Quad Ropes: Immutable, Declarative Arrays with Parallelizable Operations

Florian Biermann * Peter Sestoft

IT University of Copenhagen
Computer Science Department
Rued Langgaards Vej 7, 2300 Copenhagen, Denmark
{fbie, sestoft}@itu.dk

Abstract

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, one should use the same, somewhat performance-robust, representation for every programming task.

Quad ropes roughly retain array efficiency, as long as programmers express their programs using high-level constructs. Moreover, they allow for fast concatenation and dynamic task-based parallelism and are well suited to represent sparse arrays. We describe their operational semantics and evaluate the performance of individual functions on quad ropes as well as declarative algorithms that use our quad rope implementation.

CCS Concepts • **Software and its engineering** → **Very high level languages**; *Functional languages*; *Semantics*; • **General and reference** → *Performance*; • **Applied computing** → *Spreadsheets*

Keywords Declarative arrays, Spreadsheets, End-user development

1. Introduction

Programmers choose data structures for their programs based on their performance properties. For instance, arrays allow random access and update in constant time, and bulk operations are easily parallelizable, whereas linked lists are inherently sequential and random access takes linear time, but adding a value at the head takes constant time. The choice of data structure often has a crucial impact on performance.

For end-user development in high-level declarative programming languages it is impractical to let users choose between different data structures, since they are not primarily educated as programmers. Instead, one should use the same, somewhat performance-robust, representation for every programming task. One declarative language for end-user development is the experimental functional spreadsheet language Funcalc [13], which only distinguishes between scalar values and two-dimensional arrays.

In this paper, 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, 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 [4] and quad trees [7]: a quad rope is a binary tree with one concatenation constructor for each dimension and with small contiguous two-dimensional arrays at its leaves.

1.1 Choosing a Declarative Array Representation

To find an immutable 2D-array representation that efficiently and pragmatically caters to the needs of high-level declarative array programming, let us begin by considering some requirements for a two-dimensional array representation:

1. The data structure must be immutable; immutabillity 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 easy 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 functions 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 persistent or immutable functional arrays using versioning approaches [6, 10] without efficient concatenation. Kaplan and Tarjan [9] showed how to implement fast concatenation of persistent deques, which however do not grant random access.

Stucki et al. [16] 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 con-

* Supported by the Sino-Danish Center for Education and Research.

catenation requires managing many corner cases and often leaves us in situations where we cannot avoid excessive re-allocations.

Finkel and Bentley [7] 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 [14, 15]. A rope is an immutable binary tree with strings or arrays at its leaves [4]. The idea is to group scalar values at the leaves into small contiguous arrays and to extract parallelism by forking threads at each tree branch, where a lower bound on the leaf size avoids excess parallelism. The binary tree structure also allows for constant-time concatenation.

We can easily 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 to immutable two-dimensional arrays, apart from indexing, which runs in logarithmic time instead of constant time. Since we want to encourage a high-level style of array programming using higher-order functions, we deem this acceptable. For small array sizes, quad ropes simply fall back to the default array implementation, eliminating any overhead whatsoever.

1.2 Contributions

In the remainder of this paper, 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 [1] scheduler on quad ropes; and we discuss implementation and performance benchmarks of quad ropes.

2. Quad Rope Semantics

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

Fig. 1 shows the language used to describe quad ropes. We have two basic forms for constructing a quad rope: $\texttt{make}(i, j, r, c, e)$ represents a new quad rope of size $r \times c$ for some initial offsets i and j and an initialization function e, as defined by the rule MK in Fig. 2; whereas $\texttt{rep}(r, c, v)$ creates a quad rope that contains the same value v at all indices, by rule REP. Quad ropes can be concatenated in horizontal or vertical direction using `hcat` or `vcat`, as defined by 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 2D case in the remaining text.

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 later use:

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

The form $\texttt{slc}(i, j, r, c, q)$ describes a rectangular sub-set of a quad rope q and adheres to the slicing semantics of one-dimensional ropes [4]. Finally, indexing and updating (`get` and `set`) take logarithmic time in the `hcat` and `vcat` case on balanced quad ropes, which we will take a closer look at in Sec. 5.2. We omit rules for updating 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.

$e ::=$	n	Number.
	q	Quad rope.
	x	Variable.
	$e(e)$	Function application.
	$\lambda(\bar{x}).e$	Abstraction with $\bar{x} = x_1, x_2, \ldots, x_n$.
	$e \oplus e$	Binary application, short for $\oplus(e, e)$.
$n ::=$	v, i, j, r, c	Numeric constant.
	$n + n$	Addition.
	$n - n$	Subtraction.
	$\texttt{rows}(q)$	Number of rows in q.
	$\texttt{cols}(q)$	Number of columns in q.
	$\texttt{get}(q, i, j)$	Random access.
	$\texttt{red}(e, v, q)$	Reduction binary operator.
$q ::=$	$\texttt{rep}(r, c, v)$	Replicated quad rope of size $r \times c$.
	$\texttt{make}(i, j, r, c, e)$	Build quad rope leaf of size $r \times c$.
	$\texttt{hcat}(q, q)$	Concatenate horizontally.
	$\texttt{vcat}(q, q)$	Concatenate vertically.
	$\texttt{slc}(i, j, r, c, q)$	Build a rectangular subset of values.
	$\texttt{map}(e, q)$	Project by lifting a scalar function.
	$\texttt{zip}(e, q, q)$	Combine two quad ropes pointwise.
	$\texttt{scan}(e, e, v, q)$	Generalized prefix-sum.
	$\texttt{set}(q, i, j, v)$	Set the value of an index pair.

Figure 1. Call by value source language which we use throughout the paper to describe the quad rope semantics. We deliberately do not distinguish between concrete and abstract syntax; we use some syntax restrictions to allow for omitting a definition of static semantics.

It is useful to think of quad ropes not only as binary trees, but also as rectangles that are constructed from smaller rectangles. Fig. 3 illustrates the relationship between quad ropes as binary trees and quad ropes as 2D arrays. There does not exist a single canonical term to describe a quad rope. This is a consequence of having two concatenation forms `hcat` and `vcat`. Two quad ropes that are element-wise equal do not necessarily share the same internal structure.

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

2.1 Projection

The well known $\texttt{map}(e, q)$ form lifts a scalar unary function e to operate on a quad rope q and applies it recursively to all branches and each of their elements, as by rules MAP-MK and MAP-H in Fig. 4. The `zip` form 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 structures of two quad ropes match, we can use rules ZIP-MK to directly combine `make` 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 enough to recurse on the structure of the left hand side argument and to slice the right hand side to match the left hand side's shape, as in ZIP-GEN-H. Fig. 5 illustrates this case for a `hcat` node and another quad rope of different structure.

$$\textsc{mk}\ \frac{e \Downarrow \lambda(x,y).e'}{\texttt{make}(i,j,r,c,e) \Downarrow \texttt{make}(i,j,r,c,e)}\ 0 \leq i \wedge 0 \leq j$$

$$\textsc{rep}\ \frac{}{\texttt{rep}(r,c,v) \Downarrow \texttt{make}(0,0,r,c,\lambda(x,y).v)}$$

$$\textsc{hcat}\ \frac{}{\texttt{hcat}(q_1,q_2) \Downarrow \texttt{hcat}(q_1,q_2)}\ \texttt{rows}(q_1) = \texttt{rows}(q_2)$$

$$\textsc{vcat}\ \frac{}{\texttt{vcat}(q_1,q_2) \Downarrow \texttt{vcat}(q_1,q_2)}\ \texttt{cols}(q_1) = \texttt{cols}(q_2)$$

$$\textsc{rows-mk}\ \frac{q \Downarrow \texttt{make}(i,j,r,c,e)}{\texttt{rows}(q) \Downarrow r}$$

$$\textsc{rows-h}\ \frac{q \Downarrow \texttt{hcat}(q_1,q_2)}{\texttt{rows}(q) \Downarrow \texttt{rows}(q_1)}$$

$$\textsc{rows-v}\ \frac{q \Downarrow \texttt{vcat}(q_1,q_2)}{\texttt{rows}(q) \Downarrow \texttt{rows}(q_1) + \texttt{rows}(q_2)}$$

$$\textsc{slc-mk}\ \frac{\begin{array}{c} q \Downarrow \texttt{make}(i',j',r',c',e) \\ \min(r, r'-i) \Downarrow r'' \\ \min(c, c'-j) \Downarrow c'' \\ i+i' \Downarrow i'' \quad j+j' \Downarrow j'' \\ \texttt{make}(i'',j'',r'',c'',e) \Downarrow q' \end{array}}{\texttt{slc}(i,j,r,c,q) \Downarrow q'}\ 0 \leq i \wedge 0 \leq j$$

$$\textsc{slc-h}\ \frac{\begin{array}{c} q \Downarrow \texttt{hcat}(q_1,q_2) \\ \texttt{slc}(i,j,r,c,q_1) \Downarrow q_1' \\ \texttt{slc}(i, j-\texttt{cols}(q_1), r, c-\texttt{cols}(q_1'), q_2) \Downarrow q_2' \end{array}}{\texttt{slc}(i,j,r,c,q) \Downarrow \texttt{hcat}(q_1',q_2')}$$

$$\textsc{get-mk}\ \frac{\begin{array}{c} q \Downarrow \texttt{make}(i',j',r,c,e) \\ e(i+i', j+j') \Downarrow v \end{array}}{\texttt{get}(q,i,j) \Downarrow v}\ i' \leq i < r \wedge j' \leq j < c$$

$$\textsc{get-h1}\ \frac{q \Downarrow \texttt{hcat}(q_1,q_2)}{\texttt{get}(q,i,j) \Downarrow \texttt{get}(q_1,i,j)}\ j < \texttt{cols}(q_1)$$

$$\textsc{get-h2}\ \frac{\begin{array}{c} q \Downarrow \texttt{hcat}(q_1,q_2) \\ j - \texttt{cols}(q_1) \Downarrow j' \end{array}}{\texttt{get}(q,i,j) \Downarrow \texttt{get}(q_2,i,j')}\ j \geq \texttt{cols}(q_1)$$

Figure 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.

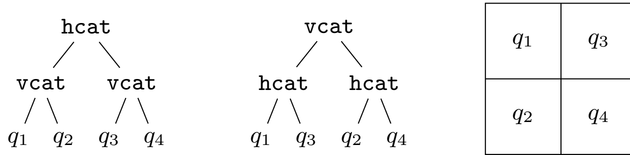

Figure 3. A quad rope illustrated twice as binary tree and once as box diagram. To simplify the example, we assume that $\forall q_i, q_j \in q_{[1,4]}. \boxed{q_i} = \boxed{q_j}$.

2.2 Reduction and Scan

We focus on parallelizable reduction to a single scalar, requiring the operator to be associative and have an identity. Thus all reduction (RED) rules in Fig. 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 2D reduction and slicing to implement row- and column-wise reduction.

One-dimensional scan is usually defined for a binary operator. To make scan as general as possible, our 2D "wavefront" scan uses a function that accepts four arguments: one prefix from the current row, one from the current column and the "diagonal" prefix. Let Σ_a be the prefix sum of some array a for a function f. Then $\Sigma_a[i,j] = f(\Sigma_a[i,j-1], \Sigma_a[i-1,j-1], \Sigma_a[i-1,j], a[i,j])$. This primitive is useful for dynamic programming on 2D arrays.

We describe `scan` semantics in Fig. 7. We use two functions ρ and γ for rows and columns, respectively, to translate row- and column-prefixes from the left branch to the right branch of `hcat` and `vcat` nodes.

The `scan` semantics does not exhibit any obvious possibilities for parallel execution. We will however see in Sec. 5 that there are configurations where it is possible to recursively evaluate some branches in parallel.

3. Block-Sparseness

3.1 Making Replication Canonical

In Sec. 2 we have defined the rule REP, saying that $\texttt{rep}(r,c,v) \Downarrow \texttt{make}(0,0,r,c,\lambda(x,y).v)$. We can optimize the representation of such constant or replicated quad ropes by making the `rep` form canonical. That is, we replace rule REP by REP' in Fig. 8, such that the replication form evaluates to itself. This retains the information that all elements in the resulting quad rope are equal. It is easy to show that $\forall i \in [0,r), j \in [0,c)$:

$$\texttt{get}(\texttt{rep}(r,c,v),i,j) \equiv \texttt{get}(\texttt{make}(0,0,r,c,\lambda(x,y).v),i,j),$$

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

The rule MAP-REP 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. 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 $\varepsilon \oplus \varepsilon = \varepsilon$.

3.2 Block-Sparse Numerical Operations

More well known optimizations for numeric operators apply to replicated quad ropes. For instance, pointwise combination using the `zip` form can make use of sparseness. If we add two quad ropes pointwise to each other and one of them is of the form $\texttt{rep}(r,c,0)$, it is sufficient to evaluate to the other quad rope; this is correct since zero is the identity element for addition.

$$\frac{q_1 \Downarrow \texttt{rep}(r,c,0)}{\texttt{zip}(+, q_1, q_2) \Downarrow q_2}$$

Similarly for pointwise multiplication, which we, among other algorithms, use to implement functional matrix multiplication on quad ropes.

$$\frac{q_1 \Downarrow \texttt{rep}(r,c,1)}{\texttt{zip}(\cdot, q_1, q_2) \Downarrow q_2} \qquad \frac{q_1 \Downarrow \texttt{rep}(r,c,0)}{\texttt{zip}(\cdot, q_1, q_2) \Downarrow q_1}$$

$$\text{MAP-MK}\;\frac{q \Downarrow \texttt{make}(i,j,r,c,e') \quad \lambda(x,y).e(e'(x,y)) \Downarrow e''}{\texttt{map}(e,q) \Downarrow \texttt{make}(i,j,r,c,e'')} \qquad \text{MAP-H}\;\frac{q \Downarrow \texttt{hcat}(q_1,q_2) \quad \texttt{map}(e,q_i) \Downarrow q_i}{\texttt{map}(e,q) \Downarrow \texttt{hcat}(q_1',q_2')}$$

$$\text{ZIP-MK}\;\frac{\begin{array}{c} q_1 \Downarrow \texttt{make}(i_1,j_1,r,c,e') \\ \lambda(x,y).(e'(x+i_1,y+j_1) \oplus \texttt{get}(q_2,x+i_2,y+j_2)) \Downarrow e \end{array}}{\texttt{zip}(\oplus,q_1,q_2) \Downarrow \texttt{map}(0,0,r,c,e)}$$

$$\text{ZIP-H}\;\frac{q_1 \Downarrow \texttt{hcat}(q_{11},q_{12}) \quad q_2 \Downarrow \texttt{hcat}(q_{21},q_{22}) \quad \texttt{zip}(\oplus,q_{1i},q_{2i}) \Downarrow q_i'}{\texttt{zip}(\oplus,q_1,q_2) \Downarrow \texttt{hcat}(q_1',q_2')}\;\texttt{cols}(q_{1i}) = \texttt{cols}(q_{2i})$$

$$\text{ZIP-GEN-H}\;\frac{\begin{array}{c} q_1 \Downarrow \texttt{hcat}(q_{11},q_{12}) \quad \texttt{slc}(0,0,\texttt{rows}(q_2),\texttt{cols}(q_{11}),q_2) \Downarrow q_{21} \\ \texttt{slc}(0,\texttt{cols}(q_{11}),\texttt{rows}(q_2),\texttt{cols}(q_{12}),q_2) \Downarrow q_{22} \quad \texttt{zip}(\oplus,q_{1i},q_{2i}) \Downarrow q_i' \end{array}}{\texttt{zip}(\oplus,q_1,q_2) \Downarrow \texttt{hcat}(q_1',q_2')}$$

Figure 4. Operational semantics for higher-order function application on quad ropes. The ZIP rules have side condition $\boxed{q_1} = \boxed{q_2}$.

$$\texttt{zip}(\oplus,\ \texttt{hcat}(\boxed{q_1}, \boxed{q_2}),\ \boxed{\quad q_3 \quad})$$
$$\Downarrow$$
$$\texttt{hcat}(\texttt{zip}(\oplus, \boxed{q_1}, \boxed{q_{3a}}), \texttt{zip}(\oplus, \boxed{q_2}, \boxed{q_{3b}}))$$

Figure 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.

$$\text{RED-MK}\;\frac{\begin{array}{c} q \Downarrow \texttt{make}(i,j,r,c,e') \\ e'(i,j) \oplus \cdots \oplus e'(i+m-1, j+n-1) \Downarrow v \end{array}}{\texttt{red}(\oplus,\varepsilon,q) \Downarrow v}$$

$$\text{RED-H}\;\frac{\begin{array}{c} q \Downarrow \texttt{hcat}(q_1,q_2) \\ \texttt{red}(\oplus,\varepsilon,q_1) \oplus \texttt{red}(\oplus,\varepsilon,q_2) \Downarrow v \end{array}}{\texttt{red}(\oplus,\varepsilon,q) \Downarrow v}$$

Figure 6. Operational semantics for reduction of a quad rope to a scalar value via the `red` form.

These optimizations can be generalized to any ring. Furthermore, the `zip` form reduces to `map` when one of the arguments is sparse.

$$\frac{q_1 \Downarrow \texttt{rep}(r,c,v)}{\texttt{zip}(\oplus,q_1,q_2) \Downarrow \texttt{map}(\lambda(x).(v \oplus x), q_2)}$$

All sparseness optimizations can also be applied if the right-hand q_2 side is sparse. For the latter optimization, if q_2 is of form `rep`, the right-hand side argument to $\oplus$ is fixed to q_2's v.

4. Two-Way Nodes vs. Four-Way Nodes

As stated in Sec. 2, a given quad rope does not necessarily have a canonical form. In particular, due to having both horizontal and vertical concatenation, $\texttt{hcat}(\texttt{vcat}(q_1,q_2),\texttt{vcat}(q_3,q_4))$ and $\texttt{vcat}(\texttt{hcat}(q_1,q_3),\texttt{hcat}(q_2,q_4))$ 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 $\texttt{node}(q_1,q_2,q_3,q_4)$ that evaluates to itself and let ϵ be the canonical empty quad rope. We define $\texttt{hcat}(q_1,q_2) \Downarrow \texttt{node}(q_1,\epsilon,q_2,\epsilon)$ and $\texttt{vcat}(q_1,q_2) \Downarrow \texttt{node}(q_1,q_2,\epsilon,\epsilon)$.

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 $\texttt{cols}(q_1) < \texttt{cols}(q_2)$, 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 $\texttt{cols}(q_2) < \texttt{cols}(q_1)$, as illustrated in Fig. 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 $\texttt{node}(q_1',\epsilon,\epsilon,q_4'), q_4' \neq \epsilon$ for the case $\texttt{slc}(0,0,r,c,q)$, where $q \Downarrow \texttt{node}(q_1,q_2,q_3,q_4), \forall q_i.\, q_i \neq \epsilon$ and $r \leq \texttt{rows}(q_1),\ c \leq \texttt{cols}(q_1)$, again as illustrated in Fig. 9. This happens regardless of the original structure of q. Slicing q_1 results in empty q_2' and q_3'. If we simply 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 `slc` 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 four-way nodes.

5. Implementation

We have implemented quad ropes[1] in the F# language for the .Net platform [17].

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 `zip` that directly follows the operational semantics from Sec. 2.1. Materialization of slices directly follows from the operational semantics for `slc`. We use explicit materialization internally where appropriate.

Our quad rope implementation uses the .Net Task Parallel Library [11] and pushes a new task to a work-stealing queue for each

[1] Available at `https://github.com/popular-parallel-programming/quad-ropes`

$$\text{SCAN-MK}\ \frac{\begin{array}{c} q \Downarrow \texttt{make}(i, j, r, 1, e) \\ e'(0,0) \Downarrow f(\rho(0), \rho(-1), \gamma(0), \texttt{get}(q, 0, 0)) \\ \forall x, 0 < x.\, e'(x, 0) \Downarrow f(\rho(x), \rho(x-1), e'(x-1, 0), \texttt{get}(q, x, 0)) \\ \forall y, 0 < y.\, e'(0, y) \Downarrow f(e'(0, y), \gamma(y-1), \gamma(y), \texttt{get}(q, 0, y)) \\ \forall x, y, 0 < x, 0 < y.\, e'(x, y) \Downarrow f(e'(x, y-1), e'(x-1, y-1), e'(x-1, y), \texttt{get}(q, x, y)) \end{array}}{\texttt{scan}(f, \rho, \gamma, q) \Downarrow \texttt{make}(0, 0, r, c, e')}\ \rho(-1) = \gamma(-1)$$

$$\text{SCAN-H}\ \frac{\begin{array}{c} q \Downarrow \texttt{hcat}(q_1, q_2) \quad \texttt{scan}(f, \rho, \gamma, q_1) \Downarrow q_1' \\ \lambda(x).(\texttt{get}(q_1', x, \texttt{cols}(q_1') - 1)) \Downarrow \rho' \quad \lambda(y).(\gamma(y + \texttt{cols}(q_1'))) \Downarrow \gamma' \\ \texttt{scan}(f, \rho', \gamma', q_2) \Downarrow q_2' \end{array}}{\texttt{scan}(f, \rho, \gamma, q) \Downarrow \texttt{hcat}(q_1', q_2')}$$

Figure 7. Operational semantics for computing prefix-sums via `scan`.

$$\text{REP'}\ \frac{}{\texttt{rep}(r, c, v) \Downarrow \texttt{rep}(r, c, v)} \qquad \text{ROWS-REP}\ \frac{q \Downarrow \texttt{rep}(r, c, v)}{\texttt{rows}(q) \Downarrow r}$$

$$\text{GET-REP}\ \frac{q \Downarrow \texttt{rep}(r, c, v)}{\texttt{get}(q, i, j) \Downarrow v}\ 0 \le i < r \land 0 \le j < c$$

$$\text{SLC-REP}\ \frac{\begin{array}{c} q \Downarrow \texttt{rep}(r', c', v) \\ r'' = \min(r, r' - i) \\ c'' = \min(c, c' - j) \end{array}}{\texttt{slc}(i, j, r, c, q) \Downarrow \texttt{rep}(r'', c'', e)}\ 0 \le i \land 0 \le j$$

$$\text{MAP-REP}\ \frac{\begin{array}{c} f \Downarrow \lambda(x).e \\ q \Downarrow \texttt{rep}(r, c, v) \quad f(v) \Downarrow v' \end{array}}{\texttt{map}(f, q) \Downarrow \texttt{rep}(r, c, v')}$$

$$\text{RED-REP}\ \frac{q \Downarrow \texttt{rep}(r, c, v) \quad \bigoplus_0^{r \cdot c} v \Downarrow v'}{\texttt{red}(\oplus, v, q) \Downarrow v'}$$

Figure 8. Additional operational semantics for a canonical `rep` form.

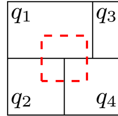

Figure 9. A four-way node configuration $\texttt{node}(q_1, q_2, q_3, q_4)$. The red rectangle illustrated the size of the slice we want to compute. The number in columns in q_4' depend on q_2. Furthermore, the resulting slice consists of only three branches, $\texttt{node}(q_1', q_2', \epsilon, q_4')$.

`hcat` or `vcat` branch. Leaves of the form `make` have size at most $s_{\max}$ in both dimensions, and we merge too-small leaves when their combined size is $\le s_{\max}$ by copying, as for one-dimensional ropes [4]. Hence, the choice of $s_{\max}$ determines the maximum amount of work that should be executed sequentially [1, 15]. Limiting leaf size to $s_{\max}$ allows for a uniform parallelization scheme for all quad rope instances and for fast performance of `set`.

Most bulk operations, such as `map` and `reduce`, can be parallalized in a straightforward fashion. Note that slicing is an inherently sequential operation and hence cannot be performed in parallel.

As noted in Sec. 2.2, the operational semantics for `scan` does not display any obvious opportunities for parallelization. Still, there are two configurations of `hcat` and `vcat` nodes for which `scan` can be parallelized:

$$\texttt{hcat}(\texttt{vcat}(a, b), \texttt{vcat}(c, d)) \tag{1}$$

$$\texttt{vcat}(\texttt{hcat}(a, c), \texttt{hcat}(b, d)) \tag{2}$$

Since `scan` computes the prefix sum from the top left to the bottom right, the sequential dependency for these configurations is $a \prec \{b, c\}$ and $\{b, c\} \prec d$ iff $\texttt{rows}(c) \le \texttt{rows}(a) \land \texttt{cols}(b) \le \texttt{cols}(a)$. This means that, if there is no dependency between b and c, $\texttt{scan}(b)$ and $\texttt{scan}(c)$ can be computed in parallel.

5.1 Lazy Tree Splitting Does Not Apply

Lazy tree splitting [1] is a scheduling technique based on lazy binary splitting [18] and uses ropes [4] to represent parallel collections. The basic idea is to enqueue new tasks on a by-need basis instead of enqueueing new tasks eagerly. Whenever a worker thread is idle, a new task is spawned off to handle half of the remaining work. The check for idle worker threads piggy-backs on efficient work-stealing queues [5] in that it peeks into the shared task queue in a non-synchronized fashion. Thereby, no communication overhead is introduced and synchronization happens whenever other functions (for instance worker threads stealing tasks) force synchronization. If the queue is empty, it is likely that spawning new tasks will pay off performance-wise [1].

At any time, we must be able to stop execution, store the already performed work and then evenly split 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 simply split in two [18].

If the work is stored in some kind of tree, e.g. a rope, we can use a zipper [8] to navigate over the tree and to keep track of the work already done and what remains to do. 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 [1].

