Progress Report: Exploring API Design for Capabilities for Programming with Arrays

Beatrice Åkerblom
Computer and Systems Science
Stockholm University
beatrice@dsv.su.se

Elias Castegren
School of EECS
Royal Institute of Technology
elica@kth.se

Tobias Wrigstad
Information Technology
Uppsala University
tobias.wrigstad@it.uu.se

Abstract

In on-going work, we are exploring reference capabilities for arrays, with the intention of carrying over previous results on statically guaranteed data-race freedom to parallel array algorithms. Reference capabilities typically restrict incoming pointers to an object to one (uniqueness), or restrict operations via multiple pointer to a single object (*e.g.,* to only read). Extending such a design to arrays involves operations such as logically partitioning an array so that even though there are multiple pointers to a single array, these pointers cannot access the same elements.

In this paper, we report on the on-going work of a prototype implementation of array capabilities, focusing in particular on the "array capability API design", meaning the native operations on capabilities such as splitting and merging arrays. Using our prototype implementation, we translate several existing array algorithms into using array capabilities and qualitatively study the result. In addition to identifying the need for additional operations, we study what features are commonly exercised, what are the recurring patterns, and how reliance on direct element addressing using indexes can be reduced. We end by discussing a possible design for a more performant implementation once the API is fixed.

CCS Concepts • **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Data types and structures**;

Keywords Type systems, Capabilities, Parallelism, Arrays

ACM Reference Format:
Beatrice Åkerblom, Elias Castegren, and Tobias Wrigstad. 2019. Progress Report: Exploring API Design for Capabilities for Programming with Arrays. In *Proceedings of ICOOOLPS'19*. ACM, New York, NY, USA, 7 pages.

1 Introduction

Array algorithms where operations are applied to disjoint parts of an array are easily adjusted to support parallelism, since parallel threads can operate on the array parts simultaneously without need for synchronisation. The adjustment, however, requires careful attention by programmers to ensure that two threads never update the same element. Common mistakes such as off-by-one-errors potentially lead to data-races and non-deterministic program behaviour. The accidental aliasing of indexes in a parallel array algorithm shares many similarities with aliasing as known from object-oriented languages, a phenomenon that has a long and rich history in programming language research. Many different techniques are available to statically prevent or dynamically manage aliasing. One of the techniques that can be used to statically detect and prevent accidental aliasing is by using reference capabilities, which give a static guarantee that programs are free from data-races, for example the Kappa type system [6, 7] as implemented in the object-oriented actor language Encore [5]. Recently, the reference capabilities of Kappa have—in the context of the language calculus *Arr-O-Matic*—been extended with array capabilities [1] that provide built-in support for divide-and-conquer style subdivision of arrays while preserving the guarantees of data-race freedom.

This paper discusses initial work on the prototype implementation of Arr-O-Matic, which lays the foundation for the integration of array capabilities in Encore. A prototype implementation outside of a static type system allows us to explore questions regarding the "API design" of the array capabilities: what are the frequent operations on array capabilities in typical parallel array algorithms; which are the fundamental operations from which most other operation can be constructed; and how shall these be exposed to programmers?

To answer these questions, we are implementing common array algorithms, parallel and sequential, using array capabilities paying close attention to requirements on the capabilities from these algorithms, and also exploring how we can decrease the reliance of explicit index addressing.

After a brief background on reference capabilities and arrays (Sections 2 and 3), we present our progress so far:

- We report on our prototype Python implementation, which focuses on the functional interface of array capabilities (Section 4).
- We investigate the expressiveness of our prototype by presenting a selection of implemented array algorithms (Section 5).
- We discuss and highlight our preliminary findings (Section 6).

ICOOOLPS'19, , London, UK
2019.

```
void p_ip_map(int *a, int start, int end, int(*f)(int)) {
    if (end - start <= SEQ_CUTOFF) {
        for (int i = start; i < end; ++i) a[i] = f(a[i]);
    } else {
        int midpoint = start + (end - start) / 2;
        cilk_spawn recursion(a, start, midpoint);
        recursion(a, midpoint, end);
        cilk_sync;
    }
}
```

Figure 1. The function above implements a parallel, in-place map in Cilk [2]. The key arguments are an array, `a`, and the indexes of a start and end part of `a` to operate on. Although all concurrently executing `p_ip_map` functions share the same array, the correct manipulation of these indexes ensure that they will not access the same elements.

2 Reference Capabilities and Arrays

A reference capability is a programming construct that can be used to control access to a certain object and what kind of operations the holder of a specific reference capability is allowed to perform on that object. Reference capabilities can be used to guarantee absence of data-races by ensuring the existence of only a single reference to an object, and/or permit aliases but limit access to only allow reading. (See *e.g.,* [7] for additional variants and details.)

In many languages, (*e.g.,* Scala; Haskell) arrays are treated in the same way as any other kind of object or value, whereas other languages, (*e.g.,* C; Java; Rust) have dedicated syntax for basic interaction (accessing and updating) and creation of arrays. Often, there is library support (*e.g.,* `java.util.Arrays` in Java) for more involved operations, like sorting or cloning.

Both approaches have merit; nevertheless some characteristics (like its low-level nature) and some ways that arrays are used (like sorting and index-based splitting) distinguish them from other objects. This becomes even more visible when combining reference capabilities with arrays in parallel programs. Parallelisation of array algorithms is often achieved through partitioning an array into discrete parts and letting the concurrent tasks operate on one part each. But unlike splitting composite object structures, like trees into sub-trees, array partitioning does not result in several references into disjoint parts of a structure. Instead, it involves (typically) sharing a common base pointer and performing pointer arithmetic-like operations using indexes, all of which are threaded through a computation. A typical example of such a parallel array operation involving splitting an array by indexes can be found in Fig. 1.

Because of this, common array algorithms cannot rely on reference capabilities for alias and access control, since the latter operate at the object level. To maintain the guarantees that the program contains no data-races, the type system will not allow aliases.

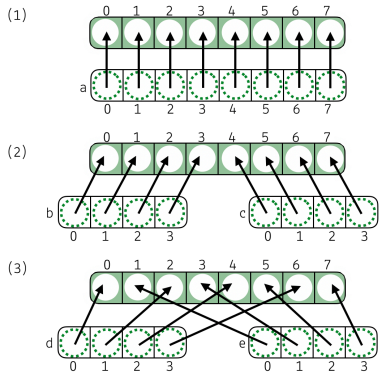

Figure 2. Scenarios 1–3 show five ways array capabilities $a - e$ can give access to (part of) a green, physical array.

3 Array Capabilities in Arr-O-Matic

The array capabilities introduced in Arr-O-Matic [1] are similar to reference capabilities in that they all grant access to some resource, and that access may come with some restrictions (*e.g.,* on the rights to read or write) that can not be modified. Moreover, array capabilities also permit split operations that will create new, disjoint, array capabilities through which only parts of the original array can be reached.

The array capability provides an abstraction of an array which may, transparently to the programmer, grant access only to part of the underlying physical array stored in memory. Furthermore, elements i and $i + 1$ in the array capability may not be adjacent in the underlying array. Fig. 2 illustrates this: (1) shows an array and an array capability a which are perfectly aligned, *i.e.,* they have the same length, and all adjacent indexes in the *logical* view will refer to adjacent elements in the *physical* array. In (2) there are two array capabilities each mapping to its own consecutive half of the underlying array so that $b[i] = a[i]$ and $c[i] = a[i + 4]$ and $len(b) = len(c) = len(a)/2$. In (3), d and e instead split the array in a strided fashion so that $d[i] = a[i \cdot 2]$ and $e[i] = a[1 + i \cdot 2]$.

Array capabilities support operations like splitting, merging etc. in different ways. The concrete implementations of these operations are discussed in section 4, but at the conceptual level they function as follows:

split An array capability a can be split into n different sub capabilities $c_{1\ldots n}$ by taking m elements from a into c_1, then the following m elements into c_2, etc. and starting over with c_1 if there are more than $n{\cdot}m$ elements. When $length(a) = n \cdot m$ the array capability is effectively divided into n consecutive chunks like (2) in Fig. 2 ($n = 2, m = 4$). Otherwise, the results are strided capabilities like (3) in Fig. 2 ($n = 2, m = 1$). Consecutive splits show up naturally in divide-and-conquer algorithms, while

strided splits are useful *e.g.,* for operations on matrices represented as arrays.

merge Two or more array capabilities can be merged together into one logical, composite capability. Merge is the dual of splitting. For example, the two array capabilities $[a, b, c]$ and $[d, e, f]$ can be merge-concatenated as $[a, b, c, d, e, f]$ or $[d, e, f, a, b, c]$; merge-strided as *e.g.,* $[a, d, b, e, c, f]$ or $[a, b, d, e, c, f]$ (for $m = 2$).

align The align operation aligns an underlying array with an array capability. For example, consider the merge-concatenation of d and e in Fig. 2 into a new capability f. Now, if we apply align to f, the elements of the underlying array will be swapped in-place, so that the elements on positions 0, 2, 4 and 6 are moved into positions 0–3, and the elements on positions 1, 3, 5 and 7 are moved into positions 4–7 respectively.

Because splitting and merging are logical operations, they can be applied multiple times to incrementally rearrange the order of the elements in an array. If the logical order resembles the program's access order, an align operation is probably a sensible optimisation (for cache locality), but does not change the semantics of the program.

A splitting operation, which will result in two or more new array capabilities emerging from the same physical array, will not result in the creation of any actual array objects or copying of elements. The only extra run-time overhead needed consists of index translation.

An important requirement on array capabilities is that they must govern access to disjoint parts of their underlying physical array (or they must only allow reading their elements.) This guarantees data-race freedom for arrays of primitive elements or immutable elements. Splitting and merging must uphold this invariant. In the statically typed Arr-O-Matic design, merging two array capabilities *consumes* these capabilities, and splitting a capability into several smaller capabilities consumes the original. To simplify programming with array capabilities, standard borrowing and burying [3, 4] techniques can be applied. Borrowing creates a copy of a capability and temporarily hides the original, until the copy (including copies constructed from the first copy) is provably out of scope and thus never used again. Ultimately, this allows safe temporary aliasing.

In previous work we explored splitting and merging theoretically, in a context with static types and borrowing [1]. As we now explore API design, and not type system design, we focus on the dynamic operations and do not care much about borrowing and burying. In the following sections we discuss our experience from implementing a prototype of the system and implementing algorithms to test our design.

4 Implementation

The prototype has been implemented in Python, since the goal has been to experiment with API design rather than performance and type system. Once the API stabilises, our experience from the prototype will serve as a basis for the actual implementation for Encore. Unlike the prototype, the Encore version will also facilitate actual parallelism.

Array capabilities are instances of the ArrayView class [9]. An array capability holds a reference to its underlying array in `data`. For simplicity, all `ArrayView` objects also have a "translation array" (field `translation`) that is a precomputed table of index translations, *i.e.,* on index i it holds the corresponding index j in the underlying array. For the array capability c in Fig. 3, the index 0 in the translation array redirects to index 4 in the underlying array.

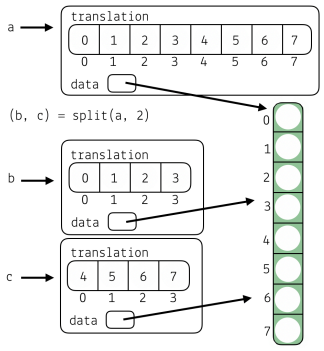

Figure 3. Array a references its underlying *data* and an array for index translation. $(b, c) = split(a, 2)$ splits a into two new array capabilities, b and c, each with indexes [0–3], sharing the same underlying array. b's translation array will return the same index as looked up, but in c 4 has been added to all indexes to access elements [4–7] in *data*.

