An Array Abstraction to Amortize Reasoning About Parallel Client Code

Alan Weide[1], Paolo A. G. Sivilotti[1], and Murali Sitaraman[2]

[1] The Ohio State University, Columbus OH 43221, USA
weide.3@osu.edu, paolo@cse.ohio-state.edu
[2] Clemson University, Clemson SC 29634, USA
murali@clemson.edu

Abstract. Data abstraction is important for enabling the automated modular verification of a large class of parallel programs even in the presence of manifest sharing during parallel execution. Though sharing is difficult to avoid when parallel execution is used to gain performance improvements, formal verification of such code still must be modular. The `Splittable_Array` abstraction introduced in this paper supports modular verification by allowing frame conditions to be dispatched only once and then be reused by multiple clients, thus amortizing the verification cost. The approach achieves this objective by introducing a non-interference contract that enables the preservation of desirable performance characteristics of traditional arrays such as constant-time access to elements. Illustrative divide-and-conquer client code using this abstraction is contrasted with similar client code that uses a traditional array. The utility of the `Splittable_Array` in a language with clean semantics is demonstrated by comparing the two clients in terms of the feasibility and tractability of modular verification. The repeated verification of software is expensive, so it should be avoided. Modularization (such as by introducing additional data abstractions) makes it possible to avoid expensive re-verification of entire client programs when only minor changes are made.

1 Introduction

Divide-and-conquer algorithms are readily adaptable to parallelization with the goal of performance improvements. Unfortunately, traditional implementations of these kinds of algorithms often pose difficulties for modular verification systems. Small modifications to these programs (e.g., dividing a data structure into four parts instead of two) can result in having to re-prove the program from scratch, even though the new proof obligations are mostly identical to the previous ones. In addition, potential aliasing can create data sharing that is a substantial hurdle to modular proofs of correctness for concurrent programs in general.

We propose a novel array abstraction, the `Splittable_Array`, to address these challenges. This abstraction (i) ensures *by construction* that all parts of the array that may be accessed are separate from one another, and (ii) leverages *clean semantics* to ensure that no dangerous data sharing can occur. This array abstraction simplifies proofs of non-interference between parallel threads, often turning them into simple syntactic checks that do not require reasoning about the values of objects. We also show how this abstraction can be implemented using a traditional array in order to maintain desirable performance characteristics of such data structures: constant-time split, combine, and lookup.

1.1 Clean Semantics and Shared Data in the Presence of Parallelism

One of the core principles of the RESOLVE programming language [1, 2] is the notion of *clean semantics* [3, 4]. When reasoning with clean semantics, a programmer can rely on the fact that two different variables may be treated as two independent entities—in effect, there is no dangerous aliasing in RESOLVE. The verification of parallel programs in this paper relies heavily on this idea and leverages it to dramatically simplify reasoning, maintain modularity, and guide the design of a new data abstraction.

Although data sharing can normally be avoided by careful language design in sequential programs [2], in parallel programs the sharing of data among threads is sometimes necessary to maximize performance. Showing determinism (thereby enabling functional verification) in such programs might require exposing some implementation details to the client about how the data is used. It is critical for tractability of verification that the exposure of these details does not break modularity and that it preserves abstraction so as not to complicate reasoning about the independence of concurrent operations.

Clean semantics and other RESOLVE design principles can be leveraged to eliminate the dangerous sharing of data. In this paper, we use these principles and the style of component design they encourage to demonstrate how thoughtful design can simplify the verification of a divide-and-conquer algorithm without compromising the performance benefits of a more traditional approach.

1.2 The Interference Contract and Partitions

The *interference contract* specification construct [5] divides the representation space of a data type into a number of *partitions*, each of which is disjoint from the others in the sense that at the representation level, each unit of data (e.g., each representation field[3]) is a member of exactly one partition. An interference contract extends behavioral specifications to include effects summaries, which define how an operation will interact with the partitions of an object. These summaries are described in terms of three partition modes: *affects*, *preserves*, and *oblivious to*. In addition to these three modes, a partition is said to have a *restructures* effect [6] if, by executing the operation, the members of that partition might be moved to a different one, or that partition might have another

[3] Because interference contracts are modular, representation fields might themselves have partitions that are members of a partition "one level up".

partition's members moved into it, even though *value* of each of those members will not have changed. All three partition modes and the *restructures* effect may be conditional on the abstract values of the parameters. When an interference contract is used in the verification of a **cobegin** block, if the constituent statements are *non-interfering*, then verification can proceed as if the execution was sequential [7].