Thus we must be able to take the 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`. Fig. 10 shows an example where the current focus is on a leaf 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 leaf, c is not yet processed.

Due to the rectangular invariant, we know that $\texttt{rows}(a) + \texttt{rows}(b) = \texttt{rows}(c)$. If we try to separate a from the quad rope,

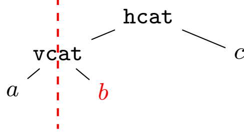

Figure 10. Trying to split a quad rope into two valid quad ropes during lazy tree splitting. In this example, the focus is currently on leaf b. The dashed red line marks where we want to split the quad rope in two.

we cannot use hcat to combine the not yet processed leaves since $\texttt{rows}(b) < \texttt{rows}(c)$. It follows that lazy tree splitting does not apply to quad ropes, because we cannot pause execution at arbitrary leaves.

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 height 1 can only be concatenated by vcat, the problem does not occur and lazy tree splitting is possible again.

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

5.2 Balancing

If the quad rope tree is highly imbalanced, for instance a linear list, our recursive parallelization scheme achieves only sequential execution. Also, indexing operations would require linear time, which is unacceptable. Hence quad rope trees should be kept balanced.

Rebalancing of a one-dimensional binary tree can be implemented via rotation in logarithmic time after an insertion or deletion. We use a depth-metric to determine whether to rotate a quad rope, as illustrated by the following F# style pseudo-code, where we match on the constructor forms from Fig. 1:

```
let rec depth = function
  | mk _ | rep _ ⇒ 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 [3] modulo leaf size.

A quad rope can be rotated only in limited ways. We can rotate nested hcat applications and nested vcat applications, but not alternating applications of hcat and vcat:

```
let cond(a, b, c) =
  depth(a) ≠ depth(b)
   ∧ max(depth(a), depth(b)) > depth(c)

let rec balance = function
  | hcat(hcat(a, b), c) when cond(a, b, c) ⇒
     hcat(a, balance(hcat(b, c)))
  | vcat(vcat(a, b), c) when cond(a, b, c) ⇒
     vcat(a, balance(vcat(b, c)))
  | ... (* Mirror cases omitted. *)
  | qr ⇒ qr
```

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

$$\texttt{depth(hcat(hcat(a, b), c))} =$$
$$\texttt{max(max(depth(a), depth(b)) + 1, depth(c)) + 1}.$$

If cond holds, at least one of a and b is deeper than c. If depth(a) is greater than depth(b), the balanced tree hcat(a,hcat(b,c))) has a depth of depth(a)+1. If however depth(a) is less than depth(b), the depth of b is defining the resulting depth, hence depth(hcat(hcat(a,b),c) = depth(hcat(a,hcat(b,c)).

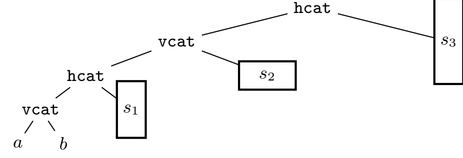

Figure 11. A quad rope constructed in adversarial manner without balancing, using hcat and vcat alternatingly, where the right-hand children s_i are instances of rep. The size of the box indicates the shape of the sparse rep leaf that it represents.

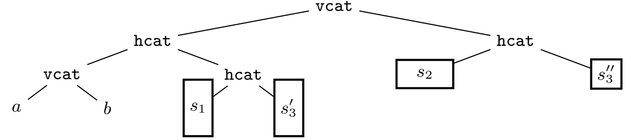

Figure 12. The adversarial quad rope from Fig. 11 after balancing. The leaf s_3' is the "upper two thirds" and s_3'' is the "lower third" of s_3.

Even though we cannot balance across dimensions, there is a point to looking into *adversarial* cases, where a quad rope is composed of a chain of alternating hcat and vcat instances at every other node. It is not obvious to us whether this adversarial pattern is common. If one of the branches is a rep leaf, as illustrated in Fig. 11, we can use slicing and re-distribute the sliced rep 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 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), rep(r, c, e))
     when depth(vcat(a, b)) > 2 ⇒
      let s₁ = rep(rows(a), c, e),
          s₂ = rep(rows(b), c, e) in
        vcat(balance(hcat(a, s₁)),
             balance(hcat(b, s₂)))
  | vcat(...) (* Swap hcat and vcat. *)
  | ... (* Mirror cases omitted. *)
  | qr ⇒ qr
```

The result of this extended balancing algorithm, applied to the quad rope from Fig. 11, is shown in Fig. 12. We never perform balancing if the remaining depth of the non-rep child is less than or equal to 2. The worst-case complexity of balancing is $O(n \log n), n = \max(r, c)$. 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 balancing invariant.

5.3 Memory Allocation

Even though the semantics allows for lazy evaluation of quad rope leaves, we have implemented quad ropes with contiguous 2D-arrays at the leaves to avoid repeated computation of values. A performance bottleneck of this implementation is leaf array allocation during quad rope creation. Consider the $\texttt{init}(r, c, f)$ function where r and c are row and column counts and f an initialization function. The row and column counts are alternately divided by two until they are at most $s_{\max}$. At this point, a naive implementation

would allocate a new 2D-array as a leaf and initialize it with f for the appropriate offsets. The leaves are then concatenated via `hcat` and `vcat`. This results in $O(\max(\frac{r}{s_{\max}}, \frac{c}{s_{\max}}))$ array allocations.

It turns out that allocating one large array on .Net is not slower than allocating one 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 first pre-allocate one large 2D-array and fill it imperatively. We use array pre-allocation in all functions that construct a new quad rope, e.g. `map` and `scan`. Therefore, we store an additional sparseness flag at each node which we check at each recursive step to not unnecessarily allocate memory for sparse branches of a quad rope. Note that the nodes of a quad rope are fully immutable.

We use an immutable view abstraction over the underlying large 2D-array to represent leaves. A positive side effect of this is that we can slice views on 2D-arrays in constant time. Hence, materializing a quad rope slice, as discussed in Sec. 5, only requires re-allocation of the tree structure which takes logarithmic time on balanced quad ropes.

Last, we can use the underlying array to implement `scan` without explicit prefixes ρ and γ, since all prefix values are accessible via the shared array.

6. Performance

We use an extended version of the .Net benchmarking infrastructure by Biboudis et al. [2][2]. Our test machine is an Intel Xeon E5-2680 v3 with 32 cores at 2.5 GHz and 32GB of memory, running 64 bit Windows 10, Version 1607, and .Net Framework 4.6.2. 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 a quarter of a second [12].

6.1 Individual Functions

Fig. 13 shows performance data for individual functions on quad ropes. We can see that sequential performance of higher-order functions on quad ropes varies, but it is generally comparable to standard immutable arrays. We can attribute the modest speed-up of `map` and `reduce` to increased locality. We get parallel speed-ups starting from four threads; this is likely due to the overhead of eager task creation, which we cannot make up for with less hardware threads. However, we cannot achieve linear parallel speedups with quad ropes using the .Net TPL [11] and recursive task creation.

Since indexing is asymptotically worse on quad ropes than on arrays, we compare getting and setting pseudo-random indices on both. The benchmark results are displayed in Fig. 14. Note that, even though indexing is much slower than on arrays, setting is much faster, since in the worst case we only need to reallocate an array of size $s_{\max} \times s_{\max}$ and a tree of depth $O(\log n)$.

6.2 Declarative Algorithms

Fig. 15 shows benchmark results for a collection of high-level functional algorithms. Matrix multiplication uses slicing, zipping and reduction and has a high level of nested parallelism. We multiply a pseudo-random matrix to an upper diagonal matrix of ones, once dense and once sparse. Note that a standard imperative algorithm on mutable arrays of the same size using in-place updates is faster by roughly a factor of ten. The algorithm for computing *Van Der Corput* sequences uses repeated concatenation of singletons in between two larger arrays and mapping over the result. *Fibonacci* is the classic recursive algorithm using indexing and concatenation. *Sieve of Erastothenes* uses persistent `set` to modify an array of values repeatedly in order to compute all primes up to a given number.

[2] See `https://github.com/biboudis/LambdaMicrobenchmarking`.

Benchmark	ms	× Arr	$t=2$	4	8	16
`init`	7.06	0.8	0.81	1.51	2.12	1.98
`map`	15.9	1.48	1.26	2.38	3.96	3.88
`reduce`	18.53	1.48	1.31	2.34	4.12	4.24
`zip`	17.16	1.05	1.02	1.61	2.43	1.75
`scan`	4.82	0.97	1.11	2.04	2.99	3.58

Figure 13. Results of performance benchmarks on quad ropes for $s_{\max} = 32$ and size 1000×1000, double precision. The second column shows the average absolute performance of sequential quad ropes in milliseconds. All other values are speed-up factors; higher is better. The third column shows sequential quad rope performance relative to the performance of standard immutable 2D-arrays. The following columns show performance of parallel execution with t hardware-threads, relative to sequential execution.

Benchmark	$n=10$	100	1000
`get`	0.38	0.21	0.12
`set`	1.56	21.52	1790.42

Figure 14. Results of performance benchmarks for indexing operations on quad ropes for $s_{\max} = 32$ and size $n \times n$. Values are speed-up factors relative to standard immutable 2D-arrays; higher is better. We generate index pairs pseudo-randomly.

Smith-Waterman is the functional array variant of a standard algorithm for computing the edit distance of two character sequences and uses a sequence of `scan` and `reduce`.

We can observe that quad ropes are faster than standard immutable 2D-arrays already in the sequential function variants, with the `Smith-Waterman` algorithm being the exception. For dense matrix multiplication, the moderate performance increase can be explained by faster slicing; sparse matrix multiplication uses the optimizations from Sec. 3.2, hence the increased speed-up.

Van Der Corput benefits from fast concatenation of quad ropes but seems restricted by calling `map`; *Fibonacci* on quad ropes runs more than five times faster than on 2D-arrays; and *Sieve of Erastothenes* is more than 44 times faster on quad ropes compared to standard 2D-arrays, as the latter performs $O(n^2)$ work when reallocating the array during setting individual index pairs. Again, the `scan`-based *Smith-Waterman* algorithm does not quite keep up with the 2D-array variant.

None of the parallelizable algorithms on quad ropes scale very well with an increased number of threads, even though quad ropes are able to exploit some of the nested parallelism expressed in matrix multiplication. The parallel slow-down of *Van Der Corput* is difficult to interpret. Profiling shows that roughly a quarter of the time is spent in `map`. One could suspect that repeatedly concatenating a singleton in between larger quad ropes may, in combination with balancing, result in a large number of singleton leaves, leading to excess parallelism. That is not the case, since we combine small leaves during balancing further down the tree. The parallel speed-up for *Smith-Waterman* is modest. The declining speed-up at $t = 16$ may be due to less exploitable parallelism, enforced by sequential dependencies.

Even though quad ropes do not excel in their parallel performance, it should be noted that the parallel algorithms on quad ropes perform much better than on standard immutable 2D-arrays (numbers elided). Especially nested parallelism seems to be a major bottleneck of the TPL's parallel for-loops.

The choice of $s_{\max}$ has an immediate effect on quad rope performance. Larger values decrease the overhead of task creation but affect cache locality negatively and encourage more frequent merging of leaf arrays. Smaller values work the other way around.

Benchmark	n	ms	$\times$ Arr	$t = 2$	4	8	16
Matrix mult. dense	100×100	44.01	1.75	0.64	1.03	1.28	1.52
Matrix mult. sparse	100×100	29.56	2.6	0.61	0.48	1.35	1.54
Van Der Corput	20	24.25	2.43	0.48	0.64	0.43	0.6
Fibonacci	1600	1.9	5.69	–	–	–	–
Sieve of Erastothenes	1600	1.34	44.92	–	–	–	–
Smith-Waterman	1000×1000	980.65	0.94	1.19	1.04	1.38	1.04

Figure 15. Results of performance benchmarks on quad ropes for $s_{\max} = 32$. The value n describes the input size. See Fig. 13 for explanation of the other columns. Our variants of the *Fibonacci* and *Sieve of Erastothenes* algorithms are not parallelizable, so no parallel performance data is available.

7. Conclusion and Future Work

In this paper, we have presented quad ropes, a two-dimensional extension of the rope data structure [4] and inspired by quad trees [7], for high-level, functional, parallel array programming. Quad ropes don't generally outperform standard, immutable two-dimensional arrays, but gracefully handle array anti-patterns as well as nested parallelism. This makes them a useful array representation in high-level functional languages where expressiveness is of importance.

The performance of quad ropes is on par with immutable 2D-arrays. Concatenation in both dimensions is asymptotically and practically faster than on arrays, as discussed in Sec. 6. Parallel speed-ups of individual functions are sub-linear and modest for more complex functions. Understanding these results in detail is future work. Nevertheless, sequential quad ropes run faster than arrays for the large majority of the presented algorithms. Even though .Net is very well engineered, it would be worthwile to implement and benchmark quad ropes outside of a managed platform.

We have given an operational semantics to describe the quad rope data structure and shown that lazy tree splitting scheduling [1] 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. It is possible that a modified memory pre-allocation mechanism, as discussed in Sec. 5.3, allows for gradual flattening of quad ropes during bulk operations, e.g. `map`. Flattened quad ropes, essentially leaves larger than $s_{\max}$, could allow us to use a lazy scheduling algorithm using indices [18]. We consider gradual flattening as future work.

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. We cannot currently balance quad ropes across dimensions without breaking them and therefore consider this as future work as well.

Finally, our plan is to use quad ropes as the main array implementation in the Funcalc [13] spreadsheet language to evaluate their usefulness in a very high-level language.

Acknowledgments

We would like to thank Alexander Asp Bock for useful discussions and feedback on an earlier draft, and the anonymous reviewers for their useful comments.

References

[1] L. Bergstrom, M. Rainey, J. Reppy, A. Shaw, and M. 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.

[2] A. Biboudis, N. Palladinos, and Y. Smaragdakis. Clash of the Lambdas, July 2014. URL `http://arxiv.org/abs/1406.6631`.

[3] G. E. Blelloch. Programming Parallel Algorithms. *Commun. ACM*, 39(3):85–97, Mar. 1996.

[4] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: an Alternative to Strings. *Software – Practice & Experience*, 25:1315–1330, Dec. 1995.

[5] D. Chase and Y. 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.

[6] P. 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.

[7] R. A. Finkel and J. L. Bentley. Quad Trees A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, Mar. 1974.

[8] G. Huet. The Zipper. *Journal of Functional Programming*, 7(05): 549–554, Sept. 1997.

[9] H. Kaplan and R. E. Tarjan. Persistent Lists with Catenation 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.

[10] A. Kumar, G. E. Blelloch, and R. 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.

[11] D. Leijen, W. Schulte, and S. 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.

[12] P. Sestoft. Microbenchmarks in Java and C#. Lecture Notes, Sept. 2013. URL `https://www.itu.dk/people/sestoft/papers/benchmarking.pdf`.

[13] P. Sestoft. *Spreadsheet Implementation Technology: Basics and Extensions*. The MIT Press, Sept. 2014. ISBN 0262526646.

[14] G. 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.

[15] G. 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, Aug. 2009. ACM.

[16] N. Stucki, T. Rompf, V. Ureche, and P. 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.

[17] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 4.0*. Apress, Berkely, CA, USA, 4th edition, 2015. ISBN 1484207416, 9781484207413.

[18] A. Tzannes, G. C. Caragea, R. Barua, and U. 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, Jan. 2010. ACM.