Array capabilities support the basic operations expected when operating on an array: we can read and write from and to its indexes, we can check its length, we can compare two `ArrayView` objects with each other, etc.

More importantly, we can *split*, *merge*, and *align* `ArrayView` objects in different ways.

Split Splitting is implemented as a function (or method in a capability object) taking three arguments: the array we want to operate on; how many parts the array will be split into; whether the split will be consecutive or strided. A consecutive split into two new arrays is what happens in the split operation performed in Fig. 3. The example (3) in Fig. 2 illustrates the result of a strided split into two new arrays.

Merge Merging is an operation on an array of `ArrayView` objects as its first argument, and a second one to determine if the merge is consecutive or strided. A consecutive merge will concatenate the `ArrayViews` in the order of the argument array: merging `[[1, 2], [3, 4]]` returns a new capability `[1, 2, 3, 4]`. A strided merge interleaves elements from the arrays: merging `[[1, 2], [3, 4]]` returns a new capability `[1, 3, 2, 4]`.

Align The align operation is only used for its side-effects. Align changes the element order in the underlying array to match the logical order of the array capability so that

translation is no longer necessary. For now, we do not support alignment on capabilities that do not govern an entire underlying array (see discussion in Section 6.2).

Let c be an array capability constructed by splitting the array a `[1, 2, 3, 4]` into `[1, 2]` and `[3, 4]` and then merging these in a strided fashion into `[1, 3, 2, 4]`. Note that the elements in a still have the same order as they had in the beginning. After a call to **`align`**`(c)`, a would be changed to `[1, 3, 2, 4]`, and c's translation array would become `[0, 1, 2, 3]`. (A more performance-oriented implementation would drop the translation array completely in this case.)

4.1 Static vs. Dynamic Index Translation

Obviously, creating a precomputed table of index–index translations is not a reasonable implementation in most cases: it will consume a large amount of memory which is not feasible with big arrays, but furthermore will stress the memory subsystem. Moving the overhead from memory to CPU is certainly possible by computing the translations on-the fly. These translation functions are not complicated for consecutive and strided splits, and function composition can be used to implement recursive splitting and merging. Some splits and merges are each other's duals, and it is desirable to detect this to not have to effectively calculate an expensive identity function.

Consider for example the splitting of a in the **`align`** example above. The translation functions for by-index access in the two capabilities would be $f_0(i) = i$ and $f_1(i) = i+2$ respectively. After the strided merge (but before the call to **`align`**), the single translation function of the resulting capability c *could be*:

$$f_2(i) = \begin{cases} f_0(i/2) & \text{if } even(i) \\ f_1(\lfloor i/2 \rfloor) & \text{if } odd(i) \end{cases}$$

As shown above, we can build on the existing translation functions as we go.

5 Array Algorithms with Capabilities

To explore how far the minimal API design—split, merge, and align—takes us, we selected algorithms solving important or typical array-related problems for implementation. The goal of this was to identify potential limitations in the previously proposed APIs and to extend them if necessary. So far we have implemented: several sorting algorithms (quicksort, mergesort, and bubblesort); various matrix operations (including creation, lookup, addition, multiplication, rotation, tiling and stencils); and parallel array reduction.

In this section, we look at four of these. For brevity, we omit the sorting algorithms from this text.

5.1 Matrix Rotation

Matrix rotation has internally been the "Hello, world!" application for array capabilities because it exercises all the three fundamental operations.

Let a be a 4×4 matrix encoded in an array in a column major fashion. We can rotate this matrix to make row major access possible producing b by first performing a strided split of the array **`split`**`(matrix, 4, strided=True)` by the number of columns and then do a merge-concatenation on the result of the split **`merge`**`(split_result, concatenate=True)`.

a	b	c	d
a	b	c	d
a	b	c	d
a	b	c	d

a

a	a	a	a
b	b	b	b
c	c	c	c
d	d	d	d

b

Note that if it is desirable that the underlying physical array matches the layout of the capability, this can be accomplished by adding a **`align`**`(matrix)` at the end (typical matrix tiling operation).

The following lines show various rotation and mirroring operations:

```
# Rotate counter clockwise
matrix = merge(reverse(split(matrix, Cols, True)), True)
# Rotate clockwise (alternative to version shown above)
matrix = merge(reverse(split(matrix, Cols, False)), False)
# Mirror matrix horizontally
matrix = merge(reverse(split(matrix, Cols, True)), False)
# Mirror matrix across main diagonal
matrix = merge(split(matrix, Cols, True), True)
# Mirror matrix across main diagonal (alternative version)
matrix = merge(split(matrix, Cols, False), False)
# Mirror matrix vertically
matrix = merge(reverse(split(matrix, Cols, False)), True)
```

The function `reverse` above simply reverses the order of elements in an array (our prototype implementation uses the standard Python reverse built-in). Again, adding an **`align`** to any of these operations would turn the logical permutations into physical ones.

5.2 Matrix Partitioning (More Tiling)

Let a be a 4×4 matrix encoded in an array in a column major fashion. We would like to create four 2×2 square tiles from a. This can be achieved by splitting the matrix twice. First `left, right =` **`split_by`**`(a, 2, 2)` splits a into two 2-by-4 sub-matrices (this operation, which distributes 2 elements per stride over 2 capabilities, is notably not part of the standard API, but can be easily encoded by splitting and merging in an interleaved fashion). To further get the top-left square tile, `top_left, bottom_left =` **`split`**`(left, 2)` we simply divide the left sub-matrix in half.

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

`left`

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

`top_left`

5.3 Array Reduction

Many array reduction algorithms reduce arrays in-place, by storing intermediate results in the array itself, for example:

```
for v in values[1:]:
    values[0] += v
```

In the context of CUDA, Harris [8] explores different strategies for parallelising array reduction, often relying on using thread IDs as part of index calculations to ensure that no two threads access the same array elements. For example, first using N threads to reduce $2 \cdot N$ elements, either adjacent or with a stride of N and storing the results in the first element's place, then $N/2$ threads for summarising the result, etc. We were pleased at how straightforward the encoding in Arr-O-Matic was—it allowed defining a single function parameterised over whether the reduced elements are adjacent or not (comments show where `cilk_spawn` and `cilk_sync` could be inserted to trivially parallelise the operation).

```
while parts > 0:
    reduce_n(split(array, parts, strided), x, not strided)
    parts = parts // 2
    x = x * 2

def reduce_n(array_of_arrays, N, S):
    for a in array_of_arrays:
        reduce_1(a, N, S) # cilk_spawn
    # cilk_sync

def reduce_1(array, N, S):
    win = fst(split(array, N, S)) # fst(x) = x[0]
    for e in win[1:]:
        win[0] += e
```

If `strided` is `True`, then each `reduce_1` operates on a capability consisting of interleaved parts of the underlying array. Operating on a 16-element array and 8 "parallel" reducers, the intermediate steps produced are shown below, where colour shows what places in the array are being updated at each step.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10	12	14	16	18	20	22	24	9	10	11	12	13	14	15	16
28	32	36	40	18	20	22	24	9	10	11	12	13	14	15	16
64	72	36	40	18	20	22	24	9	10	11	12	13	14	15	16
136	72	36	40	18	20	22	24	9	10	11	12	13	14	15	16

If `strided` is `False`, the underlying array is instead "broken up" into consecutive parts which are reduced. Operating on a 16-element array and 8 "parallel" reducers, the intermediate steps produced are shown below, where colour shows what places in the array are being updated at each step.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3	2	7	4	11	6	15	8	19	10	23	12	27	14	31	16
10	2	7	4	26	6	15	8	42	10	23	12	58	14	31	16
36	2	7	4	26	6	15	8	100	10	23	12	58	14	31	16
136	2	7	4	26	6	15	8	100	10	23	12	58	14	31	16

5.4 Stencil Application on 2D Matrix

Stencil application at an element (i, j) in a 2D matrix is the computation of a function (*e.g.*, a sum) over some neighbouring elements. Typically, algorithms that perform stencil applications applies the stencil to all elements in a matrix.