2 A Motivating Example

Listing 1 shows a generic recursive, parallel divide-and-conquer method written in a Java-like language that "does something" with each entry in `A`. The concurrent calls to `divConquer` share two arguments: `A` and `mid`. This data-sharing between concurrent threads of execution is *safe* (i.e., non-interfering) only when `A` is written by different threads in compatible (non-overlapping) ways and `mid` (in this case, a primitive variable) is copied in each call. If either of these criteria are not met (e.g., there is aliasing within `A`), then no guarantees of non-interference can be made and functional verification is complicated substantially.

Listing 1. A generic recursive, parallel divide-and-conquer solution using a Java-like language.

```
/*
 * Does something with each entry in A
 * in the interval [lowEnough, tooHigh)
 */
void divConquer(T[] A, int lowEnough, int tooHigh) {
    if (tooHigh - lowEnough > 1) {
        int mid = (lowEnough + tooHigh) / 2;
        cobegin {
            divConquer(A, lowEnough, mid);
            divConquer(A, mid, tooHigh);
        }
    } else if (tooHigh > lowEnough) {
        T e = A[lowEnough];
        A[lowEnough] = doSomething(e);
    }
}
```

2.1 Verification Challenges with this Approach

The approach in Listing 1 presents several challenges to verification, and especially to modular verification. We address some of these challenges in order of increasing complexity of reasoning: first we consider the case where `A` is an array of primitives or immutable objects, then we consider when `A` is an array of mutable objects (specifically, an array of stacks), and finally we consider when `A` is an array of objects using a shared representation.

The Overlapping Array Intervals Problem First, we consider the simplest case where `T` is a primitive or immutable reference type. In order to verify the functional correctness of this method body relative to a formalized version of its specification, it is helpful to show that the parallel portion of the code does not introduce any nondeterminism[4] through data races. For this, it is sufficient to show that the two recursive calls are non-interfering (in the sense presented in [8]). Showing non-interference, especially when each element of the array might be modified, requires showing that the intervals $[lowEnough, mid)$ and $[mid, tooHigh)$ are disjoint. This is not a hard problem in this simple case, but suppose for performance reasons that a programmer wished to modify this program to split `A` into 4 parts: $[lowEnough, q_1)$, $[q_1, mid)$, $[mid, q_3)$, and $[q_3, tooHigh)$. Now there are four intervals which must be shown to be pairwise disjoint. In the general case, showing mutual disjointness of n sets of indices is non-trivial, and it is certainly not readily scalable (the number of pairs increases quadratically with n). The explicit split/combine operations in the `Splittable_Array` abstraction discussed in Section 3 are motivated by this problem.

A related problem occurs when the partitioning is not into contiguous segments of the array, for example a partitioning of `A` into the even indices and odd indices. When partitions are arbitrary, the disjointness problem becomes much harder, and potentially even intractable. This problem motivates a more general array abstraction discussed in Section 4.

Challenges Related to Aliasing Next we identify challenges posed by a similar program, but where `T` is a mutable reference type. Listing 2 shows an example where the array contains stacks. When reasoning about the code in `mutateTops`, a requirement for the non-interference of the parallel section of code is the total independence of each stack in `A`. In particular, a specification of this method written in separation logic might look similar to the specification in (1).

$$\begin{gathered}\left\{\bigodot_{i=l}^{h-1} \mathsf{list}\ \mathsf{e^i} \cdot \alpha^i\,(A_0[i], \mathsf{nil})\right\} \\ \text{mutateTops(A, l, h);} \\ \left\{\bigodot_{i=l}^{h-1} \mathsf{list}\ \mathsf{e^{i'}} \cdot \alpha^i\,(A_0[i], \mathsf{nil})\right\}\end{gathered} \tag{1}$$

[4] It is possible for a program to exhibit nondeterminism and still be correct, but for now we are concerned only with provably deterministic parallel programs.

Listing 2. A recursive, parallel divide-and-conquer solution using a Java-like language and an array of Stacks.

```
/*
 * Mutates the top of each stack in A
 * in the interval [lowEnough, tooHigh)
 */
void mutateTops(Stack<R>[] A, int lowEnough, int tooHigh) {
    if (tooHigh − lowEnough > 1) {
        int mid = (lowEnough + tooHigh) / 2;
        cobegin {
            mutateTops(A, lowEnough, mid);
            mutateTops(A, mid, tooHigh);
        }
    } else if (tooHigh > lowEnough) {
        if (A[lowEnough].size() > 0) {
            R e = A[lowEnough].pop();
            mutate(e);
            A[lowEnough].push(e);
        }
    }
}
```

In plain English, the meaning of this specification is as follows. The precondition, $\left\{\bigodot_{i=l}^{h-1} \mathsf{list}\ \mathsf{e}^{\mathsf{i}} \cdot \alpha^i\,(A_0[i], \mathsf{nil})\right\}$, states that for each $l \leqslant i < h$, the initial value of the i-th element of array A is a nil-terminated (singly linked) list with abstract value $\mathsf{e}^{\mathsf{i}} \cdot \alpha^i$, and that each of these lists is separate in the heap (that is, they share no nodes). The postcondition states that the only thing that has changed about each element of A is the abstract value of the first node in the list (and, still, that the lists are separate).

The first challenge posed by this example with this specification is that it does not preclude aliasing between *elements* of the stacks in `A`. For example, if the top element of `A[0]` is an alias to the top element of `A[1]`, the picture might look like Figure 1. Note that it is still true that $\{\mathsf{list}\ \alpha\,(A[0], \mathsf{nil}) * \mathsf{list}\ \beta\,(A[1], \mathsf{nil})\}$, so the precondition is satisfied. The problem with this scenario is that if the mutation performed by the `mutate` operation is not idempotent, the result will not be correct (in fact, if the top of one stack is an alias to an element of a stack that is not the top, even an idempotent mutation will cause problems). In a language with clean semantics, we can rely on the fact that separate variables act as separate objects to guarantee there is no dangerous aliasing.[5]

[5] While a modified specification could be written in separation logic to handle this particular situation, it becomes extremely complex in the general case [9].

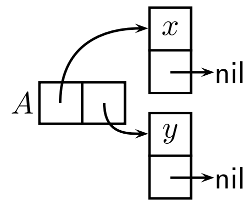

Fig. 1. An array with separate (in the heap) lists, but whose member stacks have elements that are aliases if x and y are references and $x = y$.

The Shared Representation Problem A much more subtle issue arises when instances of `T` use a shared representation. For example, the precondition as written would not be satisfied if the stack implementation were swapped out for one based on a shared cactus stack, as in Figure 2, even though such an implementation could provide correct behavior. This lack of modularity demonstrates a need for abstraction in the specification of concurrent programs (e.g., an interference contract) to facilitate reusable code that can remain proven to be correct even when different underlying data structures are used.

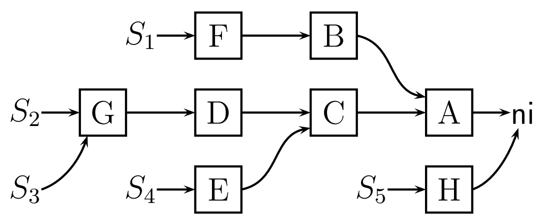

Fig. 2. A cactus stack using a partially-shared realization. Stack $S_1 = \langle F, B, A\rangle$, $S_2 = S_3 = \langle G, D, C, A\rangle$, $S_4 = \langle E, C, A\rangle$, and $S_5 = \langle H\rangle$. Note that it is *not* the case that $\mathsf{list}\ \alpha(S_1, \mathsf{nil}) * \mathsf{list}\ \beta(S_2, \mathsf{nil})$ for any α, β.

3 The Splittable_Array Abstraction

The `Splittable_Array` abstraction is a novel array abstraction that amortizes the cost of reasoning about parallel divide-and-conquer algorithms such as the one presented in Listing 2. The `Splittable_Array` component provides the client with operations that divide the array at some split point into two sub-arrays with contiguous indices. By virtue of RESOLVE's clean semantics the resulting sub-arrays may be reasoned about as totally independent objects. The abstract specification of this component is shown in Listing 3.

Notation The notation used in the remaining listings of this paper is based on the RESOLVE language [1, 2]. A *concept* is the specification of a type, including a mathematical model and specifications for operations on that type in terms of that model. In the operation contracts, the keywords **restores**, **updates**, **replaces**, **clears**, and **preserves** are *parameter modes*, which summarize the effect of the operation on that parameter. The difference between **restores** and **preserves** is that a **restores**-mode parameter might have its value changed temporarily by the implementation of the operation while a **preserves**-mode parameter may not; both express the fact that the value of the parameter after the operation is the same as it was beforehand. In the **ensures** clause of the operation contracts, the notation # denotes the "old" value of a parameter—roughly equivalent to a zero-subscript in other specification languages. In the specifications hereafter, we use traditional mathematical notation for the clauses to improve readability; the language has text-based equivalents that would appear in real programs.

Listing 3. Abstract specification for `Splittable_Array`

```
concept Splittable_Array_Template(type Entry)

  var Ids: finite set of integer
    initialization ensures Ids = ∅

  type family Splittable_Array is modeled by (
    Id: integer
    Lower_Bound: integer,
    Upper_Bound: integer,
    Contents: integer −> Entry,
    Split_Point: integer,
    Parts_In_Use: boolean
  )
  exemplar A
    constraint
      A.Lower_Bound ⩽ A.Upper_Bound ∧
      A.Lower_Bound ⩽ A.Split_Point ