&\mid \text{F}(l, \dots) \\
&\mid \text{RiCi} && \text{Only absolute references.} \\
&\mid \text{RiCi}:\text{RiCi} \\
\\
\mathcal{L} &::= \circ && \text{Hole.} \\
&\mid \mathcal{L} \oplus u && \text{Expressions are lifted from left to right.} \\
&\mid l \oplus \mathcal{L} \\
&\mid \text{F}(l, \dots, \mathcal{L}, u, \dots) \\
&\mid \text{IF}(\mathcal{L}, u, u) \\
&\mid \text{IF}(l, \mathcal{L}, u) \\
&\mid \text{IF}(l, l, \mathcal{L}) \\
\\
\Gamma &::= \text{done}(\phi(r:r) := \{e\}) \\
&\mid \text{more}(\underbrace{[(r, x) \dots]}_{\text{Transitive}}; \underbrace{[(r, x) \dots]}_{\text{Intransitive}}; \phi(r:r) := \mathcal{L})
\end{aligned}$$

Figure 4.3: Rewriting context \mathcal{L} and transformation language for λ -calc. Terms in l are a subset of terms in e with only absolute cell references.

A term in 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.

We write

$$\Gamma(\mathcal{L}[u]) \rightsquigarrow \Gamma'(\mathcal{L}[k])$$

to say that the reduction relation \rightsquigarrow reduces u to l in a context \mathcal{L} in the rewriting state Γ that we update to Γ' . We use ellipses to denote zero or more repetitions of the leading term.

In plain English, the rules in Fig. 4.4 perform the following operations:

- Rule [EXIST-I] replaces a cell reference r with an already existing variable name x from the list of intransitive substitutions.

- Rule [EXIST-T] replaces a cell reference r with an already existing variable name x from the list of transitive substitutions.
- Rule [SUBST-I] replaces an intransitive cell reference r with a fresh variable name x and records the substitution (r, x) in the list of intransitive substitutions.
- Rule [SUBST-T] replaces a transitive cell reference r with a fresh variable name x and records 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 parameters are the variable names from the list of intransitive substitutions in Γ . 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 at r_{ul} and r_{lr} for each of them and extended to match the size of the cell array. 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 in Γ . 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 at 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 rules [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 $sr_1:r_2$ that is being rewritten:

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

Finally, rule [SYNTH-PFX] uses the meta-function *fill&sort*. Recall that the PREFIX combinator requires that the argument function accepts three arguments for the value from the previous column, same row; the previous “diagonal” value; and the value from the previous row, same column (see Sec. 3.1.4). Hence, we require in [SYNTH-PFX] that there are three transitive substitutions in the order $R[0]C[-1]$, $R[-1]C[-1]$ and $R[-1]C[0]$. The meta-function *fill&sort* (1) generates placeholder substitutions for each not encountered transitive reference and (2) sorts the three substitutions after their respective references to match the requirements of the PREFIX combinator.

4.3.3 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 of the 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 straightforward 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. 4.5 leads to the creation of cyclic dependencies.

To avoid this, we perform a preemptive detection of cyclic references. We walk the dependency 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.

For better performance, we can apply a simplifying heuristic. Instead of traversing the dependency graph for each cell of a cell array, it is often sufficient to only do so for the four corners of a cell array. This is possible thanks to the often highly regular structure of spreadsheet models.

4.3.4 Correctness

We do not currently have a formal proof of correctness for our rewriting semantics. However, the slightly informal semantics for MAP and PREFIX (see Sec. 3.1.1 and 3.1.3) directly match those of the original cell array

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

Figure 4.4: The \rightsquigarrow relation for rewriting cell array formulas in λ -calc. The rules are explained in detail in Sec. 4.3.2.

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

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

Figure 4.5: A spreadsheet (top) whose rewritten variant (bottom) contains cyclic dependencies. Rewriting the cell arrays A1:A3 and B2:B3 using the rewriting relation \rightsquigarrow 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.

structure, so we believe that our rewriting semantics are correct. Moreover, we have formalized our rewriting semantics using PLT Redex [44] (see Appendix B) and performed extensive testing.

The proof would require a formal semantics for spreadsheet recalculation and arrays combinators, which at the time of writing was out of the scope of the original paper [14]. Even though we have defined an evaluation semantics for a “standard” spreadsheet formula language (see Sec. 2.2.3), we cannot use it for a correctness proof: it does not formalize array formulas. Without defined semantics for array formulas, the correctness proof cannot succeed.¹

With a complete evaluation semantics, however, we believe that we can show that spreadsheets with rewritten cell arrays behave 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 $\phi(r_1:r_2) := u \Downarrow v$ and

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

then we want to show that $\phi(r_1:r_2) := \{e\} \Downarrow v$ for all e and r_1, r_2 .

4.3.5 Extension to Cell Range Expressions

In real-life spreadsheets, cell arrays often include references to cell ranges, such as =SUM(A1:B1) to compute the sum of the values in the same row

¹At the time of writing, a technical report that formulates a complete operational semantics of Funcalc including array formulas is underway.

	A	B	C
1	3	1	=SUM(A1:B1)
2	6	1	=SUM(A2:B2)
3	9	1	=SUM(A3:B3)

Figure 4.6: A spreadsheet with a relative cell range expression in column C.

as the formula cell, as illustrated in Fig. 4.6. Assuming that the cell array is in column C, this expression translates to R1C1 format as follows:

$$\phi(R1C3:R3C3) := \text{SUM}(R[0]C[-2]:R[0]C[-1])$$

Our rewriting semantics (as defined in the original paper [14]) has until now ignored cell arrays with cell range expressions.

We can adapt the rewriting semantics by observing that the cell array actually maps SUM to rows of the cell range A1:B3. We can use the SLICE function (see Sec. 3.2.1) to extract single rows (or columns) from arrays. This means that the λ expression that we generate during rewriting must take as arguments a row and a column index; then, we can use it together with the TABULATE combinator to create an array formula expression.

In the following, we sketch an extension to the \rightsquigarrow relation from Sec. 4.3.2 where we replace a cell range expression with a call to SLICE for the reserved names x_r and x_c that stand for the row and column index, respectively. If an expression only contains a cell range sub-expression, \rightsquigarrow will make sure to use TABULATE with the resulting λ expression. If the expression contains one or more cell range expressions and one or more cell references, then it chooses the MAPI combinator: a combination of TABULATE and MAP, where the argument function also accepts the current row and column index as parameters.

The rule for substituting a cell range is defined as follows:

$$\begin{aligned}
\text{more}([(r^T, x^T) \dots]; [(r^I, x^I) \dots]; \phi(r_{ul}:r_{lr}) &:= \mathcal{L}[r_1:r_2] \rightsquigarrow & [\text{SUBST-A}] \\
\text{more}([(r^T, x^T) \dots]; [(r^I, x^I) \dots]; \phi(r_{ul}:r_{lr}) &:= \mathcal{L}[\text{SLICE}(r_{1ul}:r_{2lr}, & \\
& \quad x_r, & \\
& \quad x_c, & \\
& \quad \text{rows}[[r_{1ul}:r_{2ul}]], & \\
& \quad \text{columns}[[r_{1lr}:r_{2lr}]])) & \\
\text{where } r_{1ul} = \text{lookup}[[r_1, r_{ul}]] & & \\
r_{2ul} = \text{lookup}[[r_2, r_{ul}]] & & \\
r_{1lr} = \text{lookup}[[r_1, r_{lr}]] & & \\
r_{2lr} = \text{lookup}[[r_2, r_{lr}]] & &
\end{aligned}$$

Rule [SUBST-A]

1. infers the size of the target cell range for the cell array by looking up the upper-left and lower-right corners at the upper-left and lower-right corners of the cell array (r_{1ul} and r_{2lr});
2. computes the shape of the slice (i.e. its rows and columns) by looking up its upper-left and lower-right corners also at the lower-right and upper-left corners (r_{1lr} and r_{2ul}) of the target array—this infers the correct slice shape also for partially absolute cell ranges (e.g. $R1C1:R[0]C[-1]$); and
3. injects the inferred target cell range and the inferred shape into a call to SLICE that starts at row x_r and column x_c , which are the aforementioned reserved variables for row and column indices.

Finally, we can formulate a rule that plugs a lifted cell range expression into a call to TABULATE:

$$\begin{aligned}
\text{more}([]; []; \phi(r_{ul}:r_{lr}) &:= \mathcal{L}[l] \rightsquigarrow & [\text{SYNTH-TAB}] \\
\text{done}(\phi(r_{ul}:r_{lr}) &:= \{ \text{TABULATE}(\lambda(x_r, x_c).l, \text{rows}[[r_{ul}:r_{lr}]], \text{columns}[[r_{ul}:r_{lr}]])) \} &
\end{aligned}$$

Rule [SYNTH-TAB] generates an expression that, when evaluated, instantiates the lifted expression that slices over a cell range for each row and column index of the array formula. This rule requires that the original formula expression does not contain any other relative cell addresses.

It is possible to define a rule that combines the behavior of rules [SYNTH-MAP] and [SYNTH-TAB] using a function MAPI that accepts a function that takes index parameters as well as values. We omit a rule

for synthesizing a call to `MAPI` for brevity; such a rule would require additional modifications to the lifting language.

By applying the rules `[SUBST-A]` and `[SYNTH-TAB]`, the cell array from Fig. 4.6 rewrites to:

$$\phi(R1C3:R3C3) := \{\text{TABULATE}(\lambda(x_r, x_c).\text{SUM}(\text{SLICE}(R1C1:R3C2, x_r, x_c, 1, 2)))\}$$

4.4 Implementation

We have implemented the rewriting semantics from Sec. 4.3 in `Funcalc` [93]. Instead of writing our own detection of cell arrays, we piggyback on `Funcalc`'s algorithm for rebuilding the support graph [93, Sec. 4.2.9], a heuristic algorithm for detecting rectangular blocks of copy-equivalent cells that runs in linear time in the number of cells in the cell array.

4.4.1 Differences to λ -calc

The λ -calc language differs from `Funcalc` in that it allows anonymous functions, i.e. lambda-expressions, where `Funcalc` does not. The only way to define new functions in `Funcalc` is via sheet-defined functions (see Sec. 2.3). Superficially, this is not a problem: we can just generate a new SDF on the reserved function sheet `*Generated*` and assign unique names to generated SDFs by counting, i.e. `GEN0`, `GEN1`, and so on.

Unfortunately, doing so naively leads to a change to the recalculation semantics of the spreadsheet. When we rewrote cell arrays in λ -calc, we implicitly assumed that the generated anonymous function is reevaluated as part of the formula expression whenever the dependencies of the formula are updated. This includes dependencies via absolute cell references. In `Funcalc`, however, generating a new SDF will remove the absolute reference from the formula expression: it is now part of the body of the generated SDF. Sheet-defined functions are not part of the dependency graph in `Funcalc`. Hence, the explicit dependency between the cell array and the cells that it refers to via absolute references are removed from the dependency and support graph. Changing the support graph in such a way is problematic for minimal recalculation (see Sec. 2.2.1). In Fig. 4.7, if cell A1 were to be changed, this would trigger reevaluation of the cell array in B2:C2 in λ -calc; but in `Funcalc`, because the explicit dependency has been removed, the spreadsheet would enter

	A	B	C
1	11	22	33
2		=\$A\$1+B1	=\$A\$1+C1

λ -calc:

	A	B	C
1	11	22	33
2		={MAP($\lambda(x).$ A1+x, B1:C1)}	

Funcalc:

	A	B	C
1	11	22	33
2		={MAP(CLOSURE("GEN0"), B1:C1)}	

Figure 4.7: Differences of rewriting a cell array (top) in λ -calc (center) and a naive approach to cell array rewriting in Funcalc (bottom). In λ -calc, absolute references remain part of the expression and therefore also in the support graph. In Funcalc, the absolute reference disappears into the body of the generated SDF GEN0, which corresponds to $\lambda(x).$ A1+x.

an inconsistent state and violate the consistency requirement described in Sec. 2.2.4.

Our implementation of cell array rewriting in Funcalc circumvents this problem by using the partial application feature of the CLOSURE function (see Sec. 2.3.2). The general rule is that every reference to a cell or a cell range in the expression is replaced by a variable name during rewriting, including absolute cell references. If a cell reference is absolute, we bind the absolute reference to the newly generated closure directly. Hence, the correct expression for the rewritten array in Fig. 4.7 that preserves the dependency graph is

$$={\text{MAP}}(\text{CLOSURE}(\text{"GEN0"}, \text{A1}, \text{NA}()), \text{B1:C1})\}$$

where the generated SDF GEN0 corresponds to $\lambda(x,y).x+y$. We use a similar approach for cell range expressions. For a cell array that references cell ranges, we compute the cell range that subsumes all partial cell ranges and apply it partially to the generated function. The generated function will then slice it as described in Sec. 4.3.5.

4.4.2 Handling Over-Generalization

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

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

In real-world spreadsheets, it may happen that a transitive reference has a stride larger than one, but the PREFIX combinator and its variants do not generalize to such references. Hence, we cannot directly rewrite cell arrays with transitive cell references of a stride larger than one.

A stride larger than one seems to be an artifact of the generality of the support graph rebuilding algorithm [93, Sec. 4.2.2]. Our key observation here is that one can turn transitive cell references into intransitive cell references by splitting 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 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 we reach a lower limit on cell array size, in which case we abort.

4.4.3 Parallelization Strategies

Since Funcalc runs on the .NET platform, we use the parallelization mechanisms from the Task Parallel Library [74]. The TPL achieves parallelization via a work-stealing thread pool [74]. We can parallelize MAP by iterating over either rows or columns in a parallel for-loop. Parallelizing the PREFIX combinator is implemented exactly as described in Sec. 3.1.4 using recursive task spawning.

4.5 Performance

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. Since we consider cell array rewriting a one-time operation, times for rewriting are not included in the benchmarks and we report them separately. The raw data are available online.²

²<https://github.com/fbie/funcalc-array-benchmarks>

We report the average speedup of 30 full recalculations of the entire spreadsheet on the P3 server (see Sec. 1.3). 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 ran three warm-up iterations which we do not count prior to benchmarking.

4.5.1 Spreadsheet Selection

Idealized Spreadsheets We use two idealized spreadsheets to measure the effect of rewriting transitive and intransitive cell arrays in isolation. Each spreadsheet contains a single cell array of size $n \times n$ and we instantiate each spreadsheet once for $n = 200$ and once for $n = 400$. The first spreadsheet contains a single intransitive cell array that applies the FIB function (see Sec. 2.3.1) on each input cell. The second spreadsheet computes a summed-area table by applying the FIB function to each input cell and adding it to the sum of the prefix. The input cells contain the constant 22; computing FIB(22) takes roughly 1.5ms.

LibreOffice Calc Spreadsheets LibreOffice Calc [48] provides a set of large, real-world benchmark spreadsheets, which is available online.³ These spreadsheets contain large cell arrays and are realistically structured. Funcalc’s formula syntax differs from that of Microsoft Excel and LibreOffice Calc in a number of ways that require modifications of the spreadsheets. To be able to run the spreadsheets in Funcalc, we have implemented some standard Excel and Visual Basic functions as sheet-defined functions. Recalculating LibreOffice Calc spreadsheets sequentially takes between 32 and 168 seconds.

EUSES Spreadsheets Finally, we use real-world spreadsheets from the EUSES spreadsheet corpus [46]. We have selected 21 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. The Funcalc-compatible spreadsheets from the EUSES corpus are available online.⁴ Sequential

³<https://gerrit.libreoffice.org/gitweb?p=benchmark.git>

⁴<https://github.com/popular-parallel-programming/funcalc-euses>

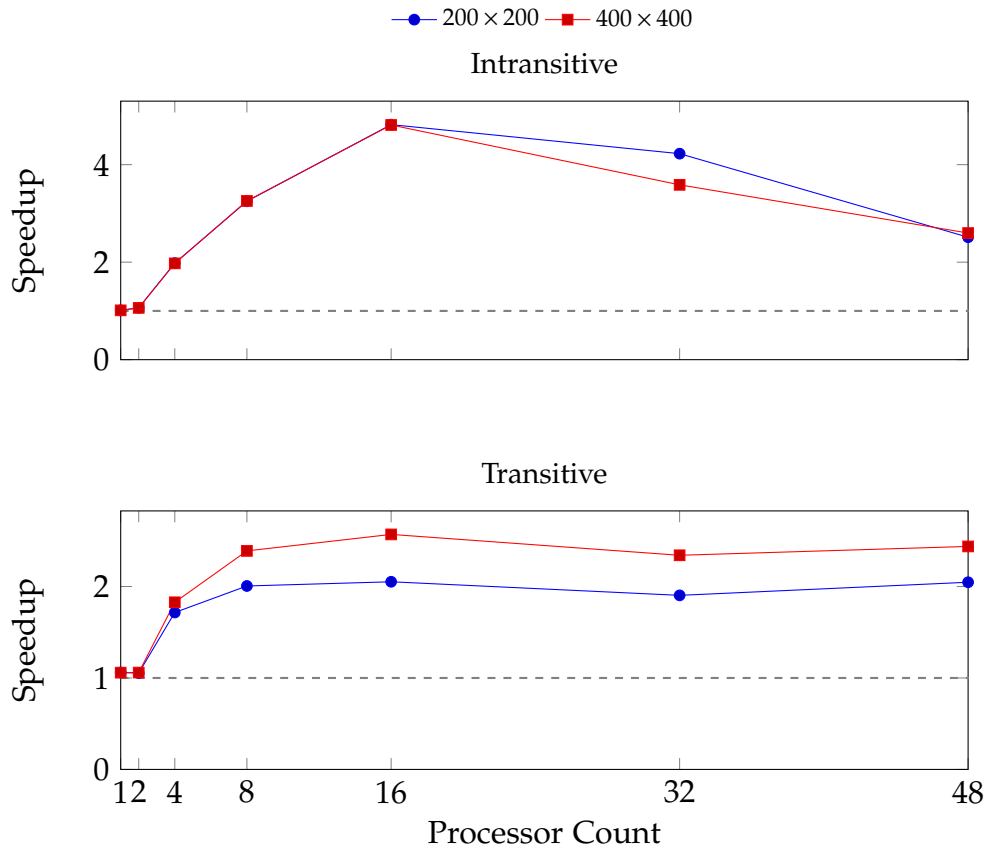


Figure 4.8: Average speedup for rewriting idealized spreadsheets. Values are speedup factors over sequential full recalculation without rewriting, averaged over 30 full recalculations; higher is better. The gray dashed line indicates 1-core performance.

Funcalc recalculates all selected EUSES spreadsheets in less than 100 milliseconds on average, so we expect little to no speedup.

4.5.2 Results

Idealized Spreadsheets

Figure 4.8 shows recalculation performance relative to sequential recalculation after rewriting idealized spreadsheets with only intransitive or only transitive cell references.

Parallelizing intransitive cell arrays with an average of 1.5ms per cell achieves more than four times speedup on 16 cores, but performance declines afterwards. There are no dependencies between the single cells,

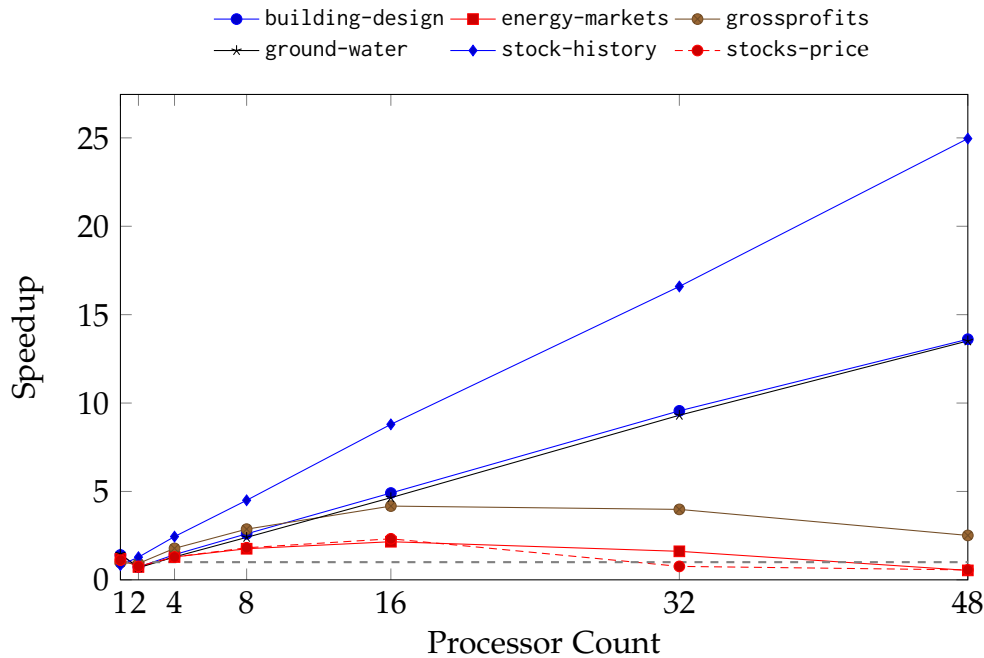


Figure 4.9: Performance results for rewritten LibreOffice Calc spreadsheets. Values are average speedup factors of 30 iterations over sequential full recalculation of the unchanged spreadsheets; higher is better.

so it is reasonable to assume that the communication cost of work-stealing between chips dominates parallel performance.

Parallelizing transitive cell arrays produces at most a 2.5-fold speedup on 16 cores; since the speedup is rather small, the cost of inter-chip communication for more than 16 cores affects performance less.

LibreOffice Calc Spreadsheets

Figure 4.9 shows the speedup for rewriting and parallelizing LibreOffice Calc spreadsheets. Overall, we achieve good speedups, with a maximum of roughly 25 times speedup on stock-history. We achieve the best speedup on average at 16 cores. Inter-chip communication cost seems to not affect all spreadsheets equally.

Table 4.1 shows how many cell arrays per spreadsheet have been rewritten. Overall, our rewriting technique is able to rewrite nearly all cell arrays in the LibreOffice Calc spreadsheets. It rewrites only 20 out of 22 cell arrays in stock-history; nevertheless, this is the spreadsheet whose performance scales nearly linearly. At the same time,

Spreadsheet	Found	Intrans.	Trans.	Rewrite (s)
building-design	6	6	0	56.14
energy-markets	76	76	0	44.58
grossprofits	9	9	0	110.1
ground-water	12	12	0	126.9
stock-history	22	20	0	235.19
stocks-price	8	8	0	35.6

Table 4.1: Statistics for rewriting LibreOffice Calc spreadsheets at load-time. Note that none of the spreadsheets contains transitive cell arrays. Cell arrays are only rewritten if they contain at least 64 cells. Rewriting times are in seconds and averaged over 30 runs. Standard deviation is less than 0.01ms for each spreadsheet.

both energy-markets and stocks-price exhibit the lowest speedup, even though all of their cell arrays have been rewritten. This indicates that the sheer number of rewritten cell arrays is not a useful predictor for performance. None of the LibreOffice Calc spreadsheets contains transitive cell arrays of more than 64 cells and that can safely be lifted without introducing a cyclic dependency.

Moreover, Table 4.1 shows that our rewriting algorithm is rather slow. We believe that this is due to preemptively checking for cyclic dependencies, as described in Sec. 4.3.3, as well as to actually having to traverse the spreadsheet again and update the cells after rewriting. In addition to that, rewriting triggers the SDF compiler, which also may add some overhead.

EUSES Spreadsheets

Figure 4.10 shows the speedup of parallelizing rewritten EUSES spreadsheets. Overall, we achieve only little speedup over sequential recalculation, as expected for small spreadsheets.

Moreover, the achievable speedup is bound by Amdahl’s law [54, Sec. 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. Unless rewriteable cell arrays either dominate the spreadsheet, as in PLANCK, the overall performance will be determined by the sequential computations.

Table 4.2 shows the number of transitive and intransitive cell arrays that could be rewritten for the EUSES spreadsheets. There are five spread-

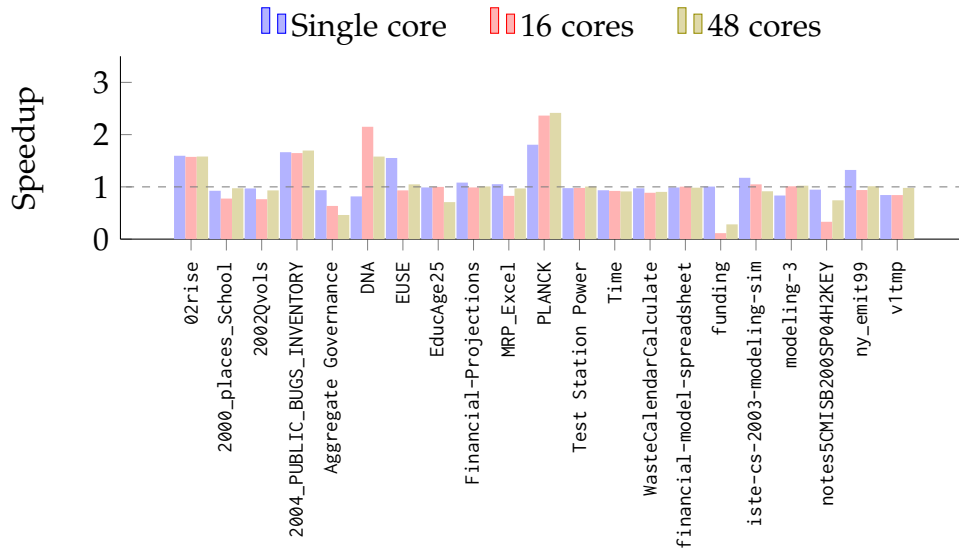


Figure 4.10: Average benchmark results for full recalculation of 21 systematically rewritten EUSES spreadsheets [46] over 50 runs. Values are speedup factors over sequential performance for the original spreadsheets on the same machine; higher is better. The dashed gray bar indicates baseline single-core performance.

sheets out of 21 that could not be rewritten at all, either due to the absence of cell arrays that contain more than 64 cells or because rewriting would have introduced a cyclic dependency. There is only one spreadsheet, `MRP_Excel`, that contains transitive cell arrays that could be rewritten. We conjecture that transitive cell arrays are a rather rare phenomenon in real-world spreadsheets, but our study is not representative.

It is notable that we already achieve a speedup of only rewriting cell arrays without parallelizing them. The spreadsheets `02rise`, `2004_PUBLIC_BUGS_INVENTORY`, `EUSE` and `PLANCK` can be recalculated about 50% faster when their cell arrays have been rewritten. This shows that rewriting is a kind of spreadsheet specialization: recalculation does not have to pay for repeatedly interpreting an expression; instead, it gets lifted into a generated SDF, which is then compiled and hence can be executed much faster when applied repeatedly.

Spreadsheet	Found	Intrans.	Trans.	Rewriting (ms)
02rise	273	1	0	98.33
2000_places_School	40	0	0	1.81
2002Qvols	32	5	0	7.95
2004_PUBLIC_BUGS_INVENTORY	5	4	0	78.73
Aggregate Governance	36	31	0	129.64
DNA	4	4	0	24.65
EducAge25	42	0	0	2.21
EUSE	8	4	0	0.18
financial-model-spreadsheet	191	0	0	40.61
Financial-Projections	78	4	0	139.9
funding	57	7	0	8
iste-cs-2003-modeling-sim	40	2	0	6.23
modeling-3	7	0	0	4.5
MRP_Excel	82	7	6	—
notes5CMISB200SP04H2KEY	4	3	0	15.26
ny_emit99	68	65	0	1954.65
PLANCK	3	2	0	6.86
Test Station Power	106	9	0	0.58
Time	73	1	0	18.29
v1tmp	35	0	0	539.62
WasteCalendarCalculate	9	4	0	0.08

Table 4.2: Statistics for rewriting EUSES spreadsheets. Spreadsheet MRP_Excel is the only spreadsheet that contains transitive cell arrays. Cell arrays are only rewritten if they contain at least 64 cells. Rewriting times are in milliseconds and averaged over 30 runs. Standard deviation is less than 0.14ms for each spreadsheet.

4.6 Alternative Usages and Related Work

Neither parallel recalculation of spreadsheets nor high-level structure analysis is a new idea. To our knowledge, however, no prior work has combined both in a practical application of functional programming. Hermans and Dig [55] implemented a rule-based rewriting system for formula refactoring to replace awkward expressions, such as `IF(A1<B1,A1,B1)`, with more canonical expressions, e.g. `MIN(A1,B1)`. Their system allows users to extend the set of rewrite rules within the spreadsheet, but does not allow inter-cell formula rewriting.

Wack [110] focused on a dataflow approach to whole-spreadsheet parallelization, in contrast to our idea that harnesses local array parallelism.

Yoder and Cohn [112, 113] investigate spreadsheets from a theoretical point of view, also with dataflow parallelism in mind. They observe that high-level array programming intuitively maps to spreadsheets [113]; 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 [82] whose definition is closely related to cell arrays, or by drawing dataflow diagrams [57] to illustrate relations between sheets and between 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 cell arrays is related to template synthesis from spreadsheets. Isakowitz et al. [64] 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 individual cell arrays.

Others [32, 40, 59] focus on detecting clones of cell arrays or tables on the same spreadsheet, which is, again, a whole-sheet analysis.

4.7 Conclusion

In this chapter, we presented a rewriting semantics to rewrite cell arrays that consist of copy equivalent cells to higher-order functional expressions using array combinators. We are able to exploit the implicit parallelism of these rewritten cell arrays and achieve a maximum of 25 times parallel speedup on 48 cores for spreadsheets where cell arrays dominate.

There are drawbacks to our technique. Naively rewriting all cell arrays in a spreadsheet can introduce cyclic references and hence change the semantics of the original spreadsheet. Detecting possible cyclic dependencies before rewriting comes at the cost of traversing (at least parts 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.

The main shortcoming of our work is the lack of a formal proof that our rewriting semantics preserves the semantics of the cell array.

Our experimental results show that only spreadsheets consisting of large cell arrays achieve good speedups. Moreover, our current rewriting algorithm is slow. This suggests that our rewriting approach should not be applied automatically. Instead, it could be implemented as a tool for expert spreadsheet developers that should be applied manually and judiciously. We believe that there is room for improving the performance of our rewriting algorithm, in particular preemptive cycle detection.

Chapter 5

Spreadsheet Dataflow Parallelism

This chapter is based on the paper “**Puncalc: Task-Based Parallelism and Speculative Reevaluation in Spreadsheets**” [12] which is joint work with Alexander Asp Bock.

5.1 Introduction

In Chapter 4, we showed how local parallelism can be exploited by means of statically rewriting cell arrays in spreadsheets. A shortcoming of this technique is that two or more independent cell arrays still are evaluated sequentially, even though there possibly is more parallelism that can be extracted. Moreover, cell array rewriting requires a *static* check for cyclic dependencies ahead of recalculation.

In this chapter, we present an algorithm for parallel evaluation of spreadsheets using a *dynamic* dataflow approach. We can interpret the support graph as a dataflow graph [112] and use it to parallelize the evaluation of independent cells. If a cell that has just been evaluated supports two or more cells, then these two cells can be computed in parallel, if there is no direct dependency between them. Our algorithm detects such cases dynamically during recalculation and hence adds no overhead of statically analyzing the spreadsheet ahead of time.

The main challenge of such a dynamic approach is to detect cyclic dependencies correctly without creating deadlocks. Our algorithm uses speculative evaluation to prevent deadlocks and to detect cyclic dependencies. The main advantage of this approach is that we do not have to perform a more strict detection of cyclic dependencies than e.g. Microsoft Excel does.

We have implemented this algorithm in the experimental spreadsheet engine Funcalc [93]. Our key contributions are:

- a parallel, task-based, topology-agnostic algorithm for minimal recalculation of spreadsheets;
- a method for detecting cyclic references during parallel recalculation using speculative reevaluation;
- a thread-local evaluation optimization that exploits a specific spreadsheet topology on the fly; and
- an evaluation of our algorithm on a set of benchmarks for different types and sizes of spreadsheets with different characteristics and topologies.

To our knowledge, no such algorithm for parallel evaluation of spreadsheets with dynamic cycle detection has previously been proposed. Experiments show that we achieve between 1.4 and 6.5 times speedup on 16 cores and nearly a 16-fold speedup on 48 cores.

The C# code in the remainder of this chapter uses LinQ extension methods [94, Sec. 28.5].

5.2 Funcalc: Sequential Implementation

In this section, we describe a simplified variant of the minimal recalculation algorithm as implemented in sequential Funcalc [93].

Minimal recalculation is a breadth-first traversal of the support graph. Funcalc uses a global work queue to maintain all cells that have been encountered during traversal and that have not yet been computed. The minimal recalculation algorithm `RecalculateMinimal()` in Fig. 5.1 (1) recursively marks the cells reachable from the recalculation roots (see Sec. 2.2.2) dirty by setting the state of all encountered cells to `Dirty` (method `MarkDirty()` in Fig. 5.3); (2) adds the recalculation roots to the global work queue; and (3) dequeues cells from the head of the queue and evaluates them by calling `Eval()` on them until the queue is empty (Line 11).

In Funcalc, cells are assigned a *cell state*, which is either `Dirty`, `Enqueued`, `Computing` or `Uptodate`. During a single recalculation, cell state changes only monotonically—e.g. a cell cannot go back from `Uptodate` to

```

1 public class Workbook {
2     public readonly Queue<Cell> queue;
3     readonly List<Cell> roots;
4
5     public void RecalculateMinimal() {
6         // Mark cells reachable from roots dirty.
7         foreach (Cell c in roots) { c.MarkDirty(); }
8         // Enqueue recalculation roots.
9         foreach (Cell c in roots) { c.Enqueue(); }
10        // Evaluate via support graph.
11        while (queue.Count > 0)
12            queue.Dequeue().Eval();
13    }}

```

Figure 5.1: A simplified Workbook class.

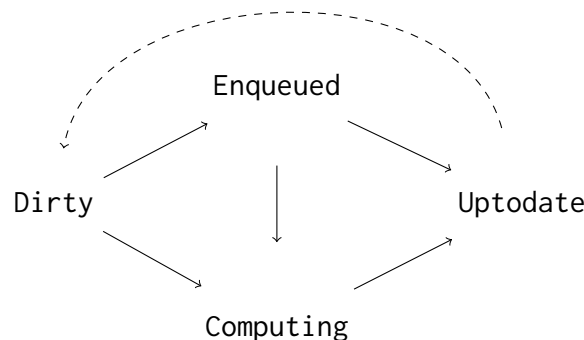


Figure 5.2: The possible state transitions of a cell. The dashed connection denotes marking dirty the cells reachable from the recalculation roots at the beginning of a particular recalculation.

Computing, according to the state transitions in Fig. 5.2. During recalculation, the cell state indicates whether a cell should be evaluated anew. Moreover, cells cache the value that their formula expression evaluates to. A cell that is Uptodate has cached its most recent value.

Figure 5.3 details the Formula class that models formula cells. During the evaluation of a formula, its state is Computing (Line 34). After evaluation, the evaluation method Eval() updates the formulas value cache and sets its state to Uptodate (Line 35ff.). It enqueues supported cells to the global work queue wb.queue if their state is Dirty and changes

their state to `Enqueued` correspondingly by calling `Enqueue()` on them (Line 37).

If a cell is being evaluated and one of its dependencies `d` is not yet `Uptodate`, the call to `e.Eval()` in Line 35 will recursively evaluate the dependency by calling `d.Eval()`. If the state of the dependency is `Computing`, recalculation has detected a cyclic reference.

If the global work queue is empty, all cells in the spreadsheet are `Uptodate`.

5.3 Puncalc: Parallel Implementation

Puncalc, short for *parallel Funcalc*, is a parallel variant of Funcalc based on the .NET Task Parallel Library (TPL) [74] using a work-stealing thread pool.

In the following sections, we introduce thread safety requirements on the underlying mutable state of cells; our approach to parallel, minimal recalculation and cycle detection; and how Puncalc complies with the consistency requirement from Sec. 2.2.4. We then extend the algorithm with a thread-local optimization that exploits a specific spreadsheet topology on the fly.

5.3.1 Thread Safety

Conceptually, Funcalc is a strictly evaluated, purely functional language. However, the implementation uses mutable state to make the language efficient, which we must make thread safe to enable correct parallelization: the global recalculation queue in `Workbook` must be thread safe; formula cells *cache* the result of their evaluation, and all threads should agree on the cached result due to the consistency requirement from Sec. 2.2.4; a cell also has a *cell state* that should be consistent among all threads; and each formula cell should only be evaluated once. We will relax the latter requirement in Sec. 5.3.3.

We handle the underlying mutable state of formula cells using the following scheme. If multiple threads try to evaluate the same cell, one thread takes *ownership* of the cell and sets the cell state to `Computing`. We call the thread that set the cell to `Computing` the cell's owner. The thread will then evaluate the formula expression, write the result to the value cache and finally set the cell state to `Uptodate`. The remaining threads

```
1  enum CellState {
2      Dirty,
3      Enqueued,
4      Computing,
5      Uptodate
6  }
7
8  public class Formula : Cell {
9      readonly Expr e; readonly Workbook wb;
10     CellState state; Value cached;
11     public readonly List<Cell> supported;
12
13     public void MarkDirty () {
14         if (state != CellState.Dirty) {
15             state = CellState.Dirty;
16             foreach (Cell c in supported) { c.MarkDirty(); }
17         }
18
19     public void Enqueue() {
20         if (state == CellState.Dirty) {
21             state = CellState.Enqueued;
22             wb.queue.Enqueue(this);
23         }
24
25     public Value Eval() {
26         switch (state) {
27             case CellState.Uptodate:
28                 break; // Nothing to do.
29             case CellState.Computing:
30                 throw new Exception("Cycle!");
31             case CellState.Dirty:
32             case CellState.Enqueued:
33                 // Evaluate cell, protect with state.
34                 state = CellState.Computing;
35                 cached = e.Eval();
36                 state = CellState.Uptodate;
37                 foreach (Cell c in supported) { c.Enqueue(); }
38                 break;
39         }
40         return cached;
41     }}
```

Figure 5.3: A simplified Formula class and its Eval() method.

block until the cell is Uptodate and then read the result of evaluating the formula from the value cache.

We could use intrinsic locks on cells to implement this scheme, but as it turns out, locking on cells comes at the cost of (1) performance: threads that wait for locks are *de-scheduled*, but often times, the result of evaluating the formula will be available soon, since the average computation time per cell is usually rather low, making de- and re-scheduling a waste of time; and (2) correctness: it is legal to create cyclic references in spreadsheets (see Sec. 2.2.1), but such cyclic references can lead to deadlocks. Suppose thread t_1 locks cell c . When another thread t_2 examines c , it sees that c is locked by thread t_1 and blocks. If threads t_1 and t_2 both evaluate cells that are part of a cycle, they will wait for each other and deadlock.

Instead of using locking, we implement our scheme using *compare and swap* on cell state. Its semantics can be described as follows [54, Sec. 5.8]:

```
public static bool CAS<T>(ref T r, T v, T c) {
    if (r == c) { // Is r equal to c?
        r = v;    // If yes, update r to v.
        return true; // Report success.
    } else
        return false; // Otherwise, report failure.
}
```

Method CAS() checks whether the value of the reference r is equal to some comparand c . If yes, it updates r to value v and returns true. Otherwise, it returns false and performs no update. It does all of that *atomically*, i.e. in a single CPU instruction, such that no other threads can interfere.¹

Class MaybeThreadSafeFormula in Fig. 5.4 implements our thread safety scheme. Note that all reads from and writes to state are lock-free; calls to Eval() are lock-free if the formula cell is Uptodate.

The overall idea of our recalculation algorithm is to let threads compete for setting the cell state to Computing using CAS() (Line 24). The thread that wins the race proceeds with evaluating the formula expression, while the other threads enter a busy-wait loop, waiting for the result of the evaluation to become available.

We try to minimize races when it comes to accessing the support set of a cell. Only the thread whose call to CAS() in Line 24 succeeded is

¹We implement CAS in C# via Interlocked.CompareExchange().

```

1  class MaybeThreadSafeFormula : Cell {
2      readonly Expr e; readonly Workbook wb;
3      volatile CellState state; Value cached;
4      public readonly List<Cell> supported;
5
6      public void MarkDirty() { ... }
7
8      public void Enqueue() {
9          if (CAS(ref state,           // Atomically update state
10              CellState.Enqueued, // to Enqueued and enqueue
11              CellState.Dirty))    // only if successful.
12              wb.queue.Enqueue(this);
13      }
14
15      public Value Eval() {
16          int s = state; // Store locally for consistency.
17          switch (s) {
18              case CellState.Uptodate:
19              case CellState.Computing:
20                  break; // Another thread may be evaluating.
21              case CellState.Dirty:
22              case CellState.Enqueued:
23                  // Attempt to take ownership and evaluate.
24                  if (CAS(ref state, CellState.Computing, s)) {
25                      cached = e.Eval();
26                      State = CellState.Uptodate;
27                      foreach (Cell c in supported) { c.Enqueue(); }
28                  }
29                  break;
30          }
31          while (state < CellState.Uptodate)
32              ; // Block until state is Uptodate.
33          return cached;
34      }}

```

Figure 5.4: A first attempt at making the simplified Formula class thread safe. The main difference to class Formula in Fig. 5.3 is that critical updates to state are done via CAS(). This implementation cannot detect cyclic dependencies in the Computing case.

allowed to enqueue the supported cells to the global work queue. The cells in the support set may however also be part of some other cell's support set, so there still is a possibility for races. Note that the `Enqueue()` method from Fig. 5.4 makes sure that each cell gets added to the end of the global work queue at most once by using `CAS()` to update the cell state.

This first attempt at implementing a thread safe `Formula` class livelocks when encountering a cyclic reference. We describe an approach to correctly detecting cyclic dependencies without livelocking in Sec. 5.3.5.

5.3.2 Parallel Minimal Recalculation

Figure 5.5 shows the main loop of *parallel* minimal recalculation in method `RecalculateMinimal()` that handles termination. The emptiness of the global work queue (now of type `ConcurrentQueue<Cell>`) alone is no longer a sufficient termination criterion: the queue may be empty while there are still cells being evaluated by other threads, which may in turn enqueue more cells. Therefore, we use a concurrent and scalable atomic `LongAdder` class, inspired by the Java 8 `LongAdder` implementation [86], to keep track of the number of cells currently being evaluated.

Parallel minimal recalculation begins similarly to its sequential counterpart by (1) marking all cells reachable from the recalculation roots `Dirty` and (2) enqueueing the roots and changing their state to `Enqueued`.

If the main thread has successfully dequeued a cell from the queue (Fig. 5.5, Line 17), it increments counter (Line 18) and spawns a new TPL task to compute it. The task evaluates the cell and subsequently decrements counter (Line 25).

The termination condition of the while loop in `RecalculatePartial()` in Fig. 5.5, Line 14 states that it should keep running as long as (1) there is at least one cell being evaluated or (2) queue is not empty; and (3) no cycles have been detected.

It is crucial that the checks for termination (1) and (2) are ordered as they are. Imagine we were to swap (1) and (2) and initially, queue is empty and counter.Value is 1. The time line in Fig. 5.6 illustrates an interleaving of operations that would then cause premature termination of the recalculation algorithm. The main thread t_m would, when evaluating the first half of the termination condition, i.e. check (2), see that queue is empty. Before t_m continues, a worker thread t_w could finish

```

1 class ParallelWorkbook {
2     public readonly ConcurrentQueue<Cell> queue;
3     readonly List<Cell> roots;
4     readonly LongAdder counter;
5
6     public void RecalculateMinimal() {
7         // Mark cells reachable from roots dirty.
8         foreach (Cell c in roots) { c.MarkDirty(); }
9         // Enqueue recalculation roots.
10        foreach (Cell c in roots) { c.Enqueue(); }
11        // Evaluate via support graph.
12        // Continue while work left and no cycle found.
13        bool cycle = false;
14        while ((0 < counter.Value || 0 < queue.Count)
15            && !cycle) {
16            Cell next;
17            if (queue.TryDequeue(out next)) { // Might be empty.
18                counter.Increment(); // Now 0 < counter.
19                Task.Run(() => { // Start a new task to evaluate.
20                    try {
21                        next.Eval();
22                    } catch (Exception e) {
23                        cycle = true; // Notify main loop on cycle.
24                    }
25                    queue.Decrement(); // Now 0 <= counter.
26                });}}}}

```

Figure 5.5: A simplified implementation of parallel recalculation. The main difference to class `Workbook` from Fig. 5.1 is the while-loop in `RecalculateMinimal()`: it only terminates if there are no cells currently being evaluated and no cells left in the queue. Moreover, it starts a new TPL task for each cell it pops from the work queue.

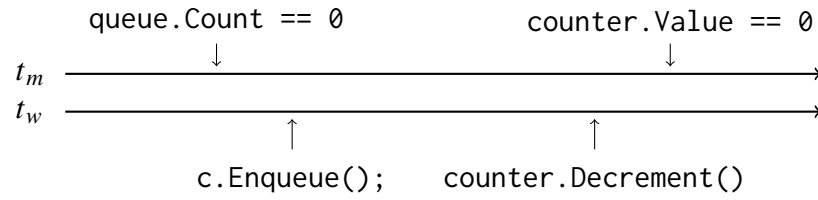


Figure 5.6: A time line (from left to right), showing the interleaving of actions that would cause premature termination of the algorithm if the termination condition were to be reversed. Thread t_m is the main thread and thread t_w is a worker thread.

evaluating a cell and enqueue one or more cells such that queue would be non-empty. Thread t_w then decrements the counter such that its value becomes 0. Now t_m sees that no cells are currently being evaluated and therefore exits the main loop, even though there are still cells in the queue. This subtle race does not occur when we order the checks as in `RecalculatePartial()` in Fig. 5.5.

5.3.3 Cyclic Dependency Detection

To detect a cyclic dependency during sequential recalculation, it is sufficient to check whether a cell is `Computing` before evaluating it. Detecting cycles in parallel is less straightforward. If any thread sees a cell that is `Computing`, it has not necessarily found a cyclic dependency as *another* thread may currently be computing the cell. In this section, we discuss the challenges of parallel cycle detection and then outline our solution.

We could circumvent the problem by sequentially checking for cycles before initiating a parallel recalculation, but this would defeat the purpose of recalculating in parallel in the first place. Moreover, a sequential static cycle check would be too conservative and not allow for dynamic cyclic dependencies that never actually get evaluated. Alternatively, a thread that encounters a `Computing` cell could immediately report a cyclic dependency, but that would be overly pessimistic and therefore useless.

What we need is a tie-breaker that allows at least one thread to proceed even if the cell is `Computing`, such that it can discover any actually existing cyclic dependencies. During parallel recalculation, a cyclic dependency occurs only if a thread t_i encounters a cell that is `Computing` and whose *owner* is t_i itself. If a cell is `Computing` but owned by another thread, t_i waits until the cell becomes `Uptodate` and then reads the cell's cached value.

```

1  static class BCellState {
2      public const int Dirty = 0,
3                      Enqueued = 1,
4                      Computing = 2,
5                      Uptodate = 3;
6
7      const int bitMask = Enqueued | Computing | Uptodate;
8
9      public static int Computing() {
10         return (CurrentThread.Id << 2) | Computing;
11     }
12
13     public static int Owner(int state) {
14         return state >> 2;
15     }
16
17     public static int State(int state) {
18         return state & bitMask;
19     }

```

Figure 5.7: The BCellState (short for “bit cell state”) class and its auxiliary methods for encoding a thread ID in a state.

How do we decide which thread is allowed to proceed? We use thread IDs, that impose an arbitrary numerical order on threads to determine *thread precedence*. A thread t_i has precedence over t_j if its ID is lower than that of t_j . If t_i and t_j wait for a cell that the respective other thread owns due to a cyclic reference, then t_i may at some point proceed and discover the cycle.

5.3.4 Encoding Ownership in Cell State

We want to manipulate state and ownership of a cell using a single atomic operation to avoid adding logic for handling partial state, e.g. a cell that is Computing but that has no owner yet. Internally, cell state is represented by two bits of an integer, encoding four cell states, as shown in class BCellState in Fig. 5.7.

We can encode the ID of the current thread in the unused bits along with the Computing cell state to claim ownership of the cell. This allows us to manipulate both using a single CAS(). Calling State(s) returns

the state bits of `s` and `Owner(s)` returns the ownership bits. For all other cell states except `Computing`, the ownership bits are always zero. The following holds:

$$\text{State}(s) \equiv s \quad (5.1)$$

$$\text{Owner}(\text{Computing}()) \equiv \text{CurrentThread.Id} \quad (5.2)$$

5.3.5 Cycle Detection via Speculative Reevaluation

This section describes the implementation of a dynamic resolution of cyclic dependencies that uses speculative evaluation. We only want to report cycles that actually exist and occur *dynamically* during recalculation.

Figure 5.8 shows the `ThreadSafeFormula` class. It differs from the `MaybeThreadSafeFormula` class in some obvious and some more subtle ways. The `Enqueue()` method remains unchanged, while the logic in `Eval()` becomes more convoluted to enable the speculative evaluation scheme described in Sec. 5.3.3. Most importantly, it uses the methods from the `BCellState` class in Fig. 5.7 to encode thread IDs in state.

Detecting Cyclic Dependencies

In the case that state is `Computing`, `Eval()` checks whether the current thread is the owner of the cell (Line 29). If it is, a cyclic dependency has been detected and recalculation must abort. To allow any threads that are waiting for the current cell to finish up, `Eval()` sets the state to `Uptodate`. These threads may then continue their computation, possibly reading a stale value. Due to the exception, the main loop in `ParallelWorkbook` will terminate. While this approach may seem simplistic, it elegantly terminates the recalculation process in case of a cyclic dependency.

If the current thread does not own the cell, `Eval()` checks whether it has precedence over the current owner (Line 32) and, if so, jumps to the `Enqueued` case. If the current thread is neither the owner of the cell nor has precedence over the current owner, it spins until it can retrieve the cached value (Line 47). If the cell is either `Dirty` or `Enqueued`, the thread attempts to evaluate it directly (Line 38). If the cell is `Uptodate`, the function just returns the cached result.

Whenever a thread attempts to evaluate, it claims ownership of the cell (Line 14). This reduces the number of redundant speculative evaluations

```

1  class ThreadSafeFormula : Cell {
2      readonly Expre e; readonly Workbook wb;
3      volatile int state; Value cached;
4      public readonly List<Cell> supported;
5
6      public void MarkDirty() { ... }
7      public void Enqueue() { ... }
8
9      public bool TakeOwnership(int oldState) {
10         return CAS(ref state, BCellState.Computing(), oldState);
11     }
12
13     private bool TryEval(Value vo, int so) {
14         if (TakeOwnership(so)) {
15             Value v = e.Eval();
16             if (CAS(ref cached, v, vo)) {
17                 state = BCellState.Uptodate;
18                 return true;
19             }
20         }
21         return false;
22     }
23
24     public Value Eval() {
25         int s = state; // Store locally for consistency.
26         switch (s) {
27             case BCellState.Uptodate: break;
28             case BCellState.Computing:
29                 int id = CurrentThread.Id;
30                 if (id == BCellState.Owner(s)) { // Cycle case: release
31                     state = BCellState.Uptodate; // cell and notify
32                     throw new Exception("Cycle!"); // main loop.
33                 } else if (id < BCellState.Owner(s)) { // Precedence case:
34                     goto case BCellState.Enqueueed; // attempt evaluation.
35                 } else
36                     break;
37             case BCellState.Dirty: // Evaluation case:
38             case BCellState.Enqueueed: // attempt evaluation.
39                 while (BCellState.State(s) < BCellState.Computing
40                     || (id < BCellState.Owner(state)
41                         && BCellState.State(state) < BCellState.Uptodate)) {
42                     if (TryEval(state, cached)) { // Evaluate and enqueue.
43                         foreach (Cell c in supported) { c.Enqueue(); }
44                         break; // Leave while-loop.
45                     }
46                 }
47                 break;
48             while (BCellState.State(state) < BCellState.Uptodate)
49                 ; // Block until state is Uptodate.
50         }
51         return cached;
52     }
53 }

```

Figure 5.8: A thread safe implementation of the simplified Formula class with dynamic cycle detection. The Enqueue() method remains unchanged from MaybeThreadSafeFormula in Fig. 5.4.

and is important for cycle detection. Let us focus on the condition of the while-loop in the Enqueued case (same as in Fig. 5.8, Line 11 ff.):

```
while (CellState.State(s) < Cell.Computing
      || (id < CellState.Owner(state)
          && CellState.State(state) < Cell.Uptodate) {
  if (TryEval(state, cached)) { // ...
```

An evaluation should be attempted if

1. the local cell state is Dirty or Enqueued; or
2. the current thread has precedence over the owner of the cell and
3. the cell state is not yet Uptodate.

This captures all cases in which a thread should attempt to evaluate a cell, i.e. (1) the current thread is the first thread to arrive at the cell, or (2) the current thread has precedence and is therefore the only one that can detect a possible cyclic dependency while (3) the cell is not yet computed.

Why does a thread have to re-try evaluating a cell if it has precedence over the current owner? If a thread t_i has precedence over thread t_j and claims ownership of cell c , and another thread t_k has precedence over t_j but not t_i , such that $ID(t_i) < ID(t_k) < ID(t_j)$, then t_k can claim ownership of c before t_i .

Imagine now that cell c has a cyclic dependency on cell u owned by t_i and t_k successfully took ownership of c , while t_i failed to take ownership of c and waits for the ongoing evaluation to finish. As soon as t_k arrives at cell u , it would detect that it does not have precedence over t_i and recalculation would become stuck. This situation can only be resolved by t_i trying again to evaluate the cell c speculatively.

If thread t with $ID(t) = n$ does not return from evaluating a cell due to a cyclic reference, then in the worst case $n - 1$ threads with lower IDs can evaluate the same cell speculatively before one of them detects the cyclic dependency. Therefore, every cyclic dependency will eventually be discovered.

Ensuring Consistency

It is now possible that two or more threads attempt to evaluate the same cell, as illustrated in Fig. 5.9. In Sec. 5.3.1, we discussed that all threads

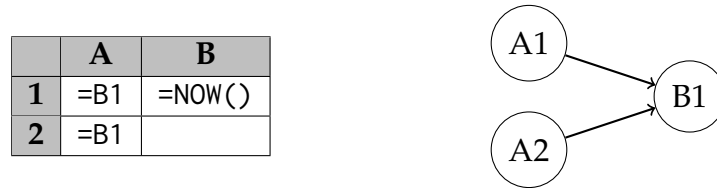


Figure 5.9: A spreadsheet (left) and its dependency graph (right). Two cells depend on the same cell containing a call to NOW(). If both cell A1 and A2 are evaluated in parallel and both recursively attempt to evaluate B1, their respective threads must agree upon which value B1 has evaluated to.

should agree on the cached value of each cell, so we must ensure that only one of the evaluating threads gets to set the cached value; the other threads must discard the result of their own evaluation and continue using the now updated cached value. Method TryEval() in Fig. 5.8 ensures that only one thread writes a new value to the cache via CAS().

In the absence of cyclic dependencies, our algorithm for parallel minimal recalculation retains the consistency requirements stated in Sec. 2.2.4. Calling CAS() in Line 16 makes sure that all threads will agree on the value of each cell in σ (Sec. 2.2.4).

The formal semantics described in Sec. 2.2.4 assigns a #CYCLE! error to cells that contain a cyclic dependency which is then propagated to other cells. This would not be useful when detecting cyclic dependencies via speculative reevaluation. For instance, a non-deterministic conditional, as illustrated in cell A4 in Fig. 5.10, could leave the spreadsheet in an inconsistent state. Imagine we were to assign #CYCLE! to cells that have a cyclic reference. Now assume that two threads that arrive at cell A4 evaluate both branches of the conditional in parallel, since calling RAND is non-deterministic; the thread that evaluates the then-branch would detect a cyclic reference from A1 to A4 and assign #CYCLE! to A1 and then, by propagation, to cell A2 as well. When it arrives back at cell A4, it could have happened that the thread that evaluated the else-branch was faster and already has written to the value cache, making the result of the speculative then-branch computation obsolete. However, there are now two cells in the spreadsheet whose value is #CYCLE!, even though the cyclic dependency cannot be observed by the user.

Our solution is to throw an exception as soon as *any* thread detects a cyclic dependency. This is slightly conservative and therefore sufficiently precise. Throwing an exception also leaves the spreadsheet in an

	A
1	=A3+1
2	=A1
3	=NOW()
4	=IF(RAND()<0.5, A2, A3)

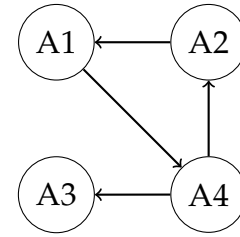


Figure 5.10: A spreadsheet (left) with a statically cyclic reference that is guarded by a non-deterministic conditional and its corresponding dependency graph (right). A cyclic error only occurs if the call to RAND in cell A4 evaluates to a value < 0.5 .

inconsistent state; threads that are still busy evaluating cells may read stale values. Therefore, Puncalc only retains the consistency requirement from Sec. 2.2.4 in the absence of cyclic dependencies. Sequential Funcalc behaves exactly equivalent.

Delayed Speculative Evaluation

In practice, we do not want to allow a thread with precedence to *immediately* evaluate a cell speculatively as in Line 33 in Fig. 5.8. Instead, it makes sense to briefly delay speculative evaluation while continuously checking the cell state. If the cell becomes Uptodate during this delay, the waiting thread does just reads the computed result; otherwise it proceeds by speculatively evaluating the cell. This heuristic makes sure that we do not needlessly start evaluating when the result is about to be available.

5.3.6 Thread-Local Evaluation

If a cell only has a single outgoing support edge, i.e. only a single cell in the spreadsheet refers to it, our algorithm will still spawn a new task for the single supported cell, even though there is no parallelism that we can exploit. Instead, the current thread could evaluate the cell locally, circumventing the global queue, thereby saving on communication overhead and avoiding spawning a new task.

We can implement an optimization for such sequential chains by detecting when a cell supports only a single cell. If so, we evaluate the supported cell locally on the current thread which continues to evaluate cells locally, until it reaches a cell that supports either zero or more than one cell, or is already Uptodate:


```
if (TryEval(state, cached)) {  
    if (supported.Count == 1) {  
        supported.Single().Eval();  
    } else {  
        foreach (Cell c in supported) { c.Enqueue(); }  
    }  
}
```

In the actual implementation, we use a while-loop to avoid overflowing the call stack. From a functional point of view, there is no difference.

5.4 Results and Validation

5.4.1 Benchmark Spreadsheets

We use the following spreadsheet suites to benchmark Puncalc:

Real-World Spreadsheets We use the LibreOffice Calc [48] benchmark sheets, as already describe in Sec. 4.5.1. We detect all formula cells that have no formula dependencies and use them as recalculation roots (column “Roots” in Table 5.1) to simulate minimal recalculation. As a result, they are initially enqueued in the global work queue, and the main thread can then dequeue cells from the queue without interference from enqueueing threads. This may have a positive effect on parallel performance and is unrealistic since users usually only edit one cell at a time.

Artificial Spreadsheets To explore Puncalc’s behavior in a controlled and systematic fashion, we use six programmatically generated spreadsheet topologies as shown in Fig. 5.11. Each cell calls a recursive SDF implementation of the Fibonacci function FIB (see Fig. 2.8 in Sec. 2.3.1) which allows us to control the amount of work per cell. We pass a parameter to FIB that corresponds to roughly 0.7ms evaluation time per call, which is the maximum average time per cell from all LibreOffice Calc spreadsheets.

5.4.2 Experimental Setup

Our test machine is the P3 server (see Sec. 1.3). We initially performed three warm-up runs and ran each benchmark for five iterations. The

Sheet	Cells	Roots	Support edges	Span	Seq. (s)
building-design	108 332	18 378	488 351 887	3	32.12
energy-markets	534 507	35 198	287 818 610	3	168.16
grossprofit	135 073	15 301	112 612 549	3	102.19
ground-water	126 404	31 601	1 099 366 302	1	81.26
stocks-history	226 503	23 402	317 049	3	64.9
stocks-price	812 693	10 876	233 376 389	3	102.74
binary-join	262 146	1	393 215	18	138.63
binary-tree	266 145	1	262 143	17	141.14
fork	300 001	1	300 301	1001	160.14
fork-join	300 002	1	300 600	1001	158.92
map	300 001	1	300 001	1	160.82
prefix	300 000	1	745 009	1100	161.32

Table 5.1: Spreadsheet statistics for the real-world spreadsheets (top) and the synthetic spreadsheets (bottom). The last column is the average time of 50 sequential minimal recalculations in seconds.

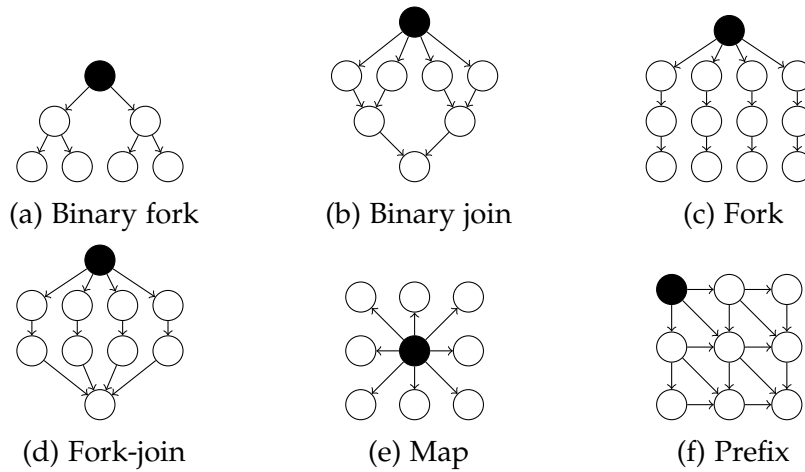


Figure 5.11: Illustrations of the underlying support graph structures of extreme, synthetic spreadsheets for benchmarking. Edges go from supporting cells to supported cells. Black nodes mark recalculation roots. We only use one recalculation root per sheet to simulate editing a single cell.

raw data are available online.² For each iteration, we ran the benchmark ten times and computed the average execution time. Sequential (1-core) running times are measured without volatile reads and writes, or any other thread safe primitives or data structures to ensure a fair comparison.

We limit the number of TPL threads to match the number of available, logical cores for each run. Additionally, we disable the TPL heuristics for thread creation and destruction so that all threads are created at start-up.

5.4.3 Validation

We have validated that our algorithm for parallel minimal recalculation in Puncalc produces the same result as sequential minimal recalculation in Funcalc for all sheets from the real-world benchmark suite by repeated testing. Hence, we believe that parallel recalculation respects the consistency requirements described in Sec. 2.2.4.

5.4.4 Performance Evaluation

There are three main observations to be made from the performance benchmarks:

Observation 5.1 *Figure 5.12 and 5.13 show that our approach scales for the majority of tested spreadsheets up to 16 cores, where we gain the largest speedup on average.*

The relative speedup decreases for all spreadsheets for more than 16 cores, except for building-design, ground-water and stocks-history from the LibreOffice benchmarks. The performance decline after 16 cores may be caused by increased contention and more speculative evaluations. Another explanation relates to the architecture of the Intel Xeon, which consists of two chips with twelve cores each. Up to 24 “logical” cores (i.e. including hyper-threading), communication does not happen across chips. Therefore, we do not have to pay an excessive synchronization cost when threads wait for Computing cells whose owners are scheduled off-chip. As already mentioned in Sec. 4.5.2, the structure of the three aforementioned sheets might correct for such expensive communication.

²<https://github.com/popular-parallel-programming/puncalc-benchmarks/tree/xeon>

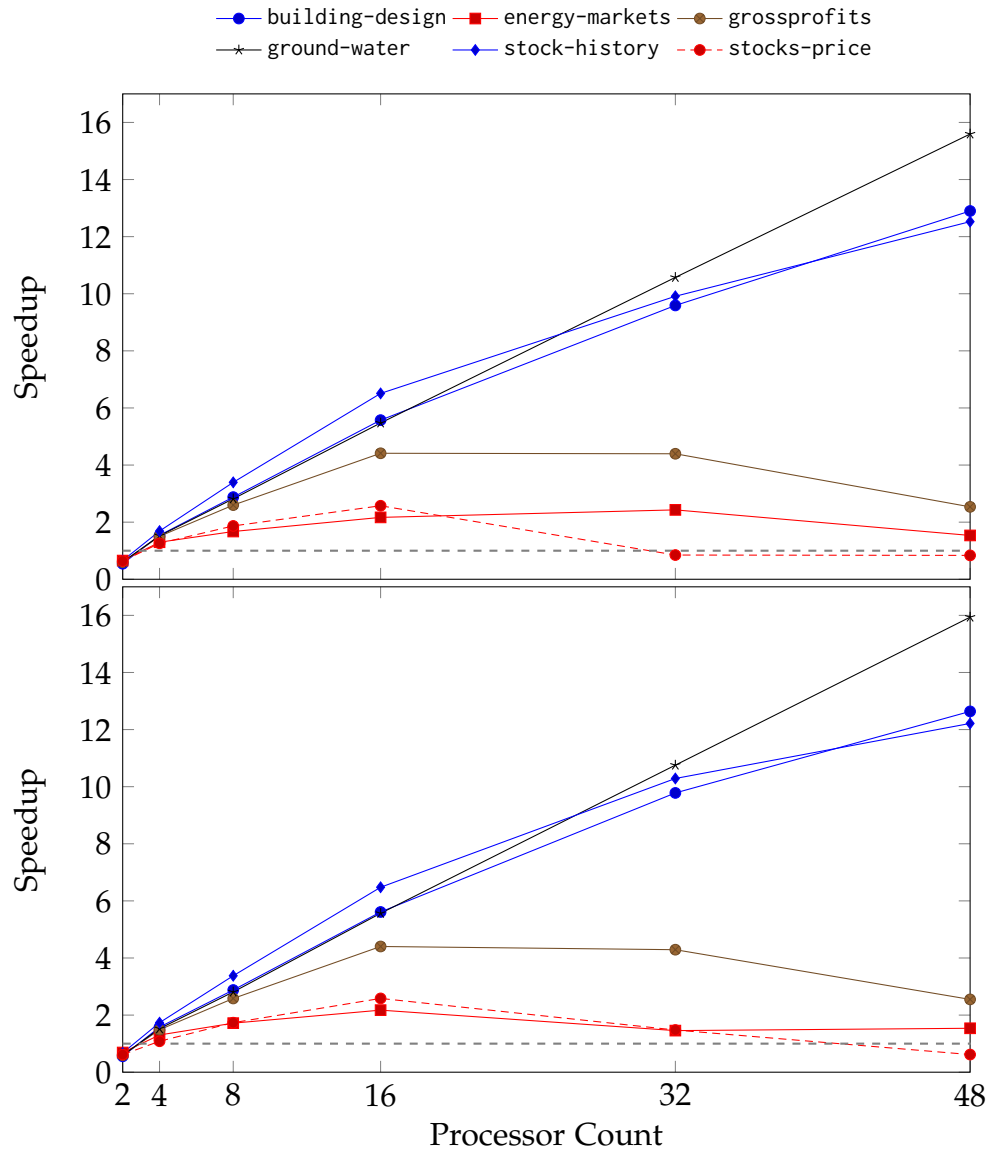


Figure 5.12: Average benchmark results over 50 runs per spreadsheet from the LibreOffice Calc spreadsheet suite. **Top:** *without* thread-local evaluation. **Bottom:** *with* thread-local evaluation. Values are speedup factors over sequential performance on the same machine; higher is better. The gray dashed line indicates 1-core performance.

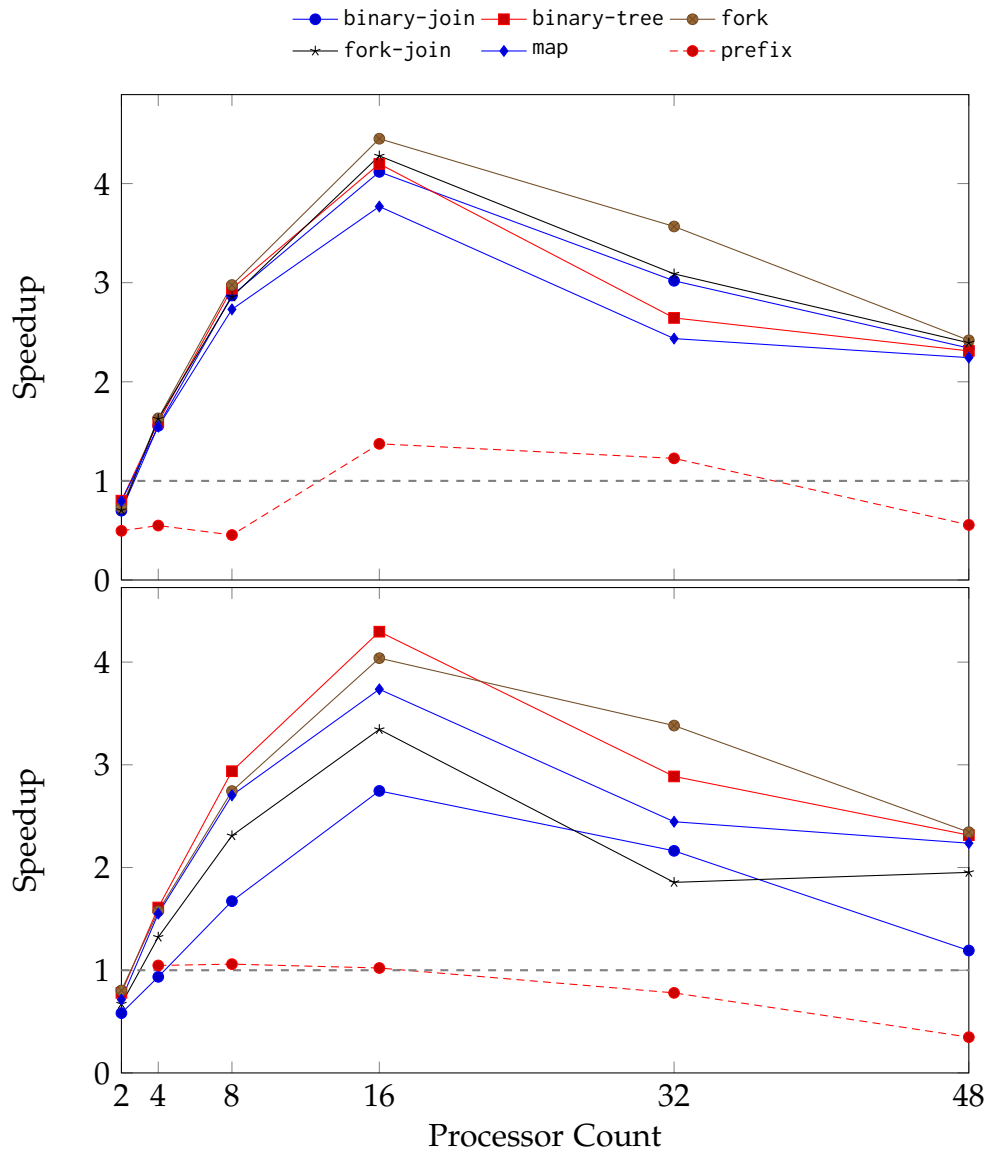


Figure 5.13: Average benchmark results over 50 runs per spreadsheet from the synthetic spreadsheet suite. **Top:** *without* thread-local evaluation. **Bottom:** *with* thread-local evaluation. Values are speedup factors over sequential performance on the same machine; higher is better. The gray dashed line indicates 1-core performance. The standard deviation is ≤ 0.1 for all benchmarks.

Observation 5.2 *Thread-local evaluation does not improve performance compared to eagerly spawning a task for each cell for real-world spreadsheets, as shown in Fig. 5.12, and often leads to worse performance than eagerly spawning tasks for synthetic spreadsheets, as shown in Fig. 5.13.*

This may be due to two factors. First, thread-local evaluation is a *depth-first* traversal, while eagerly spawning tasks is akin to *breadth-first* traversal. Therefore, thread-local evaluation makes recursive evaluation of dependencies more likely, which is slower than using the global work queue. For heavily sequential spreadsheets such as prefix (Fig. 5.11f), thread-local evaluation can alleviate the overhead of parallelization, which may be favorable for a robust implementation. However, recursive evaluation can lead to stack overflow errors. Second, the TPL uses an implementation of work-stealing [74]: idle threads steal work in the form of tasks from other threads. If we spawn fewer tasks and, hence, have more idle threads, they will attempt to steal work more often. Frequent work-stealing is more costly if it happens across chips.

Observation 5.3 *Neither the number of cells, roots, support edges or span (i.e. the longest sequential path) of a spreadsheet are good indicators for parallel performance.*

There is no apparent correlation between the performance results shown in Fig. 5.12 and 5.13 and the statistics in Table 5.1. This is much to our surprise. Unfortunately, these statistics do not inform us on the overall structure of the dataflow graph. Are there “bottleneck cells”, i.e. cells that refer to a large number of cells and support a large number of cells again? Such structural properties are not captured by the statistics in Table 5.1 and therefore, a manual analysis of the structure may be needed to discover the causes behind the observed results.

5.5 Related Work

Little research deals with parallel recalculation of spreadsheets. The general focus has instead been on detection and handling of errors [23].

5.5.1 Research Literature

There exist multiple distributed systems for spreadsheet computations, such as ActiveSheets³ [71] and Nimrod [2]. Both systems require re-engineering of the spreadsheet, which may take a substantial amount of time and require expert engineers [103]. In contrast, Puncalc runs on a shared-memory multiprocessor and automatically exploits the machine's available processors without the need to change the spreadsheet itself.

Wack [110] bridges the gap between distributed systems and automatic parallelization. His dissertation describes an improved spreadsheet model that statically partitions and schedules a set of predefined patterns and parallelizes them via message-passing in a network of work stations. Apart from using a different machine model, his work disallows cyclic dependencies [110, Sec. 2.8.3], which corresponds to static cycle detection.

5.5.2 Commercial and Open Source Applications

Microsoft Excel is probably the most well-known commercial spreadsheet application. Being closed-source, little information is available about its recalculation engine. Its GUI shows an option that allows users to enable multi-threaded recalculation, but its mechanics are unclear. Sestoft [93] gives some additional information on the internals on Excel.

AMD has, in collaboration with the LibreOffice open source project, implemented GPU parallelization for LibreOffice Calc by automatically compiling formulas involving cell ranges, such as `=SUM(A1:A100)`, into OpenCL kernels [105]. They report between 30-fold to 500-fold speedups [79], but do not take additional improvements to Libre Office Calc's internal data representation into account.

None of the applications above report results for systematic performance benchmarks or give a detailed description of the underlying algorithms.

5.6 Combining Dataflow with Cell Array Rewriting

It seems straightforward to combine Puncalc's parallel recalculation algorithm with rewriting of cell arrays, which we described in Chapter 4.

³Not to be confused with ActiveSheets by Vaziri et al. [109].

This combination essentially shrinks the dependency graph of the spreadsheet by merging cells arrays into array formulas and thereby introduces nested parallelism. Cells that are part of an array formula are not put on the global work queue; only the formula that represents the shared formula expression is enqueued.

5.6.1 Enabling Nested Parallelism

Our design of the Puncalc algorithm is not geared towards nested parallel operations. Threads wait for not-yet computed cells in a busy-wait loop. When a large array formula is computed by one thread, it can happen that many threads wait for values of this array formula and just waste their time spinning. That means that there are no threads left that can evaluate the array formula in parallel and ultimately, the algorithm only pays for the overhead of distributing parallel work without gaining the benefits from parallelization.

There are two changes that we have to make to Puncalc to enable nested parallelism. First, we must allow for more software threads than hardware threads. This is a trivial change to the program configuration and we now run twice as many threads as there are processors.

The second change allows threads to yield their time slice every once in a while when waiting to let the processor they run on perform some useful work instead. We achieve this by using a heuristic that puts a thread to sleep every n iterations that it has spun. The .NET framework implements this heuristic in the `SpinWait` class in the `System.Threading` namespace. We change the code in Line 47, Fig. 5.8 to:

```
public Value Eval() {
    int s = state;
    switch (s) {
        // ... as before.
    }
    SpinWait.SpinUntil(
        () => BCellState.State(state) == BCellState.
            Computing);
    return cached;
}
```

Method `SpinWait.SpinUntil()` returns when the anonymous function that is has been called with returns true.

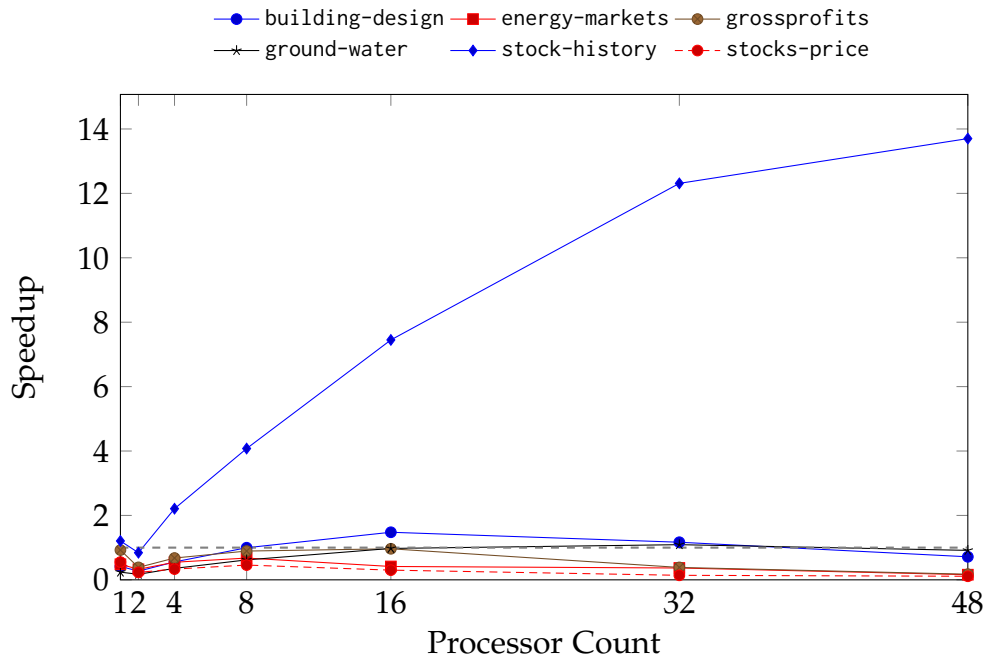


Figure 5.14: Average benchmark results over 30 runs per spreadsheet from the LibreOffice Calc spreadsheet suite with cell array rewriting and without thread-local evaluation. Values are speedup factors over sequential minimal recalculation on the same machine; higher is better. The gray dashed line indicates 1-core performance without cell array rewriting. The standard deviation is ≤ 0.19 for all benchmarks.

5.6.2 Performance Results

We use the same experimental setup as described in Sec. 5.4.2. We use the LibreOffice Calc benchmarking suite and report the average speedup of running 30 iterations for each configuration.

The graph in Fig. 5.14 shows that the performance of rewritten spreadsheets is worse already for single core minimal recalculation (Processor Count = 1). Except for grossprofits and stock-history, all spreadsheets take twice as much time for a minimal recalculation. Moreover, none of the spreadsheets except for stock-history, which already performed best for cell array rewriting with full recalculation (see Sec. 4.5.2), gain any speedup from combining Puncalc with cell array rewriting.

We already noted in Sec. 4.5 that measuring the performance of full recalculation does not reflect how cell array rewriting affects the support graph. The observation that we can make from the results in Fig. 5.14

is that cell array rewriting indeed affects the performance of minimal recalculation negatively.

However, it is surprising that Puncalc is unable to harness any nested parallelism. We believe that this is due to speculative evaluation: Puncalc attempts to parallelize sequential chains of parallel array formulas and threads spend most of their time waiting for the result of the array formulas to become available. If it takes sufficiently long to compute an array formula, speculative evaluation will trigger and threads begin to compute large chunks of the spreadsheet twice, even though there is no cyclic dependency that can be uncovered and effectively waste their time slice on performing redundant work. It would be useful to investigate more dynamic heuristics for delaying speculative evaluation, such as estimating the time it will take to evaluate a particular cell ahead of time.

5.7 Conclusion

In this chapter, we presented Puncalc, a spreadsheet engine that targets shared-memory multiprocessors and automatically extracts parallelism from spreadsheets via their underlying support graph. Puncalc obtains overall satisfactory speedups of up to nearly 16 times on 48 processors without adding any engineering overhead on the end-user's side. To our knowledge, this is the first algorithm for parallel spreadsheet recalculation with dynamic cycle detection that has been described in literature.

We have given a number of possible explanations for the performance results in Sec. 5.4.4. Furthermore, we are lacking a direct comparison of the performance of Puncalc to that of other frameworks for spreadsheet parallelization, such as those mentioned in Sec. 5.5.

Combining Puncalc with rewriting cell arrays does not allow us to harness nested parallelism from spreadsheets. We suspect that the static heuristic that we use for delayed speculative evaluation allows too many threads to speculatively evaluate large cell arrays, which often is redundant work. However, we need a more sophisticated approach to estimate how long the evaluation of a particular cell will take to improve our heuristic, which we regard as future work.

Chapter 6

End-User Array Programming

This chapter is based on the article **“Quad Ropes: Immutable, Declarative Arrays with Parallelizable Operations”** [13] which is joint work with Peter Sestoft.

6.1 Introduction

Programmers choose data structures for the programs they write based on their performance properties. For instance, arrays allow random access and update in constant time and higher-order combinators are easily parallelizable, whereas linked lists are inherently sequential and random access takes linear time, but adding a value to the beginning of the list takes constant time. The choice of data structure often has a crucial impact on the performance of the entire program.

For end-user development in high-level declarative programming languages it is impractical to let users choose between different data structures, because end-users are often not primarily educated as programmers. Instead, they should be able to use the same, somewhat performance-robust, representation for every programming task.

In this chapter, we describe the design of the *quad rope* data structure, a representation of immutable two-dimensional arrays. It avoids many of the performance pitfalls of naively using C-style two-dimensional arrays to represent immutable data collections in high-level declarative programming languages, such as repeated concatenation and update. Quad ropes roughly retain array efficiency, as long as programmers express their programs using high-level constructs.

Quad ropes are a combination of ropes [24] and quad trees [45]: a quad rope is a binary tree with one concatenation constructor for each dimension and with small, contiguous, two-dimensional arrays at its leafs.

6.1.1 Choosing a Declarative Array Representation

To find a representation of immutable, two-dimensional arrays that efficiently and pragmatically caters to the needs of high-level declarative array programming, let us begin by considering some requirements for such a representation:

1. The data structure must be immutable; immutability gives us considerable freedom for the implementation and allows for implicit parallelization, constant-time slicing, caching, speculative evaluation and evaluation order independence. It is also straightforward for end-user developers to reason about.
2. It should gracefully handle array programming anti-patterns, such as gradual array construction by repeated concatenation.
3. Higher-order combinators should be efficient and able to exploit data-parallelism.
4. Users should not experience seemingly arbitrary performance differences between operations in different dimensions, e.g. horizontal and vertical. We call this *performance symmetry*.

Random-access arrays are highly efficient for the majority of use cases, except for repeated concatenation, so it is difficult to design a data structure that behaves like immutable arrays and is equally fast. Most prior research focused on efficient immutable arrays using versioning approaches [38, 72] without efficient concatenation. Kaplan and Tarjan [67] showed how to implement fast concatenation of persistent deques, which, however, do not grant random access.

Stucki et al. [102] designed the one-dimensional relaxed radix-bound (RRB) tree with constant-time indexing and concatenation, fulfilling requirements 1–3. Extending RRB trees to two dimensions is not feasible: performance symmetric two-dimensional concatenation requires managing many corner cases and often leaves us in situations where we cannot avoid excessive re-allocations.

Finkel and Bentley [45] designed quad trees to allow for multi-dimensional key retrieval. The main idea is to recursively subdivide the rectangle that contains values into further rectangles, where empty rectangles are simply not represented. They fulfill all requirements, but may exhibit excess parallelism.

The discontinued Fortress language used ropes to implement parallel collections [100, 101]. A rope is an immutable binary tree with strings or arrays at its leafs [24]. The idea is to group scalar values at the leafs into small contiguous arrays and to extract parallelism by forking threads at each tree branch, where a well chosen minimum leaf size avoids excess parallelism. The binary tree structure also allows for constant-time concatenation.

We can generalize ropes to two dimensions to fulfill all four requirements, in the same way in which quad trees generalize binary trees. We call the resulting data structure a quad rope. Quad ropes have only a modest performance overhead compared with immutable two-dimensional arrays, except for indexing, which runs in logarithmic instead of constant time. Since we want to encourage a high-level style of array programming using higher-order combinators, we deem this acceptable. For small array sizes, quad ropes fall back to the default array implementation, eliminating any overhead whatsoever.

Quad ropes are conceptually similar to hierarchically tiled arrays (HTA) [15]. HTAs, however, have only been researched for largely imperative languages. Our approach is to define higher-order combinators on quad ropes that match the signatures of typical array combinators and use quad ropes as a “drop-in” replacement for other array representations.

6.1.2 Contributions

In the remainder of this chapter, we give an operational semantics for the quad rope data structure and discuss the implications for performance; we show that it is straightforward to use quad ropes to represent sparse matrices; we discuss balancing and parallelization of operations and show that it is not possible to use a lazy tree splitting scheduler [8] on quad ropes; and we discuss implementation and performance benchmarks of quad ropes in F# and in Funcalc.

6.2 Quad Rope Semantics

We use a straightforward operational semantics to describe quad ropes, with the usual semantics for arithmetic operations, lambda abstraction and function application. Abstractions accept an arbitrary number of arguments. We illustrate most rules using only **hcat**. The rules for **vcat** are analogous and operate on the values that describe columns.

We use the judgment $e \text{ qr}$ to say that an expression e is a quad rope value and $e \text{ arr}$ to say that an expression e is an array. The judgment $e \downarrow e'$ says that there is a complete transition sequence from e to e' . Combinators on arrays are written in gray; e.g. **map** denotes the map combinator on two-dimensional arrays. We do not give the semantics for combinators on arrays and instead focus on the semantics of quad ropes. However, we use array combinators to signify that we “escape” to array semantics and to avoid modeling any implementation details that are irrelevant for our overall idea.

Figure 6.1 shows the language used to describe quad ropes and Fig. 6.2 shows evaluation judgments for basic functions on them. A quad rope is either a **leaf**, represented by a two-dimensional array xs (rule **LEAF**); or one of **hcat** or **vcat**, which model horizontal and vertical concatenation, respectively (rules **HCAT** and **VCAT**). We could generalize quad ropes to more than two dimensions, either by statically adding further concatenation constructors, or by using a single concatenation form with an additional “dimension” parameter. However, we focus on the two-dimensional case in the remainder of this chapter.

We have two basic forms for constructing a new quad rope from the ground up: the form **init**(r, c, f) generates a new **leaf** of shape $r \times c$ where the value at index i, j is generated by calling $f(i, j)$ via rule **INIT**; the form **rep**(r, c, v) creates a quad rope that contains the same value v at all indices via rule **REP**.

The **rows** construct allows to query a quad rope for its number of rows. Both branches of a **hcat** node have the same number of rows, so it does not matter which branch we recurse on. Rules for computing the number of columns with **cols** are analogous, and both branches of a **vcat** must have the same number of columns. We define a *shape* operator $\boxed{\cdot}$ on quad ropes for that we use throughout this chapter:

$$\boxed{q} = (\text{rows}(q), \text{cols}(q))$$

$a ::= xs$	A two-dimensional array.
$\mathbf{get}(xs, i, j)$	Random access at index i, j .
$\mathbf{rows}(xs)$	Number of rows in xs .
$\mathbf{cols}(xs)$	Number of columns in xs .
$\mathbf{map}(f, xs)$	Apply function f to all elements in xs .
$\mathbf{slice}(i, j, r, c, xs)$	Build a rectangular sub-array.
$\mathbf{reduce}(\oplus, \varepsilon, xs)$	Reduction for binary operator \oplus .
$\mathbf{scan}(f, f, v, f, xs)$	Generalized prefix-sum.
$e ::= a$	Expressions on arrays.
v	Scalar value.
x	Variable name.
$\lambda(x_1, x_2, \dots).e$	Function abstraction.
$f(e_1, e_2, \dots)$	Function application.
$e \oplus e$	Binary application, short for $\oplus(e, e)$.
$\mathbf{rows}(q)$	Number of rows in q .
$\mathbf{cols}(q)$	Number of columns in q .
$\mathbf{get}(q, i, j)$	Random access at index i, j .
$\mathbf{set}(q, i, j, v)$	Update the value at index i, j to v .
$\mathbf{reduce}(\oplus, v, q)$	Reduction for binary operator \oplus .
$\mathbf{leaf}(i, j, r, c, xs)$	Quad rope leaf over xs .
$\mathbf{rep}(r, c, v)$	Replicated quad rope of size $r \times c$.
$\mathbf{init}(r, c, f)$	Build quad rope leaf of size $r \times c$.
$\mathbf{hcat}(q, q)$	Concatenate horizontally.
$\mathbf{vcat}(q, q)$	Concatenate vertically.
$\mathbf{slice}(i, j, r, c, q)$	Build a rectangular subset of values.
$\mathbf{map}(f, q)$	Apply function f to all elements in q .
$\mathbf{zipWith}(f, q, q)$	Combine two quad ropes pointwise using \oplus .
$\mathbf{scan}(f, f, v, f, q)$	Generalized prefix-sum.

Figure 6.1: The language which we use throughout this chapter to describe the quad rope semantics: xs and ys range over two-dimensional arrays; q ranges over quad ropes; f, g and k range over functions; and i, j, r and c range over integers.

$$\begin{array}{c}
\text{LEAF} \frac{xs \text{ arr}}{\mathbf{leaf}(xs) \text{ qr}} \\
\\
\text{HCAT} \frac{q_1 \text{ qr} \quad q_2 \text{ qr}}{\mathbf{hcat}(q_1, q_2) \text{ qr}} \quad \mathbf{rows}(q_1) = \mathbf{rows}(q_2) \\
\\
\text{VCAT} \frac{q_1 \text{ qr} \quad q_2 \text{ qr}}{\mathbf{vcat}(q_1, q_2) \text{ qr}} \quad \mathbf{cols}(q_1) = \mathbf{cols}(q_2) \\
\\
\text{INIT} \frac{\mathbf{init}(r, c, f) \downarrow xs}{\mathbf{init}(r, c, f) \downarrow \mathbf{leaf}(xs)} \quad 0 \leq i \wedge 0 \leq j \\
\\
\text{REP} \frac{}{\mathbf{rep}(r, c, v) \downarrow \mathbf{init}(r, c, \lambda(x, y).v)} \\
\\
\text{ROWS-L} \frac{q \downarrow \mathbf{leaf}(xs)}{\mathbf{rows}(q) \downarrow \mathbf{rows}(xs)} \quad \text{ROWS-H} \frac{q \downarrow \mathbf{hcat}(q_1, q_2)}{\mathbf{rows}(q) \downarrow \mathbf{rows}(q_1)} \\
\\
\text{ROWS-V} \frac{q \downarrow \mathbf{vcat}(q_1, q_2)}{\mathbf{rows}(q) \downarrow \mathbf{rows}(q_1) + \mathbf{rows}(q_2)} \\
\\
\text{SLICE-L} \frac{q \downarrow \mathbf{leaf}(xs) \quad \mathbf{slice}(i, j, r, c, xs) \downarrow ys}{\mathbf{slice}(i, j, r, c, q) \downarrow \mathbf{leaf}(ys)} \\
\\
\text{SLICE-H} \frac{q \downarrow \mathbf{hcat}(q_1, q_2) \quad \mathbf{slice}(i, j, r, c, q_1) \downarrow q'_1 \quad \mathbf{slice}(i, j - \mathbf{cols}(q_1), r, c - \mathbf{cols}(q'_1), q_2) \downarrow q'_2}{\mathbf{slice}(i, j, r, c, q) \downarrow \mathbf{hcat}(q'_1, q'_2)} \\
\\
\text{GET-L} \frac{q \downarrow \mathbf{leaf}(xs)}{\mathbf{get}(q, i, j) \downarrow \mathbf{get}(xs, i, j)} \quad 0 \leq i < \mathbf{rows}(q) \wedge 0 \leq j < \mathbf{cols}(q) \\
\\
\text{GET-H1} \frac{q \downarrow \mathbf{hcat}(q_1, q_2)}{\mathbf{get}(q, i, j) \downarrow \mathbf{get}(q_1, i, j)} \quad j < \mathbf{cols}(q_1) \\
\\
\text{GET-H2} \frac{q \downarrow \mathbf{hcat}(q_1, q_2) \quad j - \mathbf{cols}(q_1) \downarrow j'}{\mathbf{get}(q, i, j) \downarrow \mathbf{get}(q_2, i, j')} \quad j \geq \mathbf{cols}(q_1)
\end{array}$$

Figure 6.2: Operational semantics for quad ropes. The rules for **cols** are analogous to those of **rows** with **hcat** and **vcat** swapped. For every rule on **hcat** nodes with an “H”-suffix, there exists an analogous rule on **vcat** nodes, suffixed with “v”. The only case where we show this explicitly are the rows rules.

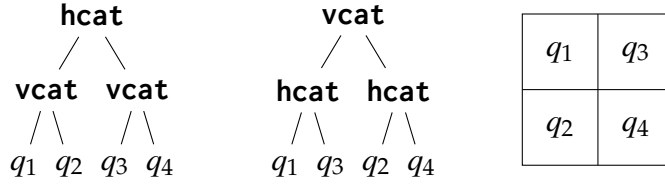


Figure 6.3: A quad rope illustrated twice as binary tree and once as box diagram. To simplify the example, we assume that $\forall q_i, q_j \in q_{[1,4]}. \boxed{q_i} = \boxed{q_j}$.

The form **slice**(i, j, r, c, q) describes a rectangular sub-set of a quad rope q , starting at index (i, j) and taking r rows and c columns. It adheres to the slicing semantics of one-dimensional ropes [24]. Finally, indexing via **get** takes logarithmic time in the **hcat** and **vcap** case on balanced quad ropes, which we will take a closer look at in Sec. 6.5.2. We omit rules for persistent update via **set**; they follow the GET rules closely, but update the leaf node at the given index with the new value and reconstruct all nodes on the path from the leaf to the root node, re-using the unchanged sibling nodes.

It is useful to think of quad ropes not only as binary trees, but also as rectangles that are composed of smaller rectangles. Figure 6.3 illustrates the relationship between quad ropes as binary trees and quad ropes as two-dimensional arrays. There does not exist a single canonical term to describe a particular quad rope. This is a consequence of using one concatenation form for each dimension, i.e. **hcat** and **vcap**. Two quad ropes that are element-wise equal and that have the same external shape do not necessarily have the same internal structure.

It is straightforward to support additional operations on quad ropes, such as transpose and row- and column-wise reversal.

6.2.1 Projection

The **map**(f, q) combinator lifts a scalar unary function f to operate on a quad rope q and applies it recursively to all branches and each of their elements, by rules MAP-L and MAP-H in Fig. 6.4. The **zipWith** combinator is the usual binary variant of **map**; however, we need to distinguish between the external shape and the internal structure of two quad ropes, since there is no canonical term for a quad rope of a given shape. If the structure of two quad ropes matches, we can use rules ZIP-L to directly

$$\begin{array}{c}
\frac{q \downarrow \mathbf{leaf}(xs) \quad \mathbf{map}(f, xs) \downarrow ys}{\mathbf{map}(f, q) \downarrow \mathbf{leaf}(ys)} \text{MAP-L} \\
\\
\frac{q \downarrow \mathbf{hcat}(q_1, q_2)}{\mathbf{map}(f, q) \downarrow \mathbf{hcat}(\mathbf{map}(f, q'_1), \mathbf{map}(f, q'_2))} \text{MAP-H} \\
\\
\frac{\mathbf{init}(\lambda(x, y).(\mathbf{get}(xs, x, y) \oplus \mathbf{get}(q_2, x, y)), \mathbf{rows}(q_1), \mathbf{cols}(q_2)) \downarrow q \quad q_1 \downarrow \mathbf{leaf}(xs)}{\mathbf{zipWith}(\oplus, q_1, q_2) \downarrow q} \text{ZIP-L} \\
\\
\frac{\begin{array}{c} q_1 \downarrow \mathbf{hcat}(q_{11}, q_{12}) \\ q_2 \downarrow \mathbf{hcat}(q_{21}, q_{22}) \\ \mathbf{zipWith}(\oplus, q_{1i}, q_{2i}) \downarrow q'_i \end{array}}{\mathbf{zipWith}(\oplus, q_1, q_2) \downarrow \mathbf{hcat}(q'_1, q'_2)} \text{ZIP-H} \quad \mathbf{cols}(q_{1i}) = \mathbf{cols}(q_{2i}) \\
\\
\frac{\begin{array}{c} q_1 \downarrow \mathbf{hcat}(q_{11}, q_{12}) \\ \mathbf{slice}(0, 0, \mathbf{rows}(q_2), \mathbf{cols}(q_{11}), q_2) \downarrow q_{21} \\ \mathbf{slice}(0, \mathbf{cols}(q_{11}), \mathbf{rows}(q_2), \mathbf{cols}(q_{12}), q_2) \downarrow q_{22} \\ \mathbf{zipWith}(\oplus, q_{1i}, q_{2i}) \downarrow q'_i \end{array}}{\mathbf{zipWith}(\oplus, q_1, q_2) \downarrow \mathbf{hcat}(q'_1, q'_2)} \text{ZIP-GEN-H}
\end{array}$$

Figure 6.4: Operational semantics for quad rope combinators. The ZIP rules have side condition $\boxed{q_1} = \boxed{q_2}$.

combine **leaf** forms and ZIP-H for recursing on the branches of **hcat** forms.

When this is not the case, we must rely on slicing. It is then sufficient to recurse on the structure of the left hand side argument and to slice the right hand side to match the shape of the left hand side, as in ZIP-GEN-H. Figure 6.5 illustrates this case for a **hcat** node and another quad rope of different structure.

$$\begin{array}{c}
\text{zipWith}(\oplus, \text{hcat}(\boxed{q_1}, \boxed{q_2}), \boxed{q_3}) \\
\downarrow \\
\text{hcat}(\text{zipWith}(\oplus, \boxed{q_1}, \boxed{q_{3a}}), \text{zipWith}(\oplus, \boxed{q_2}, \boxed{q_{3b}}))
\end{array}$$

Figure 6.5: Zipping two quad ropes of equal external shape but different internal structure. The boxes illustrate the shape of the respective branches. Branch q_{3a} is the left part of q_3 , sliced to match the width of q_1 ; branch q_{3b} is the respective right part and sliced to match the width of q_2 .

$$\begin{array}{c}
\text{RED-L} \frac{q \downarrow \text{leaf}(xs)}{\text{reduce}(\oplus, \varepsilon, q) \downarrow \text{reduce}(\oplus, \varepsilon, xs)} \\
\\
\text{RED-H} \frac{q \downarrow \text{hcat}(q_1, q_2)}{\text{reduce}(\oplus, \varepsilon, q) \downarrow \text{reduce}(\oplus, \varepsilon, q_1) \oplus \text{reduce}(\oplus, \varepsilon, q_2)}
\end{array}$$

Figure 6.6: Operational semantics for reduction of a quad rope to a scalar value via the **reduce** combinator.

6.2.2 Reduction and Scan

We focus on parallelizable reduction to a single scalar value and therefore, we require the operator \oplus that we reduce with to be associative and have an identity element. Thus all reduction rules (RED) in Fig. 6.6 have these side conditions:

- \oplus is associative; and
- $\varepsilon \oplus e = e \oplus \varepsilon = e$, so ε is the identity element for \oplus .

Reduction of an empty quad rope gives ε . We can use generalized two-dimensional reduction and slicing to implement row- and column-wise reduction.

One-dimensional scan is usually defined for a binary operator. To make **scan** on quad ropes as general as possible, our semantics uses a function that accepts four arguments instead of only two. Let us recall the definition of the general scan2d combinator from Sec. 3.1.4: it takes a

$$\begin{array}{c}
q \downarrow \mathbf{leaf}(xs) \\
\mathbf{scan}(f, \gamma, \delta, \rho, xs) \downarrow ys \\
\hline
\text{SCAN-L} \quad \mathbf{scan}(f, \gamma, \delta, \rho, q) \downarrow \mathbf{leaf}(ys) \\
\\
q \downarrow \mathbf{hcat}(q_1, q_2) \\
\mathbf{scan}(f, \gamma, \delta, \rho, q_1) \downarrow q'_1 \\
\lambda(x).(\mathbf{get}(q'_1, x, \mathbf{cols}(q'_1) - 1)) \downarrow \gamma' \\
\lambda(y).(\rho(y + \mathbf{cols}(q'_1))) \downarrow \rho' \\
\rho(\mathbf{cols}(q_1) - 1) \downarrow \delta' \\
\mathbf{scan}(f, \gamma', \delta', \rho', q_2) \downarrow q'_2 \\
\hline
\text{SCAN-H} \quad \mathbf{scan}(f, \gamma, \delta, \rho, q) \downarrow \mathbf{hcat}(q'_1, q'_2)
\end{array}$$

Figure 6.7: Operational semantics for computing two-dimensional prefix-sums via `scan`.

4-ary function f , a $m \times 1$ single-column array γ , a scalar value δ and a $1 \times n$ single-row array ρ as well as an input array xs of shape $m \times n$. Its result is a new $m \times n$ array

$$\text{scan2d } f \ \gamma \ \delta \ \rho \ xs \equiv ys$$

where:

$$ys[0,0] = f \ \gamma[0] \quad \delta \quad \rho[0] \quad xs[0,0] \quad (6.1)$$

$$ys[0,j] = f \ ys[0,j-1] \ \rho[j-1] \quad \rho[j] \quad xs[0,j] \quad (6.2)$$

$$ys[i,0] = f \ \gamma[i] \quad \gamma[i-1] \quad ys[i-1,0] \quad xs[i,0] \quad (6.3)$$

$$ys[i,j] = f \ ys[i,j-1] \ ys[i-1,j-1] \ ys[i-1,] \ xs[i,j] \quad (6.4)$$

We describe `scan` semantics for quad ropes in Fig. 6.7. We model γ and ρ using unary functions to translate prefix values from the upper-left branches to the lower-right branches of `hcat` and `vcap` nodes in rule `SCAN-H`. For each right-hand side branch, we construct two new functions γ' and ρ' that perform a lookup on the already scanned left-hand side branch.

The `scan` semantics does not exhibit any obvious possibilities for parallel execution. We will however see in Sec. 6.5 that there are configurations where it is possible to recursively evaluate some branches in parallel.

$$\begin{array}{c}
\text{REP}' \frac{}{\mathbf{rep}(r, c, v) \text{ qr}} \\
\text{ROWS-R} \frac{q \downarrow \mathbf{rep}(r, c, v)}{\mathbf{rows}(q) \downarrow r} \\
\text{GET-R} \frac{q \downarrow \mathbf{rep}(r, c, v)}{\mathbf{get}(q, i, j) \downarrow v} \quad 0 \leq i < r \wedge 0 \leq j < c \\
\text{SLICE-R} \frac{\begin{array}{l} q \downarrow \mathbf{rep}(r', c', v) \\ r'' = \min(r, r' - i) \\ c'' = \min(c, c' - j) \end{array}}{\mathbf{slice}(i, j, r, c, q) \downarrow \mathbf{rep}(r'', c'', v)} \quad 0 \leq i \wedge 0 \leq j \\
\text{MAP-R} \frac{q \downarrow \mathbf{rep}(r, c, v)}{\mathbf{map}(f, q) \downarrow \mathbf{rep}(r, c, f(v))} \\
\text{RED-R} \frac{q \downarrow \mathbf{rep}(r, c, v)}{\mathbf{red}(\oplus, \varepsilon, q) \downarrow \varepsilon \oplus \underbrace{v \oplus \dots \oplus v}_{r \cdot c}}
\end{array}$$

Figure 6.8: Additional operational semantics for a canonical **rep** form.

6.3 Block-Sparseness

6.3.1 Making Replication Canonical

In Sec. 6.2 we have defined the rule **REP**, saying that $\mathbf{rep}(r, c, v) \downarrow \mathbf{init}(r, c, \lambda(x, y).v)$. We can optimize the representation of such constant or replicated quad ropes by making the **rep** form canonical and thereby introducing *sparse blocks* to our quad rope data structure. That is, we replace rule **REP** by **REP'** in Fig. 6.8, such that the replication form is a proper quad rope itself. This retains the information that all elements in the resulting quad rope are equal. It is straightforward to show that $\forall i \in [0, r), j \in [0, c)$:

$$\mathbf{get}(\mathbf{rep}(r, c, v), i, j) \equiv \mathbf{get}(\mathbf{init}(r, c, \lambda(x, y).v), i, j),$$

since the only value that we can ever access via **get** is v .

The rule **MAP-R** shows the benefit of making replication canonical: we only need to apply the lifted function once, instead of to each individual value in the quad rope via rules **MAP-H** and **MAP-L**. Similarly, when we want to reduce a replicated quad rope with some binary associative operator \oplus and an identity element ε , we can avoid all computation if the replicated quad rope only contains ε . Since we assume $\varepsilon \oplus e = e \oplus \varepsilon = e$ it holds that $\varepsilon \oplus \varepsilon = \varepsilon$.

6.3.2 Block-Sparse Numerical Operations

More well known optimizations for numeric operators apply to replicated quad ropes. For instance, element-wise combination using the **zipWith** combinator can make use of sparseness. If we combine two quad ropes element-wise using the addition operator $+$ and one of them is of the form **rep**($r, c, 0.0$), it is sufficient return the other quad rope as a result; this is correct since zero is the identity element for addition.

$$\frac{q_1 \downarrow \mathbf{rep}(r, c, 0.0)}{\mathbf{zipWith}(+, q_1, q_2) \downarrow q_2} \quad (6.5)$$

We can proceed similarly for element-wise multiplication using the \cdot operator. This is useful when implementing functional matrix multiplication on quad ropes.

$$\frac{q_1 \downarrow \mathbf{rep}(r, c, 1.0)}{\mathbf{zipWith}(\cdot, q_1, q_2) \downarrow q_2} \quad (6.6)$$

These optimizations can be generalized to any ring. Furthermore, the **zipWith** combinator reduces to **map** when one of the argument quad ropes is sparse:

$$\frac{q_1 \downarrow \mathbf{rep}(r, c, v)}{\mathbf{zipWith}(\oplus, q_1, q_2) \downarrow \mathbf{map}(\lambda(x). (v \oplus x), q_2)} \quad (6.7)$$

All sparseness optimizations can also be applied if the right-hand side q_2 is sparse. For the optimization in (6.7), if q_2 is of form **rep**, the right-hand side argument to \oplus is fixed to the v from q_2 .

6.4 Two-Way Nodes vs. Four-Way Nodes

As stated in Sec. 6.2, a particular quad rope does not necessarily have a canonical form. In particular, due to having both horizontal and vertical concatenation, **hcat**(**vcat**(q_1, q_2), **vcat**(q_3, q_4)) and

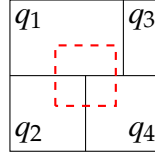


Figure 6.9: A four-way node configuration $\mathbf{node}(q_1, q_2, q_3, q_4)$. The red rectangle illustrates the size of the slice we want to compute. The number in columns in q'_4 depend on q_2 . Furthermore, the resulting slice consists of only three branches, $\mathbf{node}(q'_1, q'_2, \epsilon, q'_4)$.

$\mathbf{vcat}(\mathbf{hcat}(q_1, q_3), \mathbf{hcat}(q_2, q_4))$ are two representations of the same quad rope. This section explains why we do not use the “obvious” four-way construction operator to avoid this ambiguity.

Suppose we have a four-way node construct $\mathbf{node}(q_1, q_2, q_3, q_4)$ that is a proper quad rope and let ϵ be the canonical empty quad rope. We define $\mathbf{hcat}(q_1, q_2) \downarrow \mathbf{node}(q_1, \epsilon, q_2, \epsilon)$ and $\mathbf{vcat}(q_1, q_2) \downarrow \mathbf{node}(q_1, q_2, \epsilon, \epsilon)$.

All expressions on quad ropes can now neglect special rules for each dimension, but must take ϵ into account. Hence, the number of overall rules remains the same. Furthermore, slicing becomes vastly more complex. If we slice a **node**, we would first slice branch q_1 , and then branches q_2 and q_3 in arbitrary order, where we offset the slicing indices accordingly to the size of q_1 and the desired height and width by the size of the slicing result q'_1 .

Finally, we want to slice q_4 . It becomes clear that the index offsets depend on the structure of the node. If $\mathbf{cols}(q_1) < \mathbf{cols}(q_2)$, we can use the number of columns of q_1 and q'_1 in order to compute the number of columns of q'_4 . If $\mathbf{cols}(q_2) < \mathbf{cols}(q_1)$, as illustrated in Fig. 6.9, then we must use the number of columns of q_2 and q'_2 instead of q_1 and q'_1 .

Furthermore, without additional adjustments, we would be able to construct a new quad rope $\mathbf{node}(q'_1, \epsilon, \epsilon, q'_4)$, $q'_4 \neq \epsilon$ for the case $\mathbf{slice}(0, 0, r, c, q)$, where $q \downarrow \mathbf{node}(q_1, q_2, q_3, q_4)$, $\forall q_i. q_i \neq \epsilon$ and $r \leq \mathbf{rows}(q_1)$, $c \leq \mathbf{cols}(q_1)$, again as illustrated in Fig. 6.9. This happens regardless of the original structure of q . Slicing q_1 results in empty q'_2 and q'_3 . If we naively try to use the size of the latter two to compute the desired size of q'_4 , the result will be a quad rope that has no rectangular shape and therefore cannot be a valid quad rope instance.

One solution to these problems is to introduce additional rules for **slice** with side conditions for handling the above cases, which would complicate the semantics considerably. Another solution is to make this

situation impossible and to duplicate rules for each dimension; which is why we have chosen **hcat** and **vcat** over a four-way **node**.

6.5 A Reference Implementation

We have written a reference implementation of the quad rope data structure in F#. The source code is available online.¹ The inductive type definition for a quad rope with a sparse node constructor looks as follows:

```
type 'a qr =
  | Leaf of 'a [,]
  | HCat of 'a qr * 'a qr
  | VCat of 'a qr * 'a qr
  | Sparse of int * int * 'a
```

Thanks to the immutability of quad ropes, we can implement slicing using views. This allows for constant time slicing, which ultimately allows for a fast implementation of **zipWith** that directly follows the operational semantics from Sec. 6.2.1. Materialization of slices directly follows from the operational semantics for **slice**. We use explicit materialization internally where appropriate:

```
type 'a qr =
  | ... (* As before. *)
  | Slice of int * int * int * int * 'a qr
```

Our quad rope implementation uses the .NET Task Parallel Library [74] and pushes a new task to the work-stealing thread pool for each branch of a HCat or VCat node. Leaf nodes have at most s_{\max} rows and s_{\max} columns. We merge small leafs when their combined number of rows or columns is less than or equal to s_{\max} by copying, as for one-dimensional ropes [24]. The choice of s_{\max} determines the maximum amount of work that can be executed sequentially [8, 101]. Limiting leaf size to s_{\max} allows for a uniform parallelization scheme for all quad rope instances and for fast performance of persistent update.

Most combinators, such as **map** and **reduce**, can be parallalized in a straightforward fashion. Running two closures in parallel by starting

¹<https://github.com/popular-parallel-programming/quad-ropes>

```

1 val par2 : (unit → 'a) → (unit → 'b) → 'a * 'b
2 val mat : int → int → int → int → 'a qr → 'a qr
3
4 let rec map f = function
5   | Leaf xss → Leaf (Array2D.map f xss)
6   | HCat (q1, q2) →
7       HCat (par2 (fun () → map f q1)
8               (fun () → map f q2))
9   | VCat (q1, q2) →
10      VCat (par2 (fun () → map f q1)
11              (fun () → map f q2))
12   | Sparse (r, c, a) → Sparse (r, c, f a)
13   | Slice (r0, c0, r1, c1, q) →
14      map f (mat r0 c0 r1 c1 q)

```

Figure 6.10: A parallel implementation of **map** on quad ropes in F#. Function **mat** materializes a quad rope slice.

new tasks is wrapped in function **par2**, as illustrated in Fig. 6.10. Note that slicing is an inherently sequential operation and hence cannot be performed in parallel.

As noted in Sec. 6.2.2, the operational semantics for **scan** does not exhibit any obvious opportunities for parallelization because each **SCAN** rule is inherently sequential. Still, there are two configurations of **hcat** and **vcat** nodes for which **scan** can be parallelized:

$$\mathbf{hcat}(\mathbf{vcat}(a, b), \mathbf{vcat}(c, d)) \quad (6.8)$$

$$\mathbf{vcat}(\mathbf{hcat}(a, c), \mathbf{hcat}(b, d)) \quad (6.9)$$

Since **scan** computes the prefix sum from the top left of a quad rope to its bottom right, the sequential dependency for these configurations is $a < \{b, c\}$ and $\{b, c\} < d$. If $\mathbf{rows}(c) \leq \mathbf{rows}(a)$ and $\mathbf{cols}(b) \leq \mathbf{cols}(a)$, then there is no sequential dependency between b and c . In that case, b and c may be scanned in parallel, as described earlier in Sec. 3.1.4.

6.5.1 Lazy Tree Splitting Does Not Apply

Lazy tree splitting [8] is a scheduling technique based on lazy binary splitting [106] and uses ropes [24] to represent parallel collections. The basic idea is to spawn new tasks on a by-need basis instead of eagerly:

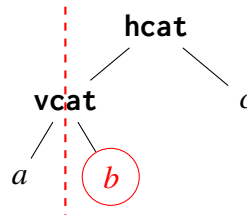


Figure 6.11: Trying to split a quad rope into two valid quad ropes during lazy tree splitting. In this example, the focus is currently on node *b*. The dashed red line marks where we want to split the quad rope in two.

during traversal, whenever a worker thread is idle, a new task is spawned off to handle half of the not yet traversed part of the rope. The check for idle worker threads piggy-backs on efficient work-stealing queues [28] and reads from the global work queue in a non-synchronized fashion. Thereby, no communication overhead is introduced and synchronization happens whenever other functions force synchronization, e.g. worker threads stealing tasks. If the global work queue is empty, it is likely that spawning new tasks will pay off performance-wise [8].

At any time, we must be able to stop execution, store the already performed work and then evenly distribute the remaining work across two tasks. When the work is stored in a random-access array, lazy binary splitting is a matter of adjusting indices; already processed work remains in the target array and the remaining index range is split in two [106].

If the work is stored in some kind of tree, e.g. a rope, we can use a zipper [62] to navigate over the tree and to keep track of the work already performed and what still remains to be done. When a worker thread is idle, we need to stop execution and split the zipper context into the processed part and the remaining part. Afterwards, we can split the remaining part of the tree in two equally sized trees and process them in parallel [8].

Thus, we must be able to take the zipper context apart in an arbitrary fashion and construct two valid trees from it. This is possible on one dimensional ropes, because two ropes can always be concatenated to each other. Unfortunately, this is not the case for quad ropes due to the existence of two concatenation constructors, **hcat** and **vcat**. Figure 6.11 shows an example where the current focus is on the node *b* that is the second argument to a **vcat** node. This means that its left neighbor *a* has already been processed, while the last node *c* is not yet processed.

Due to the side condition of rule `HCAT`, we know that $\text{rows}(a) + \text{rows}(b) = \text{rows}(c)$ and $\text{rows}(a) \neq 0$. If we try to remove a from the quad rope, we cannot use `hcat` to combine the not yet processed leafs since $\text{rows}(b) < \text{rows}(c)$. It follows that lazy tree splitting does not apply to quad ropes, because we cannot split a quad rope into two valid quad rope instances at arbitrary positions during traversal.

Regarding lazy tree splitting, ropes are a special case of quad ropes, where the maximum height or width is fixed at one. Because quad ropes of maximum height 1 can only be concatenated by `vcat`, the problem does not occur and lazy tree splitting is possible again.

As a result, we must rely on the effectiveness of the underlying task parallel library and perform eager splitting at each node.

6.5.2 Balancing

If the quad rope tree is highly imbalanced our recursive parallelization scheme achieves only sequential execution. Moreover, indexing operations would require linear time, which is unacceptable. Hence, a quad rope should always be balanced.

Rebalancing of a binary tree can be implemented via rotation in logarithmic time after insertion or deletion. We use a depth-metric to determine whether to rotate a quad rope, as illustrated by the following function:

```
let rec depth = function
  | Leaf _ | Sparse _ | Slice _ → 0
  | HCat (a, b)
  | VCat (a, b) → (max (depth a) (depth b)) + 1
```

In terms of parallelism, the depth of a quad rope is equal to its span [19] modulo leaf size. Hence, balancing ought to keep the depth low.

A quad rope can be rotated only in limited ways: we can rotate nested `HCat` nodes and nested `VCat` nodes, but not alternating chains of `HCat` and `VCat`:

```
let is_skewed a b c =
  depth a <> depth b
  && max (depth a) (depth b) > depth c

let rec balance = function
```

```

| HCat (HCat (a, b), c) when is_skewed a b c →
    HCat (a, balance (HCat (b, c)))
| VCat (VCat (a, b), c) when is_skewed a b c →
    VCat (a, balance (VCat (b, c)))
| ... (* Mirror cases omitted. *)
| qr → qr (* Otherwise, do nothing. *)

```

This pattern never increases depth. The initial depth is (HCat case):

$$\max(\max(\text{depth } a, \text{depth } b) + 1, \text{depth } c) + 1$$

If `is_skewed` evaluates to true, then at least one of `a` and `b` is deeper than `c`. It follows that the initial depth is equal to

$$\max(\text{depth } a, \text{depth } b) + 2$$

If $\text{depth } a > \text{depth } b$, the balanced quad rope

$$\text{HCat } (a, \text{HCat}(b, c))$$

has a depth of $\text{depth } a + 1$, which is an improvement over the initial depth. If, however, $\text{depth } a < \text{depth } b$, the depth of `b` defines the depth of both, the initial and the balanced quad rope, which is $\text{depth } b + 2$:

$$\text{depth } (\text{HCat } (\text{HCat}(a, b), c)) = \text{depth } (\text{HCat } (a, \text{HCat}(b, c)))$$

Even though we cannot balance across dimensions, it is useful to look into *adversarial* cases in which a quad rope is composed of a chain of alternating HCat and VCat nodes. It is not obvious to us whether this adversarial pattern is common. If one of the branches is a Sparse (i.e. **rep**) leaf, as illustrated in Fig. 6.12, we can use slicing and redistribute the sliced Sparse leaves. Note that this is not possible for HCat or VCat nodes: slicing does not actually reduce the depth of a node and materializing a slice would take $O(n \log n)$ time at each recursive balancing step, where $n = \max(r, c)$ of the resulting $r \times c$ quad rope. With this insight, we can extend the balancing algorithm as follows:

```

let rec balance = function
| ... (* Cases for HCat and VCat omitted. *)
| HCat (VCat(a, b) as q, Sparse(r, c, v))
    when depth q > 2 →
    let s_1 = Sparse(rows a, c, v) in

```

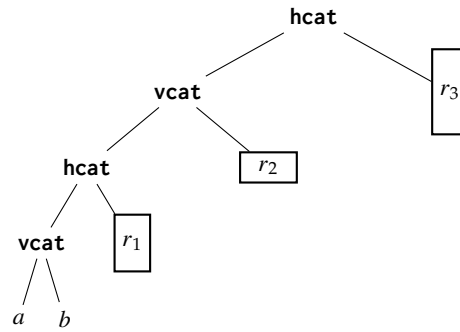


Figure 6.12: A quad rope constructed in adversarial manner without balancing, using **hcat** and **vcap** alternatingly, where the right-hand children r_i are instances of **rep**. The size of the box indicates the shape of the **rep** quad rope that it represents.

```

let s_2 = Sparse(rows b, c, v) in
  VCap (balance (HCap(a, s_1))),
        balance (HCap(b, s_2)))
| VCap (...) (* Swap HCap and VCap. *)
| ... (* Mirror cases omitted. *)
| qr → qr

```

The result of this extended balancing algorithm, applied to the quad rope from Fig. 6.12, is shown in Fig. 6.13. We never perform balancing if the remaining depth of the dense node is less than or equal to 2. The worst-case complexity of balancing a $r \times c$ shaped quad rope is $O(n \log n)$ where $n = \max(r, c)$. Since **balance** is called recursively along the rotated branch and never increases depth, repeated concatenations of quad ropes in the same dimension result in a balanced tree that maintains a balancing invariant at least as strong as the AVL-tree balancing invariant.

6.5.3 Memory Allocation

Enforcing s_{\max} during quad rope initialization via **init** requires allocation of multiple leaf arrays of shape at most $s_{\max} \times s_{\max}$. When implementing **init**(r, c, f), we alternately split row and column counts in two until they are at most s_{\max} . At this point, a naive implementation would allocate a new two-dimensional array as a leaf and initialize it with f for the appropriate offsets. The leaves are then concatenated via **hcat** and **vcap**. This results in $O\left(\frac{r \cdot c}{s_{\max}^2}\right)$ array allocations.

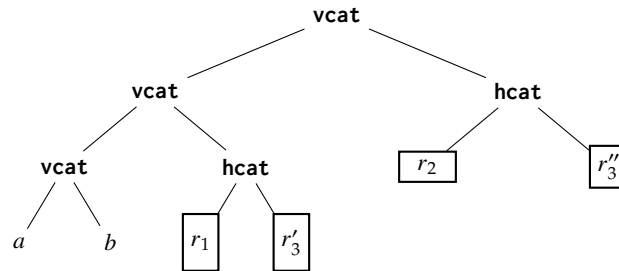


Figure 6.13: The adversarial quad rope from Fig. 6.12 after balancing. The quad rope r'_3 is the “upper two thirds” and r''_3 is the “lower third” of r_3 .

Allocating a single large array on .NET is just as fast as allocating a single small array, but allocating many small arrays is much slower than allocating a single large array. Hence, it pays off to make use of the imperative features of F#: we pre-allocate a single large array that matches the shape of the quad rope and fill it imperatively. We use array pre-allocation to implement all combinators that construct a new quad rope, i.e. **init**, **map**, **zipWith** and **scan**. We store an additional sparseness flag at each node that we check at each recursive step to not unnecessarily allocate an array for sparse branches of a quad rope. Using a single underlying array allows us to implement **scan** without passing explicit prefixes γ , δ and ρ recursively, since all prefix values are accessible in constant time via the underlying array.

We use an immutable view abstraction, a so-called *strided array*, over the underlying array to share it among Leaf nodes. A positive side effect is that we can slice views on arrays in constant time. Hence, materializing a quad rope slice, as discussed in Sec. 6.5, only requires re-allocation of the tree structure, which takes logarithmic time on balanced quad ropes. We define the strided array type as follows:

```

type 'a array_slice = { offset : int;
                        row_stride : int;
                        col_stride : int;
                        rows : int;
                        cols : int;
                        data : 'a [] }

```

All data are stored in the one-dimensional array `data`; this also allows us to transpose and reverse an `array_slice` instance in constant time. We

implement random access on the `array_slice` type by multiplying each argument index with its stride and adding the overall offset to the result:

```
let get xs r c =
  let i = xs.offset
    + r * xs.row_stride
    + c * xs.col_stride in
  xs.data[i]
```

6.6 Performance

We use an extended version of the .NET benchmarking infrastructure by Biboudis et al. [10].² Our test machine is the P3 server (see Sec. 1.3). The presented benchmark results are the mean of 10 runs, preceded by three warm-up runs to trigger JIT compilation. We use automatic iteration count adjustment to guarantee a minimum running time of 0.25 seconds [92]. We choose $s_{\max} = 128$ for all benchmarks.

6.6.1 Individual Functions

Table 6.1 shows average performance results for individual combinators on two-dimensional arrays and quad ropes of unboxed double precision floating-point numbers.

Observation 6.1 *Quad ropes perform slower than standard two-dimensional arrays when the combinator reads from and writes to memory; only **reduce** exhibits a slight speedup of roughly 1.5 times.*

This behavior may partially be due to our strided array implementation: it may access the underlying array in a pattern that inhibits prefetching or that requires frequent cache invalidation when writing. We believe that the improved performance of **reduce** is due to increased locality of reference that comes with grouping values at leaf nodes. The branch-matching logic in **zipWith** may furthermore add some overhead, which could explain the three-fold slowdown.

Figure 6.14 shows the average speedup that the quad rope implementation achieves on our test machine for an increasing number of cores.

²<https://github.com/biboudis/LambdaMicrobenchmarking>.

Benchmark	Arrays (ms)	Quad ropes (ms)	Relative
init	5.94 ± 0.029	5.4 ± 0.205	1.1
map	7.42 ± 0.019	12.29 ± 0.08	0.6
reduce	8.54 ± 0.002	5.66 ± 0.003	1.51
scan	9.47 ± 0.028	21.77 ± 0.095	0.43
zipWith	3.66 ± 0.030	13.63 ± 0.078	0.27

Table 6.1: Average running times of combinators on two-dimensional arrays and quad ropes of double precision floating-point numbers for $s_{\max} = 128$ and size 1000×1000 .

Observation 6.2 *Only **map**, **reduce** and **zipWith** achieve notable speedups for up to 16 cores, of which only **reduce** scales for more than 16 cores as well.*

The speedups for **map** and **zipWith** are moderate and decline for more than 16 cores, which is again likely due to cross-chip communication cost. The **reduce** combinator scales well and gains a nearly 12-fold speedup on 48 cores. Its performance increases slower for more than 16 cores, which is probably due to hyperthreading as well as off-chip communication. The combinators **init** and **scan** do not reach more than a two-fold speedup on any number of cores.

We compare the performance of random access (**get**) and persistent update (**set**) on immutable two-dimensional arrays and quad ropes in Table 6.2. Recall that **set**(xs, i, j, v) returns a new array ys that contains v at index (i, j) and that is otherwise equivalent to xs . Persistent update is more than 1725 times faster on quad ropes than on arrays. In the worst case, persistent update on quad ropes only needs to allocate a new array of size $s_{\max} \times s_{\max}$ and a new tree of logarithmic depth (the remaining branches can be shared with other versions of the quad rope thanks to immutability); on immutable arrays, we must copy the entire array of size $m \times n$ to update a single value.

Note that random access on quad ropes is only one order of magnitude slower. Since **zipWith** is two thirds slower on quad ropes than on two-dimensional arrays, it may be useful to consider a **zipWith** implementation that only traverses one quad rope and indexes into the other without shape matching or slicing.

We do not compare the run-time performance of concatenation: on quad ropes, the asymptotic complexity of concatenation is best-case constant and, due to balancing, worst-case logarithmic in both time and

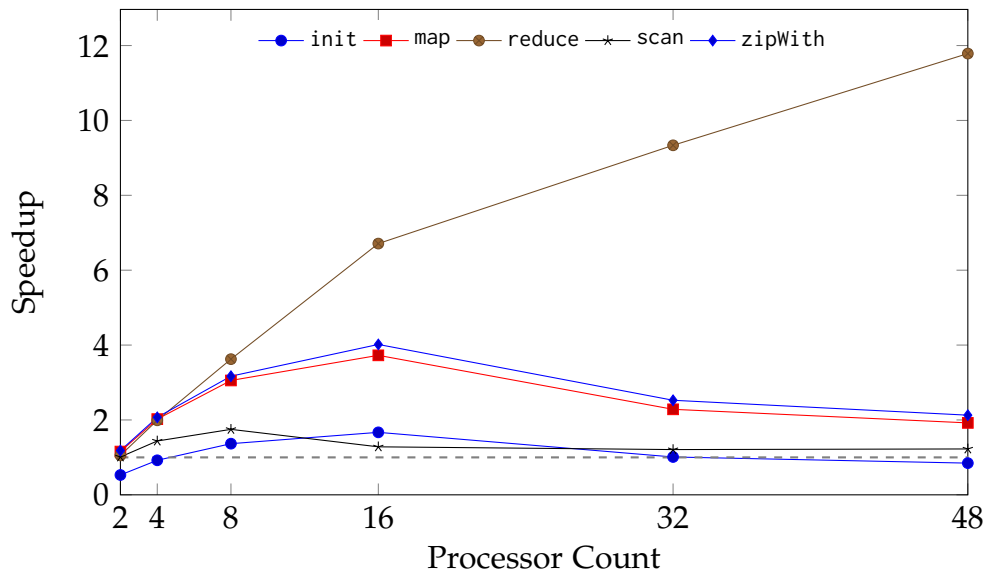


Figure 6.14: Parallel speedup for operations on quad ropes from Table 6.1 for an increasing number of processors. Values are average speedup factors; higher is better. The gray dashed line indicates single core performance.

Benchmark	Relative
get	0.1
set	1725.2

Table 6.2: Average performance of indexing operations on quad ropes with $s_{\max} = 128$ of size 1000×1000 , relative to that of standard immutable two-dimensional arrays; higher is better. Both **get** and **set** use pseudo-randomly generated index pairs.

space; on immutable arrays, it is always linear in time and space. A direct comparison would be unfair, which is why we omit it.

6.6.2 Declarative Algorithms

We use the following declarative algorithms for benchmarking:

- construction of the Fibonacci sequence of length n via recursive concatenation;
- dense matrix multiplication, as described in Sec. 3.2.2 on page 33;

```

1  val single : 'a → 'a qr
2
3  let (@) = hcat
4
5  let next i is =
6      let is' = map (fun j → i + j) is in
7      is @ (single i) @ is'
8
9  let rec vdc n =
10     if n <= 1.0 then single 0.5
11     else
12         let n' = 2.0 ** -n in
13         next n' (vdc (n - 1.0))

```

Figure 6.15: Computing the *van der Corput* sequence in F#.

- sparse matrix multiplication, using upper-triangular sparse quad ropes as arguments to the functional matrix multiplication algorithm for dense matrix multiplication;
- computing a *van der Corput* sequence [108], as shown in Fig. 6.15 [93, Sec. 6.2]; and
- the Smith and Waterman [98] algorithm for sequence alignment, as described in Sec. 4.2.2 on page 47.

The five algorithms represent different aspects of array programming. The algorithms for computing the Fibonacci and the van der Corput sequences recursively concatenate arrays to each other, which we expect to be faster on quad ropes; dense and sparse matrix multiplication exhibit nested regular parallelism; and the Smith-Waterman algorithm is a practical application of the **scan** combinator.

Table 6.3 shows the average performance results of benchmarking the high-level algorithms on two-dimensional immutable arrays and on quad ropes.

Observation 6.3 *Quad ropes outperform standard immutable two-dimensional arrays in three out of five and perform roughly as fast as arrays in two out of five tested algorithms.*

Benchmark	Input size	Arrays (ms)	Quad ropes (ms)	Relative
Fibonacci	1600	7.77 ± 0.004	1.08 ± 0.001	7.16
Matrix mult. dense	1000×1000	1376.14 ± 1.235	1236.62 ± 3.302	1.11
Matrix mult. sparse	1000×1000	–	637.36 ± 0.385	2.16
Smith-Waterman	1000×1000	405.02 ± 1.469	457.17 ± 1.776	0.89
Van Der Corput	20	24.95 ± 0.068	11.31 ± 0.034	2.21

Table 6.3: Average running times and their standard deviation of high-level algorithms on two-dimensional arrays and on quad ropes for $s_{\max} = 128$. We have no algorithm for sparse matrix multiplication on two-dimensional arrays. Therefore, we compare sparse matrix multiplication on quad ropes with dense matrix multiplication on arrays.

Even though individual combinators on quad ropes do not generally outperform arrays, these practical examples demonstrate the benefit of using a quad ropes to represent two-dimensional arrays: declarative algorithms need not be implemented taking the performance of a particular operation, for instance **hcat**, into account.

Figure 6.16 shows the performance for executing functional matrix multiplication with nested parallelism on quad ropes for an increasing number of cores. Parallelism in functional matrix multiplication is nested, because the algorithm calls the parallel **zipWith** combinator in a closure that is passed to the parallel **init** combinator. The graph shows that our recursive parallelization scheme does not scale well with nested parallelism on quad ropes. This may be due to (1) the work-stealing queue being filled with too many short-running tasks whose only job it is to spawn more tasks; and (2) that, because of nested slicing in **init**, there is not enough sequential work to do at each leaf task. We achieve at most a 1.5-fold speedup for dense matrix multiplication. Again, overall performance degrades for more than 16 cores as seen in many previous examples before. What is new here is that the performance already degrades from 8 cores and upwards. This indicates excess parallelism (2), as described above.

6.7 Quad Ropes in Funcalc

6.7.1 Implementation

In addition to the F# reference implementation from Sec. 6.5, we have implemented quad ropes for Funcalc in C#. Instead of an inductive

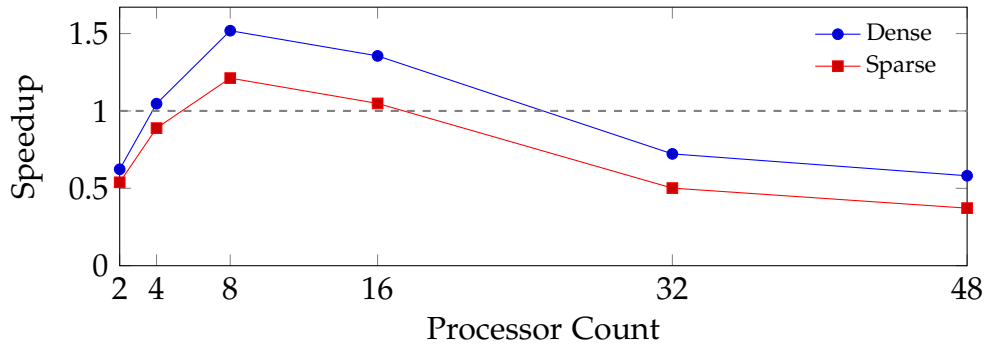


Figure 6.16: Average parallel speedup for dense and sparse matrix multiplication on quad ropes with $s_{\max} = 128$ for an increasing number of processors; higher is better. The dashed gray line indicates 1-core performance.

definition, this implementation uses class inheritance to model quad rope nodes, as shown in Fig. 6.17, using method overloading to implement the particular behavior for each node type according to the operational semantics from Sec. 6.2. Furthermore, we add a new node type `SheetView` that represents the result of a cell range expression, e.g. `A1:C3`. This allows us to avoid copying of otherwise unchanged data, an idea already implemented for `Funcalc` arrays [93].

One obstacle is that `Funcalc`’s `MAP` combinator has variadic arity (see Sec. 3.2.1): it accepts a k -ary function parameter and k arrays of equal shape. Quad ropes would require a new definition for each k to traverse k quad ropes together efficiently. Our implementation only takes measures to handle small values of k efficiently. For $k = 1$, we use `map`; for $k = 2$ we use `zipWith`; and for $k \geq 3$, we use `init` and logarithmic-time indexing via `get`.

6.7.2 Performance

The performance of quad ropes in `Funcalc` is dominated by a new factor: in `Funcalc`, number values are heap-allocated objects of type `NumberValue`. That means that, for every number value that the quad rope contains or that is generated to call a generator function with the appropriate index pair, as in `TABULATE` (see Sec. 3.2.1), memory must be allocated on the heap. Even though these intermediate `NumberValue` objects are likely to be of generation one and are garbage-collected quickly, they dominate the allocation cost compared to allocating quad rope nodes. For instance,

```

1  // The basic type.
2  interface QuadRope<T> {}
3
4  class Leaf<T> : QuadRope<T> {
5      readonly ArraySlice<T> slice;
6  }
7
8  // Cat contains functionality
9  // shared by HCat and VCat.
10 abstract class Cat<T> : QuadRope<T> {
11     readonly QuadRope<T> left, right;
12 }
13
14 class HCat<T> : Cat<T> {}
15 class VCat<T> : Cat<T> {}
16
17 class Sparse<T> : QuadRope<T> {
18     readonly int rows, columns;
19     readonly T value;
20 }
21
22 class Slice<T> : QuadRope<T> {
23     readonly int r0, r1, c0, c1;
24     readonly QuadRope<T> q;
25 }
26
27 // A specialized node type for pointing
28 // at a cell range on a spreadsheet.
29 class SheetView : QuadRope<Value> {
30     readonly int r0, r1, c0, c1;
31     readonly Sheet sheet;
32 }

```

Figure 6.17: A skeleton implementation of quad ropes in C# with an additional node type `SheetView` that represents the result of a cell range expression, e.g. `A1:C3`.

generating a quad rope of shape 100×100 for $s_{\max} = 32$ via TABULATE requires the allocation of 20 000 temporary NumberValue instances to call the initializing function with, 10 000 result values (of type Value) and only 27 quad rope nodes.³

The original Funcalc arrays use .NET's two-dimensional array type [93]. Slicing returns a view of the original array, similar to the quad rope reference implementation. Values in arrays are not unboxed; an array of NumberValue instances is an array of pointers to the memory location of the heap-allocated objects.

Experimental Setup

We compare the performance of quad ropes implemented in Funcalc to Funcalc's standard arrays using the built-in BENCHMARK function [93]. The BENCHMARK function calls a null-ary closure for n iterations and reports the average running time in nanoseconds. Our benchmark spreadsheet performs k times ten iterations for each function and reports the average running time of all iterations. We set k manually to generate useful results.

We do *not* use the P3 server for this benchmark. Our test machine is an Intel Core i7-5600U with two processors at 2.6GHz and 4GB memory, running Windows 10 on Oracle VirtualBox version 5.1.38 r122592. All experiments run on .NET Framework 4.7.1.

Results

Figure 6.18 shows the performance of dense and sparse quad ropes relative to Funcalc's standard arrays [93, Sec. 2.1.1]. Our dense quad rope implementation performs slightly slower than Funcalc's standard arrays for `init`, `map` and `zipWith`. We can see that the performance improvement of `reduce` on quad ropes, gained by increased locality of reference, as discussed in Sec. 6.6.1, is lost in the Funcalc implementation. This is likely due to all values in Funcalc being heap-allocated which results in a total loss of locality.

The input size and changing s_{\max} influence the performance behavior of our quad rope implementation drastically. For instance, running `zipWith` (equivalent to binary MAP in Funcalc) for size 100×100 runs

³NumberValue instances for the values 0 and 1 are interned, which we ignore in this calculation.

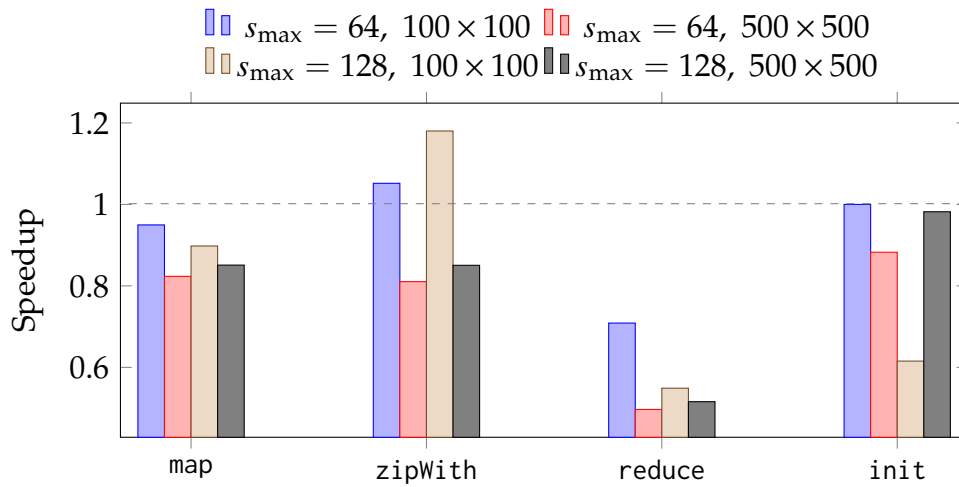


Figure 6.18: Average performance of quad rope combinators for different values of s_{\max} and differently sized inputs, relative to standard Funcalc arrays of equal sizes; higher is better. A value of one indicates equal performance, marked by the dashed gray line.

slightly faster than on standard Funcalc arrays for $s_{\max} = 64$ and roughly 20% slower for $s_{\max} = 128$.

The relative performance of functions that change the shape and structure of a quad rope is shown in Fig. 6.19. The functions `transpose`, `hrev` and `vrev` are between 13 and 14 times faster for $s_{\max} = 64$ and between 32 and 52 times faster for $s_{\max} = 128$. This is likely due to the underlying strided array representation that we use for quad ropes (see Sec. 6.5.3), which allows us to avoid allocating new memory when reshaping a quad rope. Standard Funcalc arrays implement these operations by copying. Slicing and materialization of slices is roughly equally fast for quad ropes and Funcalc arrays, since the latter also uses constant-time views.

6.8 Conclusion

In this chapter, we have presented quad ropes, a two-dimensional extension of the rope data structure [24] that is inspired by quad trees [45] for high-level, functional, parallel array programming. Quad ropes do not generally outperform standard, immutable two-dimensional arrays on a managed platform, but gracefully handle array programming anti-patterns. This makes them a useful array representation in high-level functional languages where expressiveness is of importance.

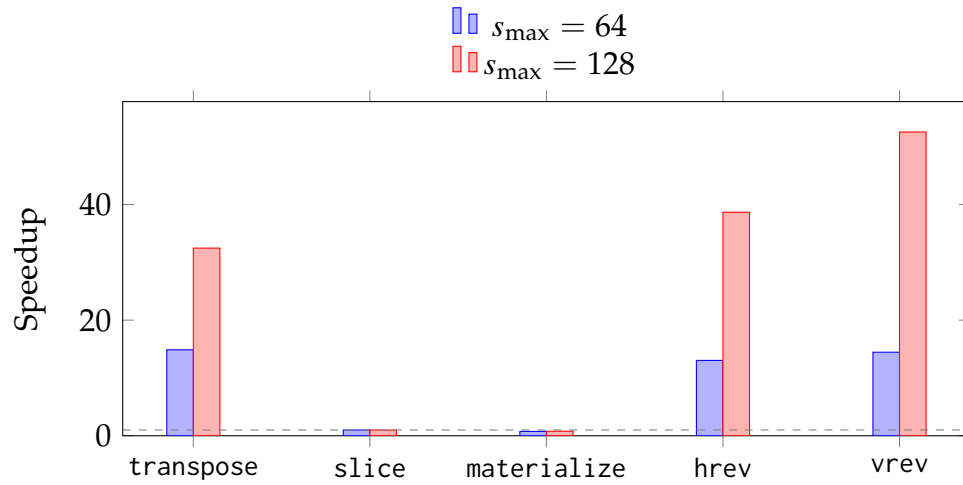


Figure 6.19: Average performance of structural functions on quad ropes of size 500×500 for different values of s_{\max} , relative to standard Funcalc arrays of equal sizes. Higher is better. A value of one indicates equal performance, marked by the dashed gray line.

We have given an operational semantics to describe the quad rope data structure and shown that scheduling of parallel work using lazy tree splitting [8] does not apply to ropes of more than one dimension. We have also shown caveats in the actual .NET implementation, such as the cost of allocating many small arrays, which we suggest to solve via memory pre-allocation. It is possible that a modified pre-allocation mechanism could allow for gradual flattening of quad ropes during traversals in combinators, e.g. `map`. Flattened quad ropes that allow for leaves larger than s_{\max} could allow us to use a lazy scheduling algorithm using pairs of indexes [106] but might affect locality of reference negatively.

Quad ropes must maintain balancing invariants in order to provide the typical performance characteristics of binary trees, such as logarithmic time indexing. Moreover, balancing is required to keep the span low, which also increases potential parallelism.

The performance of higher-order combinators on our F# reference implementation of quad ropes is roughly on par with that of immutable two-dimensional arrays. Concatenation in both dimensions is asymptotically and practically faster than on arrays. Parallel speedups of individual functions are sub-linear; nested parallel operations do not scale. Understanding these results in detail is future work. Nevertheless, sequential

quad ropes outperform two-dimensional immutable arrays in three out of five declarative algorithms.

We also have discussed a Funcalc implementation of quad ropes. The improved locality of reference that we gained in the F# implementation is alleviated by the fact that all values in Funcalc are allocated on the heap. It would require some kind of unboxing analysis to gain the same performance in Funcalc as in our F# implementation.

A major shortcoming of quad ropes is that intermediate quad ropes are always materialized. A high-level algorithm implementation that is straightforward to express is therefore not necessarily fast, or even unacceptably slow when compared to an efficient imperative implementation of the same algorithm. In the next chapter, we will explore an extension of quad ropes that eliminates intermediate arrays also across spreadsheet cells.

Chapter 7

Laziness and Deforestation in Spreadsheets

7.1 Introduction

Fusion, or deforestation [111], is a well researched and frequently applied program transformation (e.g. [27, 35, 68, 70, 73, 77, 85]) that eliminates the materialization of intermediate data structures. We can perform fusion on array combinators as well. For instance, the F# function

```
let map_twice f g xs = map g (map f xs)
```

could instead be implemented as

```
let map_twice' f g xs = map (fun x → g (f x)) xs
```

such that the intermediate array of mapping f over xs is never materialized. Fusion performs this transformation automatically.

Spreadsheets are usually evaluated strictly: the values in all cells must be available, because the user may want to inspect them at any time. This inhibits fusion of operations on arrays across cells. The spreadsheet in Fig. 7.1 illustrates this problem. If we assume that the user is only interested in the sum in cell C4 as a result, then the arrays B1:B3 and C1:C3 would in principle not have to be materialized to compute the end result; but because of the visual spreadsheet model, cells must display their values to the user, so it seems that we cannot eliminate the intermediate arrays.

Eliminating an intermediate array that is referred to multiple times can lead to redundant computation and ultimately to worse performance. Analyzing whether an array is referred to by multiple spreadsheet cells

	A	B	C
1	"a"	{=MAP(CLOSURE("F"), A1:A3)}	{=MAP(CLOSURE("G"), B1:B3)}
2	"b"		
3	"c"		
4			=SUM(C1:C3)

Figure 7.1: A spreadsheet that requires the materialization of the intermediate arrays B1:B3 and C1:C3.

is rather straightforward in Funcalc: we can check the size of the support set of a cell that contains an array in order to determine how many cells depend on the array; if the size of the support set equals one, we could safely eliminate the intermediate array. This leaves us with the challenge of fusing intermediate arrays across cells.

In this chapter, we explore an approach to array fusion that combines laziness with deforestation, an idea rooted in REPA arrays [68], by means of “fusible” thunks that we call *funks*. The main idea is that we remove intermediate arrays where possible and materialize them only if the user requests materialization implicitly—that is, when the user inspects the cells containing the intermediate arrays and they therefore have to be rendered.

In this chapter, we show how we can implement funks as a quad rope constructor (see Chapter 6), drawing ideas from other fusion frameworks [27]. One advantage of funks is that we can use them for computations on normal spreadsheets and when defining and compiling SDFs.

7.2 An Algebra of Array Combinators

When looking at the set of operations over arrays, it quickly becomes apparent that there is an underlying algebra on array combinators. More [83] formulated a theory of arrays based on APL and set theory. Since APL is not higher-order and instead lifts operators to the correct array rank implicitly, we use a different algebra. The algebraic rules of array combinators form the foundation of array fusion. In this section, we describe a small algebra of higher-order array combinators that we use to perform fusion on quad ropes.

We use the array combinators *init*, *map* and *reduce* in a small array language of one-dimensional arrays as defined in Fig. 7.2. For brevity,

n	$:=$	Natural numbers.
f	$:=$	Functions.
e	$:=$	$[e_0, e_1, \dots, e_{m-1}]$ Explicit arrays.
	$ $	$f \ e_1 \ e_2 \ \dots$ Function application.
	$ $	$f \circ g$ Function composition.
	$ $	$e_1 \oplus e_2$ Binary operator application.
	$ $	$\text{init } n \ f$ Initialization.
	$ $	$\text{map } f \ e$ Projection.
	$ $	$\text{reduce } \oplus \ e$ Reduction.

Figure 7.2: An array language for which we define an algebra of array combinators. Note that `reduce` does not take an initializer value; f and g range over functions.

we ignore reshaping operations such as transposition and reversal, even though these have interesting properties as well. Moreover, we assume that `reduce` does not take an initializing value ε and ignore the problems that can arise from this simplification. We also gloss over concatenation.

We give the semantics for the language as equational judgments of the form $e \equiv e'$, meaning an expression e is *computationally equivalent* [51] to another expression e' ; this means that both e and e' evaluate to the same result. The rules in Fig. 7.3 mean the following:

- Rule (e1) defines function composition as $(f \circ g) \ e \equiv f \ (g \ e)$, such that function g is applied to the result of applying function f to expression e . Function application has the usual semantics.
- Rule (e2) states that an expression `init n f` is equivalent to an array of length n that consists of values generated by applying f to the value of the index at each position.
- Rule (e3) states that an expression `map f e` is equivalent to an array where f has been applied to each scalar value from the original array that in turn is equivalent to e .
- Rule (e4) states that an expression `reduce \oplus e` is equivalent to computing the “sum” of all values from the array equivalent to e for the \oplus operator.

$$\begin{aligned}
& \text{(e1)} \frac{}{(f \circ g) e \equiv f(g e)} \\
& \text{(e2)} \frac{}{\text{init } n f \equiv [f 0, f 1, \dots, f (n-1)]} \\
& \text{(e3)} \frac{e \equiv [v_0, \dots, v_{n-1}]}{\text{map } f e \equiv [f v_0, \dots, f v_{n-1}]} \\
& \text{(e4)} \frac{e \equiv [v_0, v_1, \dots, v_{n-1}]}{\text{reduce } \oplus e \equiv v_0 \oplus v_1 \oplus \dots \oplus v_{n-1}}
\end{aligned}$$

Figure 7.3: Evaluation semantics for expressions in e . We omit semantics for function and operator application for brevity; they are as usual for closed expressions.

We can use this equational semantics to derive some interesting equivalences that allow us to eliminate intermediate arrays and instead to generate a new expression that is computationally equivalent to the original expression. A canonical and straightforward example is the fusion of two successive applications of `map`:

$$\frac{\frac{e \equiv [v_0, \dots, v_{n-1}]}{\text{map } g e \equiv [g v_0, \dots, g v_{n-1}]}}{\text{map } f (\text{map } g e) \equiv [f(g v_0), \dots, f(g v_{n-1})]}$$

It is possible to use the rule for function composition (e1) to eliminate the intermediate array as follows:

$$\frac{e \equiv [v_0, \dots, v_{n-1}] \quad (f \circ g) v_i \equiv f(g v_i)}{\text{map } (f \circ g) e \equiv [f(g v_0), \dots, f(g v_{n-1})]}$$

This gives rise to the first algebraic rule on array combinators:

$$\text{map } f \circ \text{map } g \equiv \text{map } (f \circ g) \quad (7.1)$$

There are two additional algebraic rules of importance that we can derive from the equivalence semantics: (1) when we map over an initialized

array, we can move the mapped function into the original initializer; and (2) when we reduce an initialized array we can avoid to materialize an array altogether.

Let us begin with (1). We have the following tree of equivalences:

$$\frac{\text{init } n \ g \equiv [g \ 0, g \ 1, \dots, g \ (n-1)]}{\text{map } f \ (\text{init } n \ g) \equiv [f(g \ 0), f(g \ 1), \dots, f(g \ (n-1))]}$$

If we move f into the init combinator using function composition we can eliminate the intermediate array via rule (e1):

$$\text{init } n \ (f \circ g) \equiv [f(g \ 0), f(g \ 1), \dots, f(g \ (n-1))]$$

Hence, the following equivalence holds:

$$\text{map } f \ (\text{init } n \ g) \equiv \text{init } (f \circ g) \ n \quad (7.2)$$

Finally, let us look at (2) and eliminate intermediate arrays altogether. This is useful when the program uses arrays only as a vehicle to structure a computation but never uses the array itself as a result. This is a combination of init and reduce :

$$\frac{\text{init } n \ f \equiv [f \ 0, f \ 1, \dots, f \ (n-1)]}{\text{reduce } \oplus \ (\text{init } n \ f) \equiv f \ 0 \oplus f \ 1 \oplus \dots \oplus f \ (n-1)}$$

We can therefore rewrite a reduction of an initialization to the sum of unpacked values:

$$\text{reduce } \oplus \ (\text{init } n \ f) \equiv f \ 0 \oplus f \ 1 \oplus \dots \oplus f \ (n-1) \quad (7.3)$$

This small algebra, consisting of Equations (7.1), (7.2) and (7.3), is sufficient to eliminate a large number of intermediate arrays.

However, we have glossed over more complicated combinators, such as zipWith and scan . We can implement zipWith by means of init and indexing, which we have omitted to define for our small array language. In that case, our algebra also allows for fusion of zipWith . Adding scan is slightly more complicated; if our language had a mapReduce combinator that were to accept a start-value ε , we could implement scan using array concatenation and indexing. However, such an implementation would be inefficient on immutable arrays.

Now that we have convinced ourselves of the existence of these three basic equivalences that make up a small algebra of array combinators, we will proceed to adapt these to quad ropes in the following section.

7.3 Funky Quad Ropes

We have implemented fusible thunks for quad ropes in OCaml [75]. The type system of the F# language is not expressive enough to model our implementation the way we did in Sec. 6.5. In F#, every type variable that occurs in the constructors of an algebraic data type (ADT) must also occur in the type of the ADT itself. This makes modeling delayed execution rather cumbersome. Generalized algebraic data types (GADTs) in OCaml are more flexible, such that a constructor can have a type parameter that does *not* occur in the type of the GADT [29]. While it would be possible to use F#'s object system, we have chosen to use OCaml for better readability.

In the following, we use a simplified quad rope implementation that does not have a special constructor for slices. Note that OCaml and F# have similar syntax; we only use OCaml from this point on and highlight keywords in **red**.

First, we define a quad rope data type `qr` as a GADT and add a new constructor `Funk`:

```
type _ qr =
  | Leaf    : 'a array2d      → 'a qr
  | HCat    : 'a qr * 'a qr → 'a qr
  | VCat    : 'a qr * 'a qr → 'a qr
  | Sparse  : int * int * 'a → 'a qr
  | Funk    : (int → int → 'a → 'b)
               * 'a qr
               * 'b qr lazy_t → 'b qr
```

The `Leaf`, `HCat`, `VCat` and `Sparse` constructors are just as before (see Sec. 6.5). Instead, focus on the `Funk` (`f`, `p`, `t`) constructor that is a *non-strict* representation of a mapping over a quad rope. It takes three arguments: (1) a function `f` from row and column indices and some value of type `'a` to a value of type `'b`, (2) a *source* quad rope `p` of type `'a` and (3) a *thunk* `t` of type `'b qr lazy_t`. Note that the type parameter `'a` does not occur in the returned type `'b qr`.

The `Funk` constructor models a quad rope at two stages: (1) before it has been computed, represented by a function that is going to be applied to each of the scalar values of the source quad rope; and (2) a possibly materialized result of exactly this computation.

The **lazy** keyword constructs a thunk of type `'a lazy_t` from an expression of type `'a`. Function `Lazy.force` of type `'a lazy_t → 'a` extracts the result from a thunk by evaluating the expression on demand and caches the result such that the function will only be evaluated once. The function `Lazy.is_val` of type `'a lazy_t → bool` returns true if a thunk has already been evaluated and false otherwise.

The overall idea is to keep a funk *unmaterialized* as long as possible, i.e. avoid to execute `Lazy.force` on its thunk. If the thunk `t` from a Funk `(f, p, t)` instance has been evaluated, i.e. it is *materialized*, every computation that refers to this Funk will read the cached value from the thunk `t`. If the thunk has not yet been evaluated, we will instead fuse whichever operation is applied to the Funk instance with the stored function `f` and keep the source quad rope `p`.

7.3.1 Non-Strict Map

We implement two versions of each combinator on quad ropes: a *strict* one, as described in Sec. 6.5; and a *non-strict* version that performs fusion on Funk instances. The core combinator for successful fusion is a variant of `map` that we call `mapi`. Its signature is

```
val mapi : (int → int → 'a → 'b) → 'a qr → 'b qr
```

and it applies its argument function not only to each scalar value in the quad rope, but also to the according index pair. In the following, we prefix strict quad rope combinators that do not delay execution with `Strict`; for instance, `Strict.map` is the strict map combinator. We will not detail strict combinators and assume they are implemented as in Sec. 6.5.

It is useful to begin with defining a composition operator for ternary functions of type `int → int → 'a → 'b`:

```
let (◦) f g = fun r c x → f r c (g r c x)
```

The `◦` operator is a shorthand for an anonymous function that applies function `g` to row- and column-arguments `r` and `c` and an argument `x` of type `'a` and applies function `f` to the result, where `f` also accepts `r` and `c` as arguments. If `g` has type `int → int → 'a → 'b` and `f` has type `int → int → 'b → 'c`, then the expression `f ◦ g` has type `int → int → 'a → 'c`.

Using the `◦` composition operator, the non-strict variant of `mapi` can be implemented as follows:

```

let rec mapi f = function
  | Funk (g, p, thunk) →
    if Lazy.is_val thunk then
      mapi f (Lazy.force thunk)
    else
      mapi (f ∘ g) p
  | q → Funk (f, q, lazy (Strict.mapi f q))

let map f = mapi (fun _ _ x → f x)

```

We distinguish between three cases, but there is only one case where we actually perform a fusion of functions by composition. Let us begin from the bottom. If the argument quad rope q is not a Funk, we construct a new Funk that records the strict mapping of function f over q .

If the argument quad rope is a Funk, we must distinguish between a materialized and a not yet materialized Funk. If the quad rope is materialized, there is no point in performing function composition; in that case, it is sufficient to recursively call `mapi` on the cached value.

If the Funk is not yet materialized, we can again use `mapi` to generate a new Funk, but we apply it to the source quad rope p . The function that we map is the composition of functions f and g , namely $f \circ g$.

This implementation makes sure that a quad rope consists of at most one Funk node in direct succession when applying `mapi` multiple times. Even though `mapi` is recursive, its call depth is bounded by the number of directly nested Funk instances, which is at most one.

7.3.2 From Init to Reduce

We borrow ideas from Keller et al. [68] and use a sparse quad rope of type `unit` as a source to call `mapi` on when implementing non-strict `init` with the following type signature:

```

val init : int → int → (int → int → 'a) → 'a qr

```

The Sparse constructor takes constant time and space and therefore is not an actually materialized quad rope. We can use non-strict `mapi` to map a function over a Sparse quad rope node that ignores the `unit` value, written as `()`, and only uses the shape of the quad rope to structure the call to `mapi`:

```

let init rows cols f =
  let p = Sparse (cols, rows, () ) in
  let g = fun r c _ → f r c in
  mapi g p

```

Now, we can construct a quad rope and map over it without ever having to materialize it. This is especially useful when we construct a quad rope that we later want to reduce.

The reduce combinator cannot be implemented in a non-strict fashion, because it does not return a new quad rope. Nevertheless, it finalizes the deforestation process by exploiting the fact that a Sparse quad rope has no internal structure. Therefore, we can loop over its rows and columns and generate the value for each row and column index on the fly. We use the function `loop_reduce` to implement this behavior. Its type signature is:

```

val loop_reduce = (int → int → 'a → 'b) →
                  ('b → 'b → 'b) →
                  int → int → 'a → 'b

```

The function argument `f` generates the value for each index pair. Our implementation uses OCaml's pointer type `'a ref` that can be updated via the `:=` operator:

```

let loop_reduce f g e (Sparse (r, c, x)) =
  let acc = ref e in (* Mutable accumulator. *)
  for i=0 to r-1 do
    for j=0 to c-1 do
      acc := g !acc (f i j x);
    done;
  done;
  !acc

```

We allow the sparse value `x` to be of any type `'a`; this is the most general type and allows us to use `loop_reduce` on sparse quad ropes of all types.

To be able to call whatever function `f` is stored in a `Funk (f, p, t)` instance, we need to implement `reduce` via a combinator `mapi_reduce` that works like a combination of `mapi` and `reduce`. Its type signature is:

```

val mapi_reduce : (int → int → 'a → 'b) →
                  ('b → 'b → 'b) →
                  'a qr → 'b

```

In the Sparse case, we call `loop_reduce`. In the Funk case, we must again decide whether we should compose the stored function `k` and the mapped function `f` or whether we can read the cached result of the computation and only map `f`. In all other cases, we resort to the strict implementation of `mapi_reduce`:

```
let rec mapi_reduce f g e = function
  | Sparse (r, c, x) as s → loop_reduce f g e s
  | Funk (k, p, t) →
    if Lazy.is_val t then
      mapi_reduce f g e (Lazy.force t)
    else
      mapi_reduce (f ∘ k) g e p
  | q → Strict.mapi_reduce f g e q

let map_reduce f = mapi_reduce (fun _ _ x → f x)
let reduce = map_reduce (fun x → x)
```

High-level expressions that generate and reduce arrays, such as `sum (init 1 100 (+))` which computes $\sum_{i=0}^{100} i$, now effectively reduce to for-loops that compute the result in constant space.

7.3.3 Materialization

When do we have to materialize a Funk? Only when we want to read a particular individual scalar; we assume that, if a program reads a particular scalar, it is likely that other scalars will be read as well, or that the same scalar will be read multiple times [68]. The `get` function returns the scalar at a particular row and column index and is straightforward to implement:

```
let rec get = function
  | Funk (_, _, t) → get (Lazy.force t)
  | q → Strict.get q
```

This is particularly interesting if a quad rope is backing an array formula (see Sec. 2.1.2) where individual scalars are read during unpacking. Only if the values are actually unpacked and displayed, an intermediate Funk will be materialized.

7.3.4 Concatenation

Concatenation in either dimension is strict in the sense that it always returns a new HCat or VCat node. By adding the Funk constructor to the quad rope type, we allow for fast concatenation of materialized and delayed quad rope instances. This means that it is possible for quad ropes to contain multiple Funk nodes, but never in direct succession; there is always at least one HCat or VCat node that separates them.

Our implementation of `Strict.mapi f q` performs at most one fusion step if it encounters a nested Funk (g, p, t) node, that is, it fuses the function f with the cached function g if the thunk t has not yet been evaluated and then evaluates the funk strictly. It never returns a new Funk node.

A downside of allowing concatenation with Funk instances is that small leafs cannot be merged to Funk nodes without materializing the latter. This can ultimately lead to a degenerated quad rope with many small or singleton leafs. As a first line of defense against this behavior, we can change the non-strict `init` combinator to return a new Leaf node if the parameters for row and column sizes are both less than s_{\max} , which is the maximum size of a leaf node. Only if the desired shape of the quad rope exceeds $s_{\max} \times s_{\max}$, it returns a Funk node. Moreover, we do not perform fusion on leafs that are smaller than $s_{\max} \times s_{\max}$ in `mapi` and instead call `Strict.mapi` directly in such cases.

The OCaml implementation of `hcat` and `vcat` performs merging of small leafs as described in Sec. 6.5.

7.4 Initial Performance Evaluation

At the time of writing, we have not yet implemented non-strict quad ropes in Funcalc. We therefore present an initial performance evaluation using the OCaml implementation. This implementation does neither balance sub-trees as described in Sec. 6.5.2 nor does it pre-allocate underlying arrays as described in Sec. 6.5.3.

7.4.1 Experimental Setting

We test three high-level algorithms on arrays: computing Pearson's correlation coefficient [87] (Pearson's), a simple prime sieve (Primes) and computing a *van der Corput* sequence [108]. We do *not* use the P3 server.

Our test machine is an Intel Core i7-5600U at 2.6GHz with two physical processors and 8 GB memory, running Ubuntu 16.04.4 LTS 64-bit and OCaml version 4.02.3. We use the `Core_bench` library for benchmarking.¹

7.4.2 Results

The results of our initial evaluation are displayed in Fig. 7.4. The displayed values are average running time for multiple iterations and for the given input size n . The `Core_bench` library chooses an appropriate iteration count automatically and ensures an overall running time of at least ten seconds.

For both, Pearson's and Primes, we can see that the larger the input becomes, the better non-strict quad ropes perform in comparison. For two quad ropes of size 1000×1000 , the non-strict quad rope implementation computes Pearson's correlation coefficient about 8.4 times faster. For Primes to exhibit a visible performance speedup, we have to increase the input size by another order of magnitude. Still, the speedup here is only roughly 2.5 times.

The algorithm for computing the van der Corput sequence is the worst case for non-strict quad ropes due to repeated recursive concatenation of singletons in the auxiliary function `next`, as illustrated in Fig. 6.15 on page 119. However, because we only allow fusion on quad ropes that are larger than $s_{\max} \times s_{\max}$ and because we merge materialized leafs, non-strict quad ropes are able to compete with strict quad ropes.

7.5 Limitations

We have already encountered one limitation: repeatedly concatenating small leafs to Funk nodes can lead to a degenerate quad rope. This is problematic for performance, because traversals of the entire quad rope take longer. A degenerate quad rope can lead to excess parallelism, because there is not enough work to be performed sequentially at each leaf.

Moreover, fusing additional array combinators becomes increasingly difficult. For instance, fusion of the `zip_with` combinator with the type signature

¹https://blog.janestreet.com/core_bench-micro-benchmarking-for-ocaml.

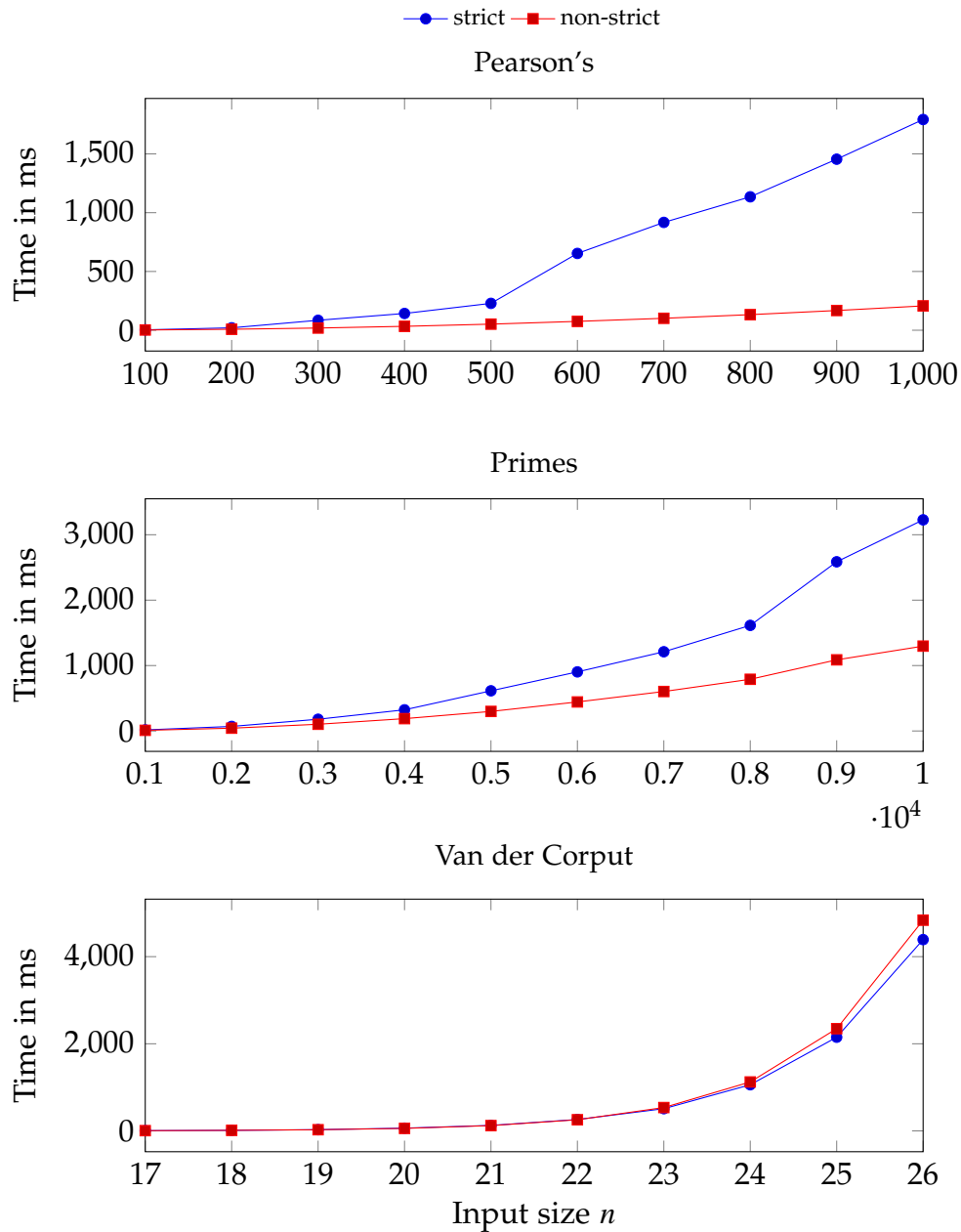


Figure 7.4: Average running time of using strict and non-strict quad ropes in different declarative algorithms. Each configuration is run repeatedly for at least ten seconds; the reported time is the overall running time divided by performed iterations; less is better. Algorithms *Pearsons* and *Primes* are both run on quad ropes of size $n \times n$; *Van der Corput* is called with n directly.

```
val zip_with : (int → int → 'a → 'b → 'c) →
               'a qr → 'b qr → 'c qr
```

would in principle require another non-strict node type. We can implement a somewhat efficient approach by using `mapi` with logarithmic-time access to construct a new anonymous function of the correct type that can be stored in a Funk node; however, this is only efficient if the quad rope that we index into is a not yet materialized Funk instance: we can use its stored function `f` to compute the value on demand during traversal. This is how we have implemented fusion of `zip_with` for our benchmarks in Sec. 7.4.

We have not yet defined fusion for the `scan2d` combinator. Recall that it is defined recursively and that the value at index (i, j) depends on the values at $(i - 1, j)$, $(i - 1, j - 1)$ and $(i, j - 1)$ and that we initialize the fringes with the values γ , δ and ρ (see Sec. 3.1.4). The function that we apply with `scan2d` has type $'a \rightarrow 'a \rightarrow 'a \rightarrow 'b \rightarrow 'a$. If `scan2d` is the last combinator that we apply to a succession of map applications, there is no problem: we can compose the mapped functions with the scanned function.

Fusion of two successive applications of `scan2d` is more complicated. There are two apparent problems: (1) we must retain all initialization values γ , δ and ρ for each application of `scan2d`; and (2), each function that we scan for must somehow be able to access all of the intermediate values from the previous applications at indexes $(i - 1, j)$, $(i - 1, j - 1)$ and $(i, j - 1)$. It seems that requirement (2) entirely defeats the purpose of fusing successive applications of `scan2d`: either we store intermediate values, or we have to recursively recompute them at every index. Therefore, we suggest to implement `scan2d` as a strict combinator only.

7.6 Related Work

As noted in Sec. 7.1, fusion is a well-researched topic in functional programming, e.g. [27, 35, 68, 70, 73, 77, 85], and there exist much more sophisticated fusion frameworks for lists, arrays and arbitrary recursive data structures than the one presented in this chapter. However, none of these allow for “going back” and to materialize eliminated intermediates. In standard functional programming languages, this is usually not a problem: intermediate values cannot be inspected interactively the way they can in spreadsheets.

Laziness is a recurring topic in spreadsheet systems research but seems not to have been investigated in much detail. FunSheet [37] is a spreadsheet system implemented in the Clean language [25] which shares many properties with Haskell. Clean is purely functional and non-strict; as a result, all computations in FunSheet are non-strict, too. This is an approach that we could adapt in Funcalc as well. However, it seems wasteful to delay scalar computations in spreadsheets. In Sec. 5.4.4, we have shown statistics that suggest that, on large spreadsheets, each cell takes on average less than a millisecond to compute. It is reasonable to assume that it would take equally long to allocate a thunk.

Lisper and Malmström [78] describe Haxcel, a Haskell interface for Microsoft Excel, where users can write Haskell expressions that operate over cell ranges. Haskell is a non-strict language and hence expressions in Haxcel are non-strict, too. However, the authors are not concerned with performance but expressiveness and do not discuss whether or how fusion or laziness impact Haxcel.

Vaziri et al. [109] implemented ActiveSheets², a system for processing streams of “live” data in spreadsheets. The idea is that a user can use a spreadsheet to construct a stream processing pipeline for arbitrary data sources, such as for instance constantly changing stock market prices. Moreover, ActiveSheets provides live views of remote data sources. Even though operations on streams are fusible, the authors do not focus on fusion, laziness or other performance-related topics.

7.7 Conclusion

In this chapter, we have presented a small framework for fusion on quad ropes that could be used to perform fusion of array combinators even across rows or columns in Funcalc. Intermediate arrays can remain unmaterialized until the user explicitly inspects them by navigating to the containing cells, which will force the materialization of intermediates on-demand. We have not implemented this technique in Funcalc, but presented a small proof-of-concept implementation in OCaml that exhibits a notable performance improvement for some high-level algorithms and gracefully handles worst-case scenarios.

Our approach to fusion on quad ropes is unified in the sense that it works for compiled functions, as demonstrated in Sec. 7.4, and it is likely

²Not to be confused with ActiveSheets by Kotler et al. [71].

that it will work equally well on interpreted spreadsheets. Further research, however, is necessary to validate this claim. Moreover, it remains to be shown how our fusion approach combines with parallelization, proper balancing of quad rope trees and pre-allocation of memory. In particular, parallelization would require that thunks are thread safe and that each thunk is at most evaluated once.

Nevertheless, this seems to be the first attempt at cross-cell fusion for improving spreadsheet performance in literature.

Chapter 8

Conclusion

In this dissertation, we have explored the design space of implicitly parallelizing spreadsheet programs on shared-memory multicore processors using techniques from data-parallel functional programming. Table 8.1 classifies the contributions of this thesis into static or dynamic approaches that extract parallelism locally or globally by chapter. In this chapter, we will summarize the results from each contribution and highlight open problems and future work in **bold**.

In Chapter 4, we described *cell array rewriting*, a *static* approach to extracting *local* parallelism from array-like structures in spreadsheets. We have used the observation that certain naturally occurring, high-level cell structures correspond to higher-order array combinators [112] to systematically rewrite spreadsheet formulas to higher-order expressions on arrays. Our approach achieves up to 25 times speedup on a 48 core Intel Xeon processor. A downside of our technique is that the rewriting

	Static	Dynamic
Local	Cell array rewriting, Chapter 4.	Quad ropes, Chapter 6 and 7.
Global		Puncalc, Chapter 5.

Table 8.1: A classification of the contributions of this thesis into a design-space matrix of parallel spreadsheet programming. We can distinguish between static and dynamic approaches that parallelize spreadsheets locally or globally. This thesis has not explored the design space for static approaches to global parallelization of spreadsheets.

algorithm has to ensure statically that no cyclic dependencies will be introduced during rewriting by traversing the dependency graph of the spreadsheet. This check currently takes roughly as long as an entire recalculation of the target spreadsheet, even when using simplifying heuristics. An open problem is **whether the performance of the preemptive detection of cyclic references can be improved**, for instance by exploiting the underlying support graph data structure [93].

In Chapter 5, we presented Puncalc, a *dynamic* dataflow approach to extracting *global* parallelism from large spreadsheets during evaluation. Our evaluation strategy relies on a work-stealing thread pool implementation [28, 74]. Our main contribution is the specification of an adequate termination condition for the evaluation algorithm, a thread safety scheme that allows for lock-free reads of cached values and a method for detecting cyclic references during parallel evaluation. A thread may be allowed to speculatively evaluate a cell dependency recursively if the result of the dependency does not become available within a predefined delay. To our knowledge, this is the first description of a parallel spreadsheet evaluation strategy that detects cyclic references dynamically. We achieve up to 16 times speedup on 48 cores without having to spend time on a prior analysis.

However, choosing the length of the delay for speculative evaluation statically can lead to excessively many speculative evaluations: if a cell takes substantially longer to compute than the specified delay, there is a high chance that many threads may attempt to evaluate it speculatively, wasting computing time that instead could be used on other parallel tasks. This is likely why combining cell array rewriting (Chapter 4) with Puncalc (Chapter 5) does not yield good parallel speedups. **Choosing the delay for speculative evaluation dynamically** is an open problem. It may be useful to employ a cost model to estimate an appropriate delay for each cell dynamically. Moreover, we have shown that neither the count of formula cells in a spreadsheet, the number of dependencies between them nor the spreadsheet's span is a good predictor for parallel performance. **Developing a useful predictor for parallelizability of a particular spreadsheet** remains future work.

In Chapter 6 we described the quad rope data structure, a high-level representation of immutable two-dimensional arrays for end-user programming that *dynamically* extracts *local* parallelism. Quad ropes allow for block-sparseness, constant-time slicing and fast concatenation in row- and column-direction. Hence, quad ropes gracefully handle

typical array-programming anti-patterns and thereby allow for more expressive algorithms; the idea is that end-users need not to worry about choosing the “right” data structure for their programs and instead can focus on solving domain-specific problems using a good “all-rounder” data structure for every task.

Even though quad rope combinators do not generally outperform array combinators, they improve the performance of algorithms that construct arrays recursively without being noticeably slower on other kinds of algorithms. Individual parallel quad rope combinators scale well; however, experiments with functional matrix multiplication suggest that nested parallel operations on quad ropes do not. **Gradual flattening could enable a lazy splitting approach to scheduling** to improve the performance of nested parallel operations. What remains to be shown is **the feasibility of using quad ropes as high-level array representations in a spreadsheet model of computation** via user studies.

In Chapter 7 we presented a fusion framework that allows for elimination of intermediate arrays by recording an application of an array combinator as a closure and a thunk in a concept that we call *funk*. When an array is not materialized, we access the recorded closure and perform fusion via function composition; if it is materialized, we access the cached result of the thunk. The idea is to delay array computations across spreadsheet cells and to only materialize intermediate arrays when the user implicitly requires them to by inspecting their values. We described our prototype implementation of funks on quad ropes and show that our fusion framework improves the performance of declarative algorithms that users could implement as a sheet-defined function (SDF). While our approach seems promising, **implementing funks in Funcalc, assessing their feasibility and evaluating their performance in a full spreadsheet context** remains future work. Moreover, we have not yet investigated how our fusion framework works together with parallelization.

The field for *global* and *static* techniques in Table 8.1 is intentionally left blank; it is being studied in Alexander Asp Bock’s PhD project.

In conclusion, we believe that we have demonstrated the feasibility of applying techniques from data-parallel functional programming, in particular array programming, to a spreadsheet model of computation. When recalculating highly parallel spreadsheets with the appropriate technique, we are able to achieve up to 25 and 16 times parallel speedup without putting additional work on the end-user. Moreover, by freeing users from performance considerations when implementing declarative

algorithms on arrays, we allow them to write expressive algorithms that perform reasonably fast, even if they use array-programming anti-patterns. We believe that the implicit parallelization of spreadsheet calculations, together with expressive sheet-defined functions, will promote the spreadsheet paradigm as a serious computational platform and allow millions of end-users and domain experts to build even more advanced spreadsheet models without sacrificing performance for ease of use.

Bibliography

- [1] Robin Abraham and Martin Erwig. Inferring Templates from Spreadsheets. In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, pages 182–191, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134312. URL <http://dx.doi.org/10.1145/1134285.1134312>.
- [2] D. Abramson and P. Roe. Parallel execution mechanism for spreadsheets, December 2001. URL <https://www.google.com/patents/US20010056440>. US Patent App. 09/891,951.
- [3] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, October 1989. doi: 10.1145/69558.69562. URL <http://doi.acm.org/10.1145/69558.69562>.
- [4] John Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359579. URL <http://dx.doi.org/10.1145/359576.359579>.
- [5] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. FlashRelate: Extracting Relational Data from Semi-structured Spreadsheets Using Examples. *SIGPLAN Not.*, 50(6):218–228, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737952. URL <http://dx.doi.org/10.1145/2813885.2737952>.
- [6] K. E. Batcher. Sorting Networks and Their Applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring)*, pages 307–314, New York, NY,

- USA, 1968. ACM. doi: 10.1145/1468075.1468121. URL <http://dx.doi.org/10.1145/1468075.1468121>.
- [7] Lee Benfield. FMD: Functional Development in Excel. In *Proceedings of the 2009 Video Workshop on Commercial Users of Functional Programming: Functional Programming As a Means, Not an End*, CUFP '09, New York, NY, USA, 2009. ACM. doi: 10.1145/1668113.1668121. URL <http://doi.acm.org/10.1145/1668113.1668121>.
- [8] Lars Bergstrom, Mike Rainey, John Reppy, Adam Shaw, and Matthew Fluet. Lazy Tree Splitting. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 93–104, New York, NY, USA, 2010. ACM. doi: 10.1145/1863543.1863558. URL <http://doi.acm.org/10.1145/1863543.1863558>.
- [9] Robert Bernecky and Sven-Bodo Scholz. Abstract Expressionism for Parallel Performance. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 54–59, New York, NY, USA, 2015. ACM. doi: 10.1145/2774959.2774962. URL <http://doi.acm.org/10.1145/2774959.2774962>.
- [10] Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Clash of the Lambdas, July 2014. URL <http://arxiv.org/abs/1406.6631>.
- [11] Florian Biermann. Declarative Parallel Programming in Spreadsheet End-User Development: A Literature Review. Technical Report TR-2016-192, IT University of Copenhagen, February 2016. URL https://pure.itu.dk/portal/files/80807389/ITU_TR_2016_192.pdf.
- [12] Florian Biermann and Alexander A. Bock. Puncalc: Task-Based Parallelism and Speculative Reevaluation in Spreadsheets. In *High-Level Parallel Programming - HLPP 2018*, July 2018. To appear.
- [13] Florian Biermann and Peter Sestoft. Quad Ropes: immutable, declarative arrays with parallelizable operations. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY 2017*, pages 1–8. ACM Press, 2017. ISBN 9781450350693. doi: 10.1145/3091966.3091971. URL <http://dx.doi.org/10.1145/3091966.3091971>.

- [14] Florian Biermann, Wensheng Dou, and Peter Sestoft. Rewriting High-Level Spreadsheet Structures into Higher-Order Functional Programs. In Francesco Calimeri, Kevin Hamlen, and Nicola Leone, editors, *Practical Aspects of Declarative Languages*, volume 10702 of *Lecture Notes in Computer Science*, pages 20–35. Springer International Publishing, 2018. doi: 10.1007/978-3-319-73305-0_2. URL http://dx.doi.org/10.1007/978-3-319-73305-0_2.
- [15] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, Mar'ia J. Garzarán, David Padua, and Christoph von Praun. Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '06, pages 48–57, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: 10.1145/1122971.1122981. URL <http://dx.doi.org/10.1145/1122971.1122981>.
- [16] G. E. Blelloch. Prefix Sums and Their Applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990. URL http://repository.cmu.edu/compsci/2018/?utm_source=repository.cmu.edu%2Fcompsci%2F2018&utm_medium=PDF&utm_campaign=PDFCoverPages. Also appears in *Synthesis of Parallel Algorithms*, Reif (ed.), Morgan Kaufmann, 1993.
- [17] Guy E. Blelloch. Scans as primitive parallel operations. *Computers, IEEE Transactions on*, 38(11):1526–1538, November 1989. ISSN 0018-9340. doi: 10.1109/12.42122. URL <http://dx.doi.org/10.1109/12.42122>.
- [18] Guy E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Technical report, Pittsburgh, PA, USA, 1993. URL <http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-95-170.html>.
- [19] Guy E. Blelloch. Programming Parallel Algorithms. *Commun. ACM*, 39(3):85–97, March 1996. ISSN 0001-0782. doi: 10.1145/227234.227246. URL <http://dx.doi.org/10.1145/227234.227246>.
- [20] Guy E. Blelloch. Functional Parallel Algorithms. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional*

- Programming*, ICFP '10, page 247, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863579. URL <http://dx.doi.org/10.1145/1863543.1863579>.
- [21] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 213–225, New York, NY, USA, 1996. ACM. ISBN 0-89791-770-7. doi: 10.1145/232627.232650. URL <http://dx.doi.org/10.1145/232627.232650>.
 - [22] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-parallel Language. *SIGPLAN Not.*, 28(7):102–111, July 1993. ISSN 0362-1340. doi: 10.1145/155332.155343. URL <http://dx.doi.org/10.1145/155332.155343>.
 - [23] Alexander Bock. A Literature Review of Spreadsheet Technology. Technical Report TR-2016-199, IT University of Copenhagen, November 2016. URL https://pure.itu.dk/portal/files/81449279/ITU_TR_2016_199.pdf.
 - [24] Hans-J Boehm, Russ Atkinson, and Michael Plass. Ropes: an Alternative to Strings. *Software – Practice & Experience*, 25:1315–1330, December 1995. doi: 10.1002/spe.4380251203. URL <http://dx.doi.org/10.1002/spe.4380251203>.
 - [25] T. H. Brus, M. C. J. D. Eekelen, M. O. Leer, and M. J. Plasmeijer. *Clean — A language for functional graph rewriting*, volume 274, chapter 20, pages 364–384. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987. ISBN 978-3-540-18317-4. doi: 10.1007/3-540-18317-5_20. URL http://dx.doi.org/10.1007/3-540-18317-5_20.
 - [26] Rommert J. Casimir. Real Programmers Don'T Use Spreadsheets. *SIGPLAN Not.*, 27(6):10–16, June 1992. ISSN 0362-1340. doi: 10.1145/130981.130982. URL <http://dx.doi.org/10.1145/130981.130982>.
 - [27] Manuel M. T. Chakravarty and Gabriele Keller. Functional Array Fusion. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, volume 36 of ICFP '01, pages

- 205–216, New York, NY, USA, October 2001. ACM. ISBN 1-58113-415-0. doi: 10.1145/507635.507661. URL <http://dx.doi.org/10.1145/507635.507661>.
- [28] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM. ISBN 1-58113-986-1. doi: 10.1145/1073970.1073974. URL <http://dx.doi.org/10.1145/1073970.1073974>.
- [29] James Cheney and Ralf Hinze. First-Class Phantom Types. Technical Report TR2003-1901, Cornell University, July 2003. URL <http://hdl.handle.net/1813/5614>.
- [30] Tie Cheng and Xavier Rival. An Abstract Domain to Infer Types over Zones in Spreadsheets. In Antoine Miné and David Schmidt, editors, *Static Analysis*, volume 7460 of *Lecture Notes in Computer Science*, pages 94–110. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-33125-1_9. URL http://dx.doi.org/10.1007/978-3-642-33125-1_9.
- [31] Tie Cheng and Xavier Rival. Static Analysis of Spreadsheet Applications for Type-Unsafe Operations Detection. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 26–52. Springer Berlin Heidelberg, 2015. doi: 10.1007/978-3-662-46669-8_2. URL http://dx.doi.org/10.1007/978-3-662-46669-8_2.
- [32] Shing C. Cheung, Wanjun Chen, Yepang Liu, and Chang Xu. CUSTODES: Automatic Spreadsheet Cell Clustering and Smell Detection Using Strong and Weak Features. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 464–475, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884796. URL <http://dx.doi.org/10.1145/2884781.2884796>.
- [33] Wai-Mee Ching. Automatic Parallelization of APL-style Programs. In *Conference Proceedings on APL 90: For the Future, APL '90*, pages 76–80, New York, NY, USA, 1990. ACM. doi: 10.1145/97808.97826. URL <http://doi.acm.org/10.1145/97808.97826>.

- [34] Koen Claessen, Mary Sheeran, and Satnam Singh. *The Design and Verification of a Sorter Core*, volume 2144, chapter 28, pages 355–368. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-42541-0. doi: 10.1007/3-540-44798-9_28. URL http://dx.doi.org/10.1007/3-540-44798-9_28.
- [35] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, volume 42 of *ICFP '07*, pages 315–326, New York, NY, USA, October 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291199. URL <http://dx.doi.org/10.1145/1291151.1291199>.
- [36] Franklin C. Crow. Summed-area Tables for Texture Mapping. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 207–212, New York, NY, USA, 1984. ACM. ISBN 0-89791-138-5. doi: 10.1145/800031.808600. URL <http://dx.doi.org/10.1145/800031.808600>.
- [37] Walter A. C. A. J. de Hoon, Luc M. W. J. Rutten, and Marko C. J. D. van Eekelen. Implementing a Functional Spreadsheet in Clean. In *Journal of Functional Programming*, pages 383–414, 1995. URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.1465>.
- [38] Paul F. Dietz. Fully persistent arrays. In F. Dehne, J. R. Sack, and N. Santoro, editors, *Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74. Springer Berlin Heidelberg, 1989. doi: 10.1007/3-540-51542-9_8. URL http://dx.doi.org/10.1007/3-540-51542-9_8.
- [39] Wensheng Dou, Shing C. Cheung, and Jun Wei. Is Spreadsheet Ambiguity Harmful? Detecting and Repairing Spreadsheet Smells Due to Ambiguous Computation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 848–858, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568316. URL <http://dx.doi.org/10.1145/2568225.2568316>.
- [40] Wensheng Dou, Shing C. Cheung, Chushu Gao, Chang Xu, Liang Xu, and Jun Wei. Detecting Table Clones and Smells in Spreadsheets. In *Proceedings of the 2016 24th ACM SIGSOFT International*

- Symposium on Foundations of Software Engineering*, FSE 2016, pages 787–798, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950359. URL <http://dx.doi.org/10.1145/2950290.2950359>.
- [41] Wensheng Dou, Chang Xu, S. C. Cheung, and Jun Wei. CACheck: Detecting and Repairing Cell Arrays in Spreadsheets. *IEEE Transactions on Software Engineering*, PP(99):1, 2016. doi: 10.1109/TSE.2016.2584059. URL <http://dx.doi.org/10.1109/TSE.2016.2584059>.
- [42] Wensheng Dou, Liang Xu, Shing C. Cheung, Chushu Gao, Jun Wei, and Tao Huang. VEnron: A Versioned Spreadsheet Corpus and Related Evolution Analysis. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 162–171, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4205-6. doi: 10.1145/2889160.2889238. URL <http://dx.doi.org/10.1145/2889160.2889238>.
- [43] M. Felleisen. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, September 1992. ISSN 03043975. doi: 10.1016/0304-3975(92)90014-7. URL [http://dx.doi.org/10.1016/0304-3975\(92\)90014-7](http://dx.doi.org/10.1016/0304-3975(92)90014-7).
- [44] Matthias Felleisen, Robert B. Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, July 2009. ISBN 9780262259545. URL <http://www.worldcat.org/isbn/9780262259545>.
- [45] R. A. Finkel and J. L. Bentley. Quad Trees A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, March 1974. ISSN 0001-5903. doi: 10.1007/bf00288933. URL <http://dx.doi.org/10.1007/bf00288933>.
- [46] Marc Fisher and Gregg Rothermel. The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms. In *Proceedings of the First Workshop on End-user Software Engineering*, WEUSE I, pages 1–5, New York, NY, USA, 2005. ACM. ISBN 1-59593-131-7. doi: 10.1145/1082983.1083242. URL <http://dx.doi.org/10.1145/1082983.1083242>.

- [47] Matthew Fluet, Mike Rainey, and John Reppy. A Scheduling Framework for General-purpose Parallel Languages. *SIGPLAN Not.*, 43(9):241–252, September 2008. doi: 10.1145/1411203.1411239. URL <http://doi.acm.org/10.1145/1411203.1411239>.
- [48] The Document Foundation. LibreOffice Calc. Accessed on 24.05.2018. URL <https://www.libreoffice.org/discover/calc/>.
- [49] Clemens Grelck and Sven-Bodo Scholz. SAC: Off-the-shelf Support for Data-parallelism on Multicores. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP '07*, pages 25–33, New York, NY, USA, 2007. ACM. doi: 10.1145/1248648.1248654. URL <http://doi.acm.org/10.1145/1248648.1248654>.
- [50] Cordelia V. Hall. Using Hindley-Milner Type Inference to Optimise List Representation. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, LFP '94*, pages 162–172, New York, NY, USA, 1994. ACM. doi: 10.1145/182409.156781. URL <http://doi.acm.org/10.1145/182409.156781>.
- [51] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2nd edition, 2016. ISBN 1107150302, 9781107150300. URL <http://portal.acm.org/citation.cfm?id=3002812>.
- [52] Troels Henriksen and Cosmin E. Oancea. A T2 Graph-reduction Approach to Fusion. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC '13*, pages 47–58, New York, NY, USA, 2013. ACM. doi: 10.1145/2502323.2502328. URL <http://doi.acm.org/10.1145/2502323.2502328>.
- [53] Troels Henriksen, Niels G. W. Serup, Martin Elsmann, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. *SIGPLAN Not.*, 52(6):556–571, June 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062354. URL <http://dx.doi.org/10.1145/3140587.3062354>.
- [54] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Elsevier/Morgan Kaufmann, March 2008.

- ISBN 9780123705914. URL <http://www.worldcat.org/isbn/9780123705914>.
- [55] Felienne Hermans and Danny Dig. BumbleBee: A Refactoring Environment for Spreadsheet Formulas. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pages 747–750, New York, NY, USA, November 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2661673. URL <http://dx.doi.org/10.1145/2635868.2661673>.
- [56] Felienne Hermans and Emerson Murphy-Hill. Enron’s Spreadsheets and Related Emails: A Dataset and Analysis. In *Proceedings of the International Conference on Software Engineering, Software Engineering in Practice Track*, 2015. doi: 10.1109/ICSE.2015.129. URL <http://dx.doi.org/10.1109/ICSE.2015.129>.
- [57] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE ’11*, pages 451–460, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985855. URL <http://dx.doi.org/10.1145/1985793.1985855>.
- [58] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 441–451. IEEE, June 2012. ISBN 978-1-4673-1066-6. doi: 10.1109/icse.2012.6227171. URL <http://dx.doi.org/10.1109/icse.2012.6227171>.
- [59] Felienne Hermans, Ben Sedee, Martin Pinzger, and Arie van Deursen. Data Clone Detection and Visualization in Spreadsheets. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE ’13*, pages 292–301, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. doi: 10.1109/ICSE.2013.6606575. URL <http://dx.doi.org/10.1109/ICSE.2013.6606575>.
- [60] Ralf Hinze. An Algebra of Scans. In Dexter Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 186–210. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-27764-4_11. URL http://dx.doi.org/10.1007/978-3-540-27764-4_11.

- [61] Ralf Hinze and Clare Martin. Batcher's odd-even merging network revealed. *Journal of Functional Programming*, 28, June 2018. ISSN 0956-7968. doi: 10.1017/s0956796818000163. URL <http://dx.doi.org/10.1017/s0956796818000163>.
- [62] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(05):549–554, September 1997. ISSN 1469-7653. doi: 10.1017/s0956796897002864. URL <http://dx.doi.org/10.1017/s0956796897002864>.
- [63] Gwan-Hwan Hwang, Jenq K. Lee, and Dz-Ching Ju. An Array Operation Synthesis Scheme to Optimize Fortran 90 Programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 112–122, New York, NY, USA, 1995. ACM. doi: 10.1145/209936.209949. URL <http://doi.acm.org/10.1145/209936.209949>.
- [64] Tomás Isakowitz, Shimon Schocken, and Henry C. Lucas. Toward a Logical/Physical Theory of Spreadsheet Modeling. *ACM Trans. Inf. Syst.*, 13(1):1–37, January 1995. ISSN 1046-8188. doi: 10.1145/195705.195708. URL <http://dx.doi.org/10.1145/195705.195708>.
- [65] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, 1962. ISBN 0471430145. URL <http://www.worldcat.org/isbn/0471430145>.
- [66] Simon P. Jones, Alan Blackwell, and Margaret Burnett. A User-centred Approach to Functions in Excel. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 165–176, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: 10.1145/944705.944721. URL <http://dx.doi.org/10.1145/944705.944721>.
- [67] Haim Kaplan and Robert E. Tarjan. Persistent Lists with Catena-tion via Recursive Slow-down. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 93–102, New York, NY, USA, 1995. ACM. ISBN 0-89791-718-9. doi: 10.1145/225058.225090. URL <http://dx.doi.org/10.1145/225058.225090>.

- [68] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10*, pages 261–272, New York, NY, USA, September 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1932681.1863582. URL <http://dx.doi.org/10.1145/1932681.1863582>.
- [69] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. Vectorisation Avoidance. In *Proceedings of the 2012 Haskell Symposium, Haskell '12*, pages 37–48, New York, NY, USA, 2012. ACM. doi: 10.1145/2364506.2364512. URL <http://doi.acm.org/10.1145/2364506.2364512>.
- [70] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream Fusion, to Completeness. *SIGPLAN Not.*, 52(1):285–299, January 2017. ISSN 0362-1340. doi: 10.1145/3009837.3009880. URL <http://dx.doi.org/10.1145/3009837.3009880>.
- [71] L. Kotler, D. Abramson, P. Roe, and D. Mather. Activesheets: Super-computing with spreadsheets. In *Proceedings of the 2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference*, 2001.
- [72] Ananya Kumar, Guy E. Blelloch, and Robert Harper. Parallel Functional Arrays. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 706–718, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009869. URL <http://dx.doi.org/10.1145/3009837.3009869>.
- [73] John Launchbury and Tim Sheard. Warm Fusion: Deriving Build-catas from Recursive Definitions. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA '95*, pages 314–323, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. doi: 10.1145/224164.224223. URL <http://dx.doi.org/10.1145/224164.224223>.
- [74] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM*

- SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, volume 44 of *OOPSLA '09*, pages 227–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640106. URL <http://dx.doi.org/10.1145/1640089.1640106>.
- [75] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.06: Documentation and user’s manual. Technical Report 54, November 2017.
- [76] Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. Guiding Parallel Array Fusion with Indexed Types. In *Proceedings of the 2012 Haskell Symposium*, Haskell ’12, pages 25–36, New York, NY, USA, 2012. ACM. doi: 10.1145/2364506.2364511. URL <http://doi.acm.org/10.1145/2364506.2364511>.
- [77] Ben Lippmeier, Manuel M. T. Chakravarty, Gabriele Keller, and Amos Robinson. Data Flow Fusion with Series Expressions in Haskell. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell ’13, pages 93–104, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2383-3. doi: 10.1145/2503778.2503782. URL <http://dx.doi.org/10.1145/2503778.2503782>.
- [78] Björn Lisper and Johan Malmström. Haxcel: A Spreadsheet Interface to Haskell. In *14th Int. Workshop on the Implementation of Functional Languages*, pages 206–222, 2002. URL http://www.es.mdh.se/pdf_publications/367.pdf.
- [79] Michael Meeks. LibreOffice Calc: Spreadsheets on the GPU. 2014. URL <http://www.iwocl.org/iwocl-2014/abstracts/libreoffice-spreadsheets-on-the-gpu/>.
- [80] J. Mendes. ClassSheet-driven spreadsheet environments. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 235–236, September 2011. doi: 10.1109/VLHCC.2011.6070409. URL <http://dx.doi.org/10.1109/VLHCC.2011.6070409>.
- [81] Gary Miller. The Spreadsheet Paradigm: A Basis for Powerful and Accessible Programming. In *Companion Proceedings of the*

- 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH Companion 2015, pages 33–35, New York, NY, USA, 2015. ACM. doi: 10.1145/2814189.2814201. URL <http://doi.acm.org/10.1145/2814189.2814201>.
- [82] R. Mittermeir and M. Clermont. Finding high-level structures in spreadsheet programs. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 221–232, 2002. doi: 10.1109/WCRE.2002.1173080. URL <http://dx.doi.org/10.1109/WCRE.2002.1173080>.
- [83] Trenchard More. Axioms and Theorems for a Theory of Arrays. *IBM J. Res. Dev.*, 17(2):135–175, March 1973. ISSN 0018-8646. doi: 10.1147/rd.172.0135. URL <http://dx.doi.org/10.1147/rd.172.0135>.
- [84] J. T. O'Donnell. MPP implementation of abstract data parallel architectures for declarative programming languages. In *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of*, pages 629–636, October 1988. doi: 10.1109/FMPC.1988.47507. URL <http://dx.doi.org/10.1109/FMPC.1988.47507>.
- [85] Atsushi Ohori and Isao Sasano. Lightweight Fusion by Fixed Point Promotion. *SIGPLAN Not.*, 42(1):143–154, January 2007. ISSN 0362-1340. doi: 10.1145/1190216.1190241. URL <http://dx.doi.org/10.1145/1190216.1190241>.
- [86] Oracle. Class LongAdder. Accessed on 24.05.2018. URL <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html>.
- [87] Karl Pearson. Note on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London*, 58:240–242, 1895. URL <http://www.jstor.org/stable/115794>.
- [88] R. H. Perrott. A Language for Array and Vector Processors. *ACM Trans. Program. Lang. Syst.*, 1(2):177–195, October 1979. doi: 10.1145/357073.357075. URL <http://doi.acm.org/10.1145/357073.357075>.
- [89] Jorma Sajaniemi. Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization. *Journal of Visual Languages & Computing*, 11(1):49–82, February 2000. ISSN 1045926X. doi:

- 10.1006/jvlc.1999.0142. URL <http://dx.doi.org/10.1006/jvlc.1999.0142>.
- [90] A. V. S. Sastry and William Clinger. Parallel Destructive Updating in Strict Functional Languages. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, LFP '94*, pages 263–272, New York, NY, USA, 1994. ACM. doi: 10.1145/182409.182486. URL <http://doi.acm.org/10.1145/182409.182486>.
- [91] Christopher Scaffidi. Counts and Earnings of End-User Developers, September 2017. URL <http://web.engr.oregonstate.edu/~scaffidi/papers/eud-counts>.
- [92] Peter Sestoft. Microbenchmarks in Java and C#. Lecture Notes, September 2013. URL <https://www.itu.dk/people/sestoft/papers/benchmarking.pdf>.
- [93] Peter Sestoft. *Spreadsheet Implementation Technology: Basics and Extensions*. The MIT Press, September 2014. ISBN 0262526646. URL <http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6940404>.
- [94] Peter Sestoft and Henrik I. Hansen. *C# Precisely*. The MIT Press, second edition, November 2011. ISBN 0262516861, 9780262516860. URL <http://portal.acm.org/citation.cfm?id=2159552>.
- [95] Mary Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *Journal of Functional Programming*, 21:59–114, January 2011. ISSN 1469-7653. doi: 10.1017/s0956796810000304. URL <http://www.cse.chalmers.se/~ms/>.
- [96] Rishabh Singh and Sumit Gulwani. Transforming Spreadsheet Data Types Using Examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, pages 343–356, New York, NY, USA, 2016. ACM. doi: 10.1145/2837614.2837668. URL <http://doi.acm.org/10.1145/2837614.2837668>.
- [97] J. M. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991. ISSN 00189219. doi: 10.1109/5.92044. URL <http://dx.doi.org/10.1109/5.92044>.

- [98] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981. ISSN 00222836. doi: 10.1016/0022-2836(81)90087-5. URL [http://dx.doi.org/10.1016/0022-2836\(81\)90087-5](http://dx.doi.org/10.1016/0022-2836(81)90087-5).
- [99] Marc Stadelmann. A Spreadsheet Based on Constraints. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*, UIST '93, pages 217–224, New York, NY, USA, 1993. ACM. doi: 10.1145/168642.168664. URL <http://doi.acm.org/10.1145/168642.168664>.
- [100] Guy L. Steele. Parallel Programming and Parallel Abstractions in Fortress. In *Proceedings of the 8th International Conference on Functional and Logic Programming*, FLOPS'06, page 1, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33438-6, 978-3-540-33438-5. doi: 10.1007/11737414_1. URL http://dx.doi.org/10.1007/11737414_1.
- [101] Guy L. Steele. Organizing Functional Code for Parallel Execution or, Foldl and Foldr Considered Slightly Harmful. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, volume 44 of ICFP '09, pages 1–2, New York, NY, USA, August 2009. ACM. doi: 10.1145/1631687.1596551. URL <http://dx.doi.org/10.1145/1631687.1596551>.
- [102] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. RRB Vector: A Practical General Purpose Immutable Sequence. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 342–354, New York, NY, USA, 2015. ACM. doi: 10.1145/2784731.2784739. URL <http://doi.acm.org/10.1145/2784731.2784739>.
- [103] Alaaeddin Swidan, Felienne Hermans, and Ruben Koesoemowidjojo. Improving the Performance of a Large Scale Spreadsheet: A Case Study. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 673–677. IEEE, March 2016. ISBN 978-1-5090-1855-0. doi: 10.1109/saner.2016.100. URL <http://dx.doi.org/10.1109/saner.2016.100>.
- [104] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# 4.0*. Apress, Berkely, CA, USA, 4th edition, 2015. ISBN 1484207416,

9781484207413. URL <http://portal.acm.org/citation.cfm?id=2888485>.
- [105] Jim Trudeau. Collaboration and Open Source at AMD: LibreOffice, July 2015. URL <http://developer.amd.com/community/blog/2015/07/15/collaboration-and-open-source-at-amd-libreoffice/>. Accessed on 31.07.2015.
- [106] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy Binary-splitting: A Run-time Adaptive Work-stealing Scheduler. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 45 of *PPoPP '10*, pages 179–190, New York, NY, USA, January 2010. ACM. doi: 10.1145/1837853.1693479. URL <http://dx.doi.org/10.1145/1837853.1693479>.
- [107] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism. *ACM Trans. Program. Lang. Syst.*, 36(3), September 2014. ISSN 0164-0925. doi: 10.1145/2629643. URL <http://dx.doi.org/10.1145/2629643>.
- [108] J. G. van der Corput. Verteilungsfunktionen. I. Mitt. *Proc. Akad. Wet. Amsterdam*, 38:813–821, 1935.
- [109] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream Processing with a Spreadsheet. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 360–384. Springer Berlin Heidelberg, 2014. doi: 10.1007/978-3-662-44202-9_15. URL http://dx.doi.org/10.1007/978-3-662-44202-9_15.
- [110] Andrew P. Wack. *Partitioning Dependency Graphs for Concurrent Execution: A Parallel Spreadsheet on a Realistically Modeled Message Passing Environment*. PhD thesis, University of Delaware, Newark, DE, USA, 1996. URL <http://portal.acm.org/citation.cfm?id=269551>.
- [111] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *Proceedings of the 2Nd European Symposium on Programming, ESOP '88*, pages 344–358, London, UK, UK, 1988. Springer-Verlag.

- ISBN 3-540-19027-9. doi: 10.1016/0304-3975(90)90147-A. URL [http://dx.doi.org/10.1016/0304-3975\(90\)90147-A](http://dx.doi.org/10.1016/0304-3975(90)90147-A).
- [112] Alan Yoder and David L. Cohn. Observations on Spreadsheet Languages, Intension and Dataflow. Technical Report TR-94-22, University of Notre Dame, 1994. URL http://www.cse.unsw.edu.au/~plaice/archive/ISLIP/1994/islip1994_060.pdf.
- [113] Alan G. Yoder and David L. Cohn. Domain-Specific and General-Purpose Aspects of Spreadsheet Language. In Sam Kamin, editor, *DSL'97 – First ACM SIGPLAN Workshop on Domain-Specific Languages*, pages 37–47, 1997. URL <http://homepages.ecs.vuw.ac.nz/~elvis/db/references/yoder.ps>.

Appendix A

Funcalc Source Code

The techniques presented in Chapters 4, 5 and 6 have been implemented in Funcalc. The code for these implementations is far too large to be included in print. However, the source code is stored on GitHub and is accessible upon request. At the time of writing, the Funcalc repository resides at

<https://github.com/popular-parallel-programming/funcalc>.

Funcalc runs on .NET for Windows. It is possible to compile using Mono, but currently, Funcalc will crash due to library incompatibilities. There is a Windows build script `build.bat` that can be run with different compilation flags to combine various techniques. Compile with `-r` to build in “release” mode.

For instance, to compile Funcalc with parallel quad ropes and sequential minimal recalculation, run the following command on the command line:

```
> build.bat -r -n -p -q
```

After compiling, use the following command to benchmark the minimal recalculation for five repetitions on eight cores of the spreadsheet `my-sheet.xml`:

```
> funcalc.bat -r minimal 5 8 my-sheet.xml
```

The source code is organized as follows:

Chapter 4 Branch fbie/cell-array-transformation.

- n disables parallel minimal recalculation;
- w enables cell array rewriting at load-time; and
- p enables data-parallel operations on arrays.

Chapter 5 Branch parallel-stable.

- n disables parallel minimal recalculation; and
- l enables thread-local evaluation.

Chapter 6 Branch fbie/quad-ropes-par.

- n disables parallel minimal recalculation;
- q enables quad ropes; and
- p enables data-parallel operations on quad ropes.

Appendix B

PLT Redex Model of Cell Array Rewriting Semantics

The PLT Redex model for λ -calc and the rewriting semantics (Chapter 4) is available at:

<https://github.com/popular-parallel-programming/funcalc-redex>

```
#lang racket
(require redex)

(define-language mini-calc
  (n ::= real)
  (i ::= integer)
  (t ::= string)
  (v ::= n t (err string))
  (ca ::=
    (rc i i) ; absolute
    (rc [i] i) ; row-relative
    (rc i [i]) ; column-relative
    (rc [i] [i])) ; relative
  ;; Formula expressions.
  (e ::=
    v
    ca
    (ca : ca)
    (e + e)
    (e = e))
```

```

      (IF e e e)
      (SUM e ...))
;; A sheet whose cells reduce from expressions to values.
(s ::= ( $\sigma$  (ca := e) ...)))

(define-extended-language mini-calc-S mini-calc
  (E ::=
    hole
    (E + e)
    (v + E)
    (E = e)
    (v = E)
    (IF E e e)
    (SUM v ... E e ...))
  (S ::= ( $\sigma$  (ca_v := v) ... (ca := E) (ca_e := e) ...)))

(define-metafunction mini-calc
  lookup : ca ca -> ca
  [(lookup (rc [i_1] i_2) (rc i_3 _))
   (rc ,(+ (term i_1) (term i_3)) i_2)]
  [(lookup (rc i_1 [i_2]) (rc _ i_4))
   (rc i_1 ,(+ (term i_2) (term i_4)))]
  [(lookup (rc [i_1] [i_2]) (rc i_3 i_4))
   (rc ,(+ (term i_1) (term i_3)) ,(+ (term i_2) (term i_4)))]
  [(lookup ca _) ca])

(define-extended-language  $\lambda$ -calc mini-calc
  (e ::=
    ....
    x
    (ca : ca)
    (e e ...)
    (MAP f e ...)
    (PREFIX f e ...)
    (TABULATE f e e)
    (HREP e e)
    (VREP e e)
    (SLICE e e e e e)
    (INDEX e e e))
  (f ::= ( $\lambda$  (x ...) e))
  (v ::= .... [[v ... ] ...] f)
  (x ::= variable-not-otherwise-mentioned))

```

```

#:binding-forms
( $\lambda$  (x ...) e #:refers-to (shadow x ...)))

(define-union-language  $\lambda$ -calc-S0  $\lambda$ -calc mini-calc-S)
(define-extended-language  $\lambda$ -calc-S  $\lambda$ -calc-S0
  (E ::=
    ....
    (f v ... E e ...)
    (MAP f v ... E e ...)
    (PREFIX f v ... E e ...)
    (TABULATE f E e)
    (TABULATE f v E)
    (HREP v ... E e ...)
    (VREP v ... E e ...)
    (SLICE v .. E e ...)
    (INDEX v ... E e ...)))

(define-extended-language  $\lambda$ -calc-L  $\lambda$ -calc
  (l ::=
    v
    x ; Variable names replace relative cell references.
    (l + l)
    (l = l)
    (IF l l l)
    (rc i i) ; Only absolute cell references.
    ((rc i i) : (rc i i)) ; Only absolute cell ranges.
    (MAP f l ...)
    (HREP l l)
    (VREP l l)
    (PREFIX f l ...)
    (SUM l ...)
    (SLICE l l l l l) ; SLICE(arr, r1, c1, r2, c2)
    (TABULATE f l l))
  (L ::=
    hole
    (L + e)
    (l + L)
    (L = e)
    (l = L)
    (IF L e e)
    (IF l L e)
    (IF l l L)

```

```

    (SUM 1 ... L e ...))
(c ::=
  ; A lifting in progress.
  (more ((ca x) ...) ; Transitive substitutions
        ((ca x) ...) ; Intransitive substitutions
        (x x)
        ((ca : ca) := e))
  ; Lifted result
  (done ((ca : ca) := 1))))

(define-metafunction  $\lambda$ -calc-L
  extd : e e  $\rightarrow$  e
  [(extd e1 e2)
   (HREP e1 (COLUMNS e2))
   (side-condition (eq? 1 (term (COLUMNS e1))))]
  [(extd e1 e2)
   (VREP e1 (ROWS e2))
   (side-condition (eq? 1 (term (ROWS e1))))]
  [(extd e1 _) e1])

(define (intersect?/racket xs ys)
  (ormap ( $\lambda$  (x) (member x ys)) xs))

(define-metafunction  $\lambda$ -calc-L
   $\omega$  : (rc [i] [i])  $\rightarrow$  i
  [( $\omega$  (rc [0] [_])) 1]
  [( $\omega$  (rc [_] [0])) 3]
  [( $\omega$  (rc [_] [_])) 2])

(define (sort-trans/racket xs)
  (sort xs ( $\lambda$  (x y) (< (term ( $\omega$  ,x)) (term ( $\omega$  ,y))))
        #:key first))

(define-metafunction  $\lambda$ -calc-L
  stride : (rc [i] [i])  $\rightarrow$  i
  [(stride (rc [ir] [ic]))
   ,(max (abs (term ir)) (abs (term ic))))]

(define-metafunction  $\lambda$ -calc-L
  isAbs : ca  $\rightarrow$  boolean
  [(isAbs (r i c i)) #t]
  [(isAbs _) #f])

```

```

(define-metafunction  $\lambda$ -calc-L
  sort&fill : ((ca x) ...) -> ((ca x) ...)
  [(sort&fill ((ca x) ...))
   ,(sort-trans/racket (term ((ca x) ...)))])

(define-metafunction  $\lambda$ -calc-L
  row : (rc i i) -> i
  [(row (rc i _)) i])

(define-metafunction  $\lambda$ -calc-L
  column : (rc i i) -> i
  [(column (rc _ i)) i])

(define lift
  (reduction-relation  $\lambda$ -calc-L
    #:domain c
    #:arrow ~>
    ; subst-intrans- $\exists$ : An intransitive substitution exists already.
    (~> (more ((ca_1 x_1) ...) ; Transitive
              ((ca_2 x_2) ... (ca x) (ca_3 x_4) ...) ; Intransitive
              (x_c x_r)
              ((ca_ul : ca_lr) := (in-hole L ca))))

      (more ((ca_1 x_1) ...) ; Transitive
              ((ca_2 x_2) ... (ca x) (ca_3 x_4) ...) ; Intransitive
              (x_c x_r)
              ((ca_ul : ca_lr) := (in-hole L x))))
    exist-i)

    ; subst-trans- $\exists$ : A transitive substitution exists already.
    (~> (more ((ca_1 x_1) ... (ca x) (ca_2 x_2) ...) ; Transitive
              ((ca_3 x_4) ...) ; Intransitive
              (x_c x_r)
              ((ca_ul : ca_lr) := (in-hole L ca))))

      (more ((ca_1 x_1) ... (ca x) (ca_2 x_2) ...) ; Transitive
              ((ca_3 x_4) ...) ; Intransitive
              (x_c x_r)
              ((ca_ul : ca_lr) := (in-hole L x))))
    exist-t)
  )

```

; subst-intrans: The reference is intransitive and there does not
; already exist a substitution.

```
(~> (more ((ca_1 x_1) ...) ; Transitive
      ((ca_2 x_2) ...) ; Intransitive
      (x_c x_r)
      ((ca_ul : ca_lr) := (in-hole L ca))) ; Lifting
```

```
(more ((ca_1 x_1) ...) ; Transitive
      ((ca_2 x_2) ... (ca x)) ; Intransitive
      (x_c x_r)
      ((ca_ul : ca_lr) := (in-hole L x)))
```

```
(fresh x)
(where (ca_r ...) (enumerate (ca_ul : ca_lr)))
(side-condition (not (term (isAbs ca))))
(side-condition (not (intersect?/racket
                      (term ((lookup ca ca_r) ...))
                      (term (ca_r ...)))))
(side-condition (not (member (term ca) (term (ca_2 ...)))))
subst-i)
```

; subst-trans: The reference is transitive and there does not
; already exist a substitution.

```
(~> (more ((ca_1 x_1) ...) ; Transitive
      ((ca_2 x_2) ...) ; Intransitive
      (x_c x_r)
      ((ca_ul : ca_lr) := (in-hole L ca)))
(more ((ca_1 x_1) ... (ca x)) ; Transitive
      ((ca_2 x_2) ...) ; Intransitive
      (x_c x_r)
      ((ca_ul : ca_lr) := (in-hole L x)))
```

```
(fresh x)
(where (ca_r ...) (enumerate (ca_ul : ca_lr)))
(side-condition (not (term (isAbs ca))))
(side-condition (intersect?/racket
                      (term ((lookup ca ca_r) ...))
                      (term (ca_r ...)))))
(side-condition (not (member (term ca) (term (ca_1 ...)))))
(side-condition (= 1 (term (stride ca))))
subst-t)
```

; subst-area: Substitute an area by a call to SLICE.


```

(~> (more ((ca_t x_t) ...) ; Transitive
      ((ca_i x_i) ...) ; Intransitive
      (x_c x_r)
      ((ca_ul : ca_lr) := (in-hole L (ca_1 : ca_2))))
(more ((ca_t x_t) ...) ; Transitive
      ((ca_i x_i) ...) ; Intransitive
      (x_c x_r)
      ((ca_ul : ca_lr)
      :=
      (in-hole
      L
      (SLICE (ca_ul1 : ca_lr2)
              x_r
              x_c
              (ROWS
               ((lookup ca_1 ca_ul) : (lookup ca_2 ca_ul)))
              (COLUMNS
               ((lookup ca_1 ca_ul) : (lookup ca_2 ca_ul)))))))
(where ca_ul1 (lookup ca_1 ca_ul)) ; Upper-left of area.
(where ca_lr2 (lookup ca_2 ca_lr)) ; Lower-right of area.
(side-condition (not (term (isAbs ca_1))))
(side-condition (not (term (isAbs ca_2))))
subst-area)

```

; synth-map: The expression has been lifted to a λ -body and there
; are no transitive references.

```

(~> (more () ; Transitive
      ((ca_i x_i) ...) ; Intransitive
      (x_c x_r)
      ((ca_ul : ca_lr) := 1))

      (done ((ca_ul : ca_lr) := (MAP ( $\lambda$  (x_c x_r x_i ...) 1)
                                     (extd (ca_ul0 : ca_lr0)
                                     (ca_ul : ca_lr)) ...)))
      (where (ca_ul0 ...) ((lookup ca_i ca_ul) ...))
      (where (ca_lr0 ...) ((lookup ca_i ca_lr) ...))
      (side-condition (not (empty? (term (ca_i ...)))))
      synth-map)

```

; synth-prefix: The expression has been lifted to a λ -body and
; there are transitive references.

```

(~> (more ((ca_t x_t) ...) ; Transitive

```

```

      ((ca_i x_i) ...) ; Intransitive
      (x_c x_r)
      ((ca_ul : ca_lr) := 1))
(done ((ca_ul : ca_lr)
      :=
      (PREFIX (λ (x_c x_r x_t1 x_t2 x_t3 x_i ...) 1)
              (ca_c0 : ca_c1)
              ca_s
              (ca_r0 : ca_r1)
              (extd (ca_ul0 : ca_lr0) (ca_ul : ca_lr)) ...)))
(where (ca_ul0 ...) ((lookup ca_i ca_ul) ...))
(where (ca_lr0 ...) ((lookup ca_i ca_lr) ...))
(where ((ca_t1 x_t1) (ca_t2 x_t2) (ca_t3 x_t3))
      (sort&fill ((ca_t x_t) ...)))
; Construct initial row and column address
(where ca_s (lookup ca_t2 ca_ul))
(where ca_c0 (lookup ca_t1 ca_ul))
(where ca_r0 (lookup ca_t3 ca_ul))
(where ca_c1 (rc (row ca_lr) (column ca_c0)))
(where ca_r1 (rc (row ca_r0) (column ca_lr)))
(side-condition (not (empty? (term (ca_t ...)))))
synth-prefix)

;; synth-tabulate: Generate an array even if there are no
;; input-arrays per-se.
(~> (more ()
      ()
      (x_c x_r)
      ((ca_ul : ca_lr) := 1))

      (done ((ca_ul : ca_lr)
            :=
            (TABULATE (λ (x_c x_r) 1)
                      (ROWS (ca_ul : ca_lr))
                      (COLUMNS (ca_ul : ca_lr)))))
      synth-tabulate)))

```

Appendix C

Quad Rope Reference Implementation

The F# code for the quad rope reference implementation (Chapter 6) is available at:

<https://github.com/popular-parallel-programming/quad-ropes>

Compile the reference implementation by running

```
> build.bat --paket  
> build.bat -r
```

on the command line and then execute the benchmarks via:

```
> scripts\benchmark-all.bat
```


Appendix D

Source Code for Quad Rope Fusion

The OCaml code for laziness and fusion of operations on quad ropes (Chapter 7) is available at:

<https://github.com/popular-parallel-programming/funky-quad-ropes>

```
module Shim =
  struct
    let (◦) f g =
      fun x → f (g x)

    let ($) f x = f x
  end

(* Signature for two-dimensional collections. *)
module type Collection2D =
  sig
    type 'a t
    val init : int → int → (int → int → 'a) → 'a t
    val get : 'a t → int → int → 'a
    val rows : _ t → int
    val cols : _ t → int
    val hcat : 'a t → 'a t → 'a t
    val vcat : 'a t → 'a t → 'a t
    val map : ('a → 'b) → 'a t → 'b t
    val zipWith : ('a → 'b → 'c) → 'a t → 'b t → 'c t
    val map_reduce : ('a → 'b) → ('b → 'b → 'b) → 'b
      → 'a t → 'b
```

```

val reduce : ('a → 'a → 'a) → 'a → 'a t → 'a
val replicate : int → int → 'a → 'a t
val slice : 'a t → int → int → int → int → 'a t
val transpose : 'a t → 'a t
end

```

```

module Array2D =
  struct
    type 'a t = 'a array array

    open Shim

    let init rows cols f : _ =
      Array.init rows $ fun i → Array.init cols (f i)

    let get (xss : _ t) i j =
      Array.get (Array.get xss i) j

    let rows (xss : _ t) =
      Array.length xss

    let cols (xss : _ t) = (* Assume all columns are of
      equal length. *)
      if rows xss = 0 then 0 else Array.length $ Array.get
        xss 0

    let hcat xss yss =
      if rows xss = rows yss then
        init (rows xss)
          (cols xss + cols yss)
          (fun r c →
            if c < cols xss then
              get xss r c
            else
              get yss r (c - cols xss))
      else
        failwith "not same number of rows"

    let vcat xss yss =
      if cols xss = cols yss then
        init (rows xss + rows yss)

```

```

        (cols xss)
        (fun r c →
            if r < rows xss then
                get xss r c
            else
                get yss (r - rows xss) c)
    else
        failwith "not same number of columns"

let map f xss : _ t =
    Array.map (fun xs → Array.map f xs) xss

let mapi f xss : _ t =
    Array.mapi (fun i xs → Array.mapi (fun j x → f i j
        x) xs) xss

let zipWith f (xss0 : _ t) (xss1 : _ t) : _ t =
    mapi (fun i j x → f x (get xss1 i j)) xss0

let zipWithi f (xss0 : _ t) (xss1 : _ t) : _ t =
    mapi (fun i j x → f i j x (get xss1 i j)) xss0

let mapi_reduce (f : int → int → 'a → 'b) (g : 'b
    → 'b → 'b) (e : 'b) (xss : 'a t) : 'b =
    let acc = ref e in
    for r = 0 to rows xss - 1 do
        for c = 0 to cols xss - 1 do
            acc := g !acc (f r c (get xss r c));
        done;
    done;
    !acc

let map_reduce f =
    mapi_reduce (fun _ _ x → f x)

let reduce f =
    mapi_reduce (fun _ _ x → x) f

let replicate r c x =
    init r c (fun _ _ → x)

let slice xs r c h w =

```

```

    init h w (fun r' c' → get xs (r + r') (c + c'))

    let transpose xs =
      init (cols xs) (rows xs) (fun c r → get xs r c)
    end

module QuadRope =
  struct

    open Shim

    type _ quad_rope =
      | Funk : (int → int → 'a → 'b) * 'a quad_rope * '
        b quad_rope lazy_t → 'b quad_rope
      | Leaf : 'a Array2D.t → 'a quad_rope
      | HCat : 'a quad_rope * 'a quad_rope → 'a quad_rope
      | VCat : 'a quad_rope * 'a quad_rope → 'a quad_rope
      | Sparse : int * int * 'a → 'a quad_rope

    type 'a t = 'a quad_rope

    let rec rows : 'a . 'a quad_rope → int = function
      | Funk (_, q, _) → rows q
      | Leaf xss → Array2D.rows xss
      | HCat (q1, q2) → rows q1 (* rows q1 = rows q2 *)
      | VCat (q1, q2) → rows q1 + rows q2
      | Sparse (r, _, _) → r

    let rec cols : 'a . 'a quad_rope → int = function
      | Funk (_, q, _) → cols q
      | Leaf xss → Array2D.cols xss
      | HCat (q1, q2) → cols q1 + cols q2
      | VCat (q1, q2) → cols q1 (* cols q1 = cols q2 *)
      | Sparse (_, c, _) → c

    let max_size = 32

    let hcat q1 q2 =
      if rows q1 = rows q2 then
        match q1 with
        | Leaf l1 →
          (match q2 with

```



```

    | Leaf l2 when cols q1 + cols q2 <= max_size
      →
        Leaf (Array2D.hcat l1 l2)
    | HCat ((Leaf l2 as q21), q22) when cols q1 +
      cols q21 <= max_size →
        HCat (Leaf (Array2D.hcat l1 l2), q22)
    | _ → HCat (q1, q2))
  | HCat (q11, (Leaf l12 as q12)) →
    (match q2 with
    | Leaf l2 when cols q12 + cols q2 <= max_size
      →
        HCat (q11, Leaf (Array2D.hcat l12 l2))
    | _ → HCat (q1, q2))
  | _ → HCat (q1, q2)
else
  failwith "not same number of rows"

let vcat q1 q2 =
  if cols q1 = cols q2 then
    match q1, q2 with
    | Leaf l1, Leaf l2 when rows q1 + rows q2 <=
      max_size →
        Leaf (Array2D.vcat l1 l2)
    | _ → VCat (q1, q2)
  else
    failwith "not same number of columns"

let rec get q r c =
  match q with
  | Funk (_, _, thunk) → get (Lazy.force thunk) r c
  | Leaf xss           → Array2D.get xss r c
  | HCat (q1, q2)      → if c < cols q1 then get q1 r
    c else get q2 r (c - cols q1)
  | VCat (q1, q2)      → if r < rows q1 then get q1 r
    c else get q2 (r - rows q1) c
  | Sparse (_, _, x)   → x

let init rows cols f =
  let rec init row_off col_off rows cols =
    if rows < max_size && cols < max_size then
      Leaf (Array2D.init rows cols (fun r c → f (r +
        row_off) (c + col_off)))

```

```

else if rows < cols then
  let c2 = cols / 2 in
  hcat (init row_off col_off rows c2) (init
    row_off (col_off + c2) rows (cols - c2))
else
  let r2 = rows / 2 in
  vcat (init row_off col_off r2 cols) (init (
    row_off + r2) col_off (rows - r2) cols)
in init 0 0 rows cols

let mapi f =
  let rec mapi_i : 'a 'b . int → int → (int → int
    → 'a → 'b) → 'a quad_rope → 'b quad_rope =
  fun r0 c0 f → (function
    | Funk (g, q, thunk) →
      if Lazy.is_val thunk then
        mapi_i r0 c0 f $ Lazy.force thunk
      else
        let k = fun r c x → f r c (g r c x) in
        mapi_i r0 c0 k q
    | Leaf xss →
      Leaf (Array2D.mapi (fun r c x → f (r0 + r)
        (c0 + c) x) xss)
    | HCat (q1, q2) →
      HCat (mapi_i r0 c0 f q1, mapi_i r0 (c0 +
        cols q1) f q2)
    | VCat (q1, q2) →
      VCat (mapi_i r0 c0 f q1, mapi_i (r0 + rows
        q1) c0 f q2)
    | Sparse (r, c, x) →
      init r c (fun r c → f (r0 + r) (c0 + c) x))
  in mapi_i 0 0 f

let map f =
  mapi (fun _ _ x → f x)

let zipWithi f q1 q2 =
  if rows q1 <> rows q2 || cols q1 <> cols q2 then
    failwith "shape mismatch"
  else
    (* Cheap and slow; materializes Funk nodes. *)

```

```

init (rows q1) (cols q1) (fun r c → f r c (get q1
    r c) (get q2 r c))

let zipWith f =
  zipWithi (fun _ _ x y → f x y)

let replicate rows cols x =
  Sparse (max 0 rows, max 0 cols, x)

let mapi_reduce f g e q =
  let sparse_loop rows cols x r0 c0 f g e =
    let acc = ref e in
    for r = r0 to r0 + rows - 1 do
      for c = c0 to c0 + cols - 1 do
        acc := g !acc (f r c x);
      done;
    done;
    !acc
  in
  let rec mapi_reduce_i :
    'a 'b . int → int → (int → int → 'a → 'b) →
    ('b → 'b → 'b) → 'b → 'a quad_rope → 'b =
    fun r0 c0 f g e →
    (function
      | Funk (k, q, thunk) →
        if Lazy.is_val thunk then
          mapi_reduce_i r0 c0 f g e $ Lazy.force thunk
        else
          let h = fun r c x → f r c (k r c x) in
          mapi_reduce_i r0 c0 h g e q
      | Leaf xss →
        Array2D.mapi_reduce (fun r c x → f (r0 + r) (
          c0 + c) x) g e xss
      | HCat (q1, q2) →
        g (mapi_reduce_i r0 c0 f g e q1) (
          mapi_reduce_i r0 (c0 + cols q1) f g e q2)
      | VCat (q1, q2) →
        g (mapi_reduce_i r0 c0 f g e q1) (
          mapi_reduce_i (r0 + rows q1) c0 f g e q2)
      | Sparse (r, c, x) →
        sparse_loop r c x r0 c0 f g e) (* TODO: Add
        recursive splitting. *)

```

```

    in mapi_reduce_i 0 0 f g e q

let map_reduce f =
  mapi_reduce (fun _ _ x → f x)

let reduce f =
  map_reduce (fun x → x) f

let rec slice q i j h w =
  init h w (fun r c → get q (i + r) (j + c))

let rec transpose = function
  | Leaf xs → Leaf (Array2D.transpose xs)
  | HCat (q1, q2) → VCat (transpose q1, transpose q2)
  | VCat (q1, q2) → HCat (transpose q1, transpose q2)
  | Sparse (r, c, x) → Sparse (c, r, x)
  | Funk (_, _, t) → transpose $ Lazy.force t
end

module Funky =
  struct
    type 'a t = 'a QuadRope.t

    open Shim
    open QuadRope

    let s_max = QuadRope.max_size
    let rows = QuadRope.rows
    let cols = QuadRope.cols
    let get = QuadRope.get
    let hcat = QuadRope.hcat
    let vcat = QuadRope.vcat

    let is_leaf_size rows cols =
      rows <= QuadRope.max_size && cols <= QuadRope.
        max_size

    let rec mapi : 'a 'b . (int → int → 'a → 'b) → 'a
      quad_rope → 'b quad_rope =
      fun f → (function
        | q when is_leaf_size (rows q) (cols q) →
          QuadRope.mapi f q

```

```

    | Funk (g, q, thunk) →
      if Lazy.is_val thunk then
        mapi f $ Lazy.force thunk
      else
        mapi (fun r c x → f r c (g r c x)) q
    | q → Funk (f, q, lazy (QuadRope.mapi f q))

let map f =
  mapi (fun r c x → f x)

let init rows cols f =
  if is_leaf_size rows cols then
    QuadRope.init rows cols f
  else
    let p = replicate rows cols () in
    let g = fun r c _ → f r c in
    mapi g p

let zipWithi f q1 q2 =
  let g =
    (match q1 with
     | Funk (f1, p1, t1) when not (Lazy.is_val t1) →
       fun r c → f1 r c (get p1 r c)
     | _ → get q1)
  in
  mapi (fun r c x → f r c (g r c) x) q2

let zipWith f =
  zipWithi (fun _ _ x y → f x y)

(* Reduction never results in a new Funk at the
 * same hierarchy level, so we use the strict
 * implementations. *)
let mapi_reduce = QuadRope.mapi_reduce
let map_reduce  = QuadRope.map_reduce
let reduce      = QuadRope.reduce
let replicate   = QuadRope.replicate

let slice q i j h w =
  match q with
  | Funk (f, p, t) when not (Lazy.is_val t) →

```

```

    mapi (fun r c x → f (r + i) (c + j) x) (QuadRope
        .slice p i j h w)
  | _ → QuadRope.slice q i j h w

let transpose = function
  | Funk (f, p, t) when not (Lazy.is_val t) →
    mapi (fun c r x → f r c x) (transpose p)
  | q → QuadRope.transpose q
end

module Test(M : Collection2D) =
  struct
    open Shim

    let sum =
      M.reduce ( +. ) 0.

    let test_f f rows cols =
      let xs = M.init rows cols (fun _ _ → Random.float
        1000. +. 1.) in
      let ys = M.init rows cols (fun _ _ → Random.float
        1000. +. 1.) in
      f xs ys

    let test_pearsons =
      let pearsons xs ys =
        let size = fun xs → M.rows xs * M.cols xs in
        let mean = fun xs → M.reduce ( +. ) 0. xs /.
          float (size xs) in
        let x_mean = mean xs in
        let y_mean = mean ys in
        let x_err = M.map (fun x → x -. x_mean) xs in
        let y_err = M.map (fun y → y -. y_mean) ys in
        let x_sqerr = M.map (fun x → x -. x_mean ** 2.)
          xs in
        let y_sqerr = M.map (fun y → y -. y_mean ** 2.)
          ys in
        (sum (M.zipWith ( *. ) x_err y_err)) /. (sqrt (sum
          x_sqerr) *. sqrt (sum y_sqerr))
      in
      test_f pearsons

```

```

let test_vdc n =
  let singleton =
    fun x → M.init 1 1 (fun _ _ → x) in
  let next =
    fun i is → M.hcat is (M.hcat (singleton i) (M.map
      ((+. ) i) is)) in
  let rec vdc n =
    if n <= 1. then
      singleton 0.5
    else
      let n' = 2. ** -.n in
      next n' (vdc (n -. 1.))
  in
  sum (vdc (float n))

let test_primes n =
  let rec sieve p ns =
    if n <= 2 then
      M.replicate 0 0 (false, 0)
    else if p = n then
      ns
    else
      sieve (p + 1) (M.map (fun (f, m) → f || (m <> p
        && m mod p = 0), m) ns)
  in
  let primes = sieve 2 $ M.init 1 (n - 2) (fun _ m →
    false, m + 2) in
  M.map_reduce (fun (f, m) → if f then 0 else 1) ( +
    ) 0 primes
end

```