A stencil application typically uses two matrices—one read-only and one store-only. In our test scenario, we read a plus shaped stencil with a width and height of N elements, add the $2N - 1$ elements of each stencil and write it to the centre position of the stencil in the write-only matrix.

```
# Typical index–based read operation (wraps around)
def matrix_read(matrix, x, y, rows, cols):
    return matrix[(y % rows) * cols + (x % cols)]

# Sum all the elements in a stencil
def stencil_sum(matrix, x, y, length, rows, cols):
    sum = matrix_read(matrix, x, y, rows, cols)
    for i in range(1, length + 1):
        sum += matrix_read(matrix, x, y + i, rows, cols)
        sum += matrix_read(matrix, x + i, y, rows, cols)
        sum += matrix_read(matrix, x, y - i, rows, cols)
        sum += matrix_read(matrix, x - i, y, rows, cols)
    return sum
```

The stencil application happens on a row-by-row basis. Outside of Python, this code would have each row (or some selection of rows) in parallel threads. The `do_row()` function applies the stencil to each cell in the row.

```
def do_row(array, y, m, rows, cols, width):
    for x, cell in enumerate(split(array, len(array))):
        cell[0] = stencil_sum(m, x, y, width, rows, cols)
```

Finally, to start the operation, we borrow both the read and write matrices, split the write-only matrix by row, and pass each row to the `do_row()` function to do its work.

```
read = create_matrix(Rows, Cols)
write = create_matrix(Rows, Cols)

for y, row in enumerate(split(write, Rows, False)):
    do_row(row, y, read, Rows, Cols, 2)
```

Notably, the only uses of the array capability API are split, which is used twice.

6 Concluding Remarks

We have identified a number of additional, "convenience" functions for array capabilities, which can be encoded using the existing API. These are discussed in turn.

Split At When implementing quicksort (see [1]), we saw that our splitting operations were not expressive enough. The algorithm required splitting at a specific index, which is not necessary in the middle. This resulted in adding the "split at" function, which takes an `ArrayView` object and an index as arguments. The function returns an array with two `ArrayView` objects, where the first one will contain the elements from position 0 to the element before the splitting

position. Splitting an `ArrayView [1, 2, 3, 4]` at 0 would for example return `[[], [1, 2, 3, 4]]` while splitting the same `ArrayView` at 2 would return `[[1, 2], [3, 4]]`.

Notably, `b, c = split_at(a, i)` for an array a of length n could already be encoded as an n-way split into single-element capabilities followed by two separate merges of the first i capabilities into b and the remaining capabilities into c. This is good news for us as it means that the current type system of Arr-O-Matic can express "split at."

Split By Similarly as above, we realised the need to cater to a special case of the strided split that takes an extra argument to set the number of elements moved to each subcapability at a time: the **`split_by`** operation used in § 5.2. Splitting an `ArrayView [1, 2, 3, 4, 5, 6, 7, 8]` in 2 parts in a strided way with the number of elements set to 2 would for example return `[[1, 2, 5, 6], [3, 4, 7, 8]]`.

6.1 Uses of Split and Merge

Both splits (consecutively and strides) are used in many of our example algorithms. Strided splits are especially common when the `ArrayView` is used for matrix implementation as this maps perfectly onto how matrices are encoded in arrays.

Both consecutive merge and strided merge were used when implementing the example algorithms.

Our preliminary investigation shows that merging is less commonly used than splitting if splitting a borrowed capability also produces borrowed capabilities. On the other hand, merging is pretty common in the implementation of convenience methods such as **`split_at`** and **`split_by`**.

Borrowing simplifies the implementation of algorithms that deconstruct and thus *destroy* the original array to make this behaviour explicit and allow the type system to prove absence of data-races. If such an operation starts with borrowing, there is no need to reconstruct the original array—we can simply lose all the borrowed pointers to all deconstructed parts and resurrect the original array.

6.2 Restrictions to Use of Align

The reason why the alignment operation is not supported on "partial capabilities" is due to its interaction with other array capabilities whose translations must be updated. This both introduces a need for synchronisation and a possible need for sibling array capabilities to know of each others' existence. These problems can be avoided by having align create a new underlying array, but this is also a simple operation that can easily be constructed by a programmer using existing building blocks. We have also not yet found a need for such partial alignment in real world examples.

6.3 Reducing Direct Index Access

The number of direct element accesses by index can often be reduced primarily since all `ArrayViews` are independent, also when created by *e.g.*,a split. They are not just a part of another array delimited by a start and end index. This is also the case when using slices in *e.g.*,Python. One case where this decreases the number of index accesses to none is in the matrix rotation examples in Section 5.1. Notably, `ArrayViews` achieves this both without copying (as is the case when using Python slices) and allowing more than one actively used partial `ArrayView` from the same underlying array at the same time.

Reducing index access increases the level of abstraction of array programming, and we suspect that it makes implementations less susceptible to the typical off-by-one errors.

6.4 Relation to the Arr-O-Matic Type System

The Python prototype naturally lacks the static type system used to construct the array capabilities introduced in Arr-O-Matic [1]. We have however implemented the type system rules as dynamic checks. This means that any errors will manifest themselves at runtime instead of at compile time as will be the case in the intended production implementation. This strategy immediately captures implementations which would break the static reasoning component of Arr-O-Matic but without guiding our thinking during the implementation. We consider this a strength of our methodology.

6.5 Conclusion

Through our prototype implementation we were able to do a preliminary evaluation on the functional API of the capability design. By encoding the static checks as dynamic checks, we are confident that we do not break the static checking of the capabilities. So far, we can conclude that:

- Two new splitting operations ("split at" and "split by") were identified and added to the API. These are convenience operators that can be implemented by combining normal split and merge and thus do not break the static type system guarantees.
- Splitting consecutively and in strides are used in many our example algorithms validating inclusion of both.
- Both consecutive merge and strided merge were used when implementing the example algorithms validating their inclusion.
- The importance of the merge operation is decreased when borrowing is applied to the splitting operation.
- The dependence on direct index based accesses decreased in general in our implementations.

The next step in our implementation work will be to integrate our results so far in Encore and to design a better index translation mechanism that focuses on performance.

References

[1] Beatrice Åkerblom, Elias Castegren, and Tobias Wrigstad. 2019. Reference Capabilities for Safe Parallel Array Programming. *Programming* (2019). To appear.

[2] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing* 37, 1 (1996), 55–69.

[3] John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience* 31, 6 (2001), 533–553. https://doi.org/10.1002/spe.370

[4] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing. In *Proceedings of the 15th European Conference on Object-Oriented Programming ECOOP, Budapest, Hungary, June 18-22, 2001*. Springer Berlin Heidelberg, 2–27. https://doi.org/10.1007/3-540-45337-7_2

[5] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy*. 1–56. https://doi.org/10.1007/978-3-319-18941-3_1

[6] Elias Castegren. 2018. *Capability-Based Type Systems for Concurrency Control*. Ph.D. Dissertation. Uppsala UniversityUppsala University, Division of Computing Science, Computing Science.

[7] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *Proceedings of the 30th European Conference on Object-Oriented Programming, ECOOP, July 18-22, 2016, Rome, Italy*. 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.5

[8] Mark Harris. 2007. Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology. http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[9] Beatrice Åkerblom. 2019. Source Code for the Python implementation used in this paper. https://github.com/batris/Array-Capabilities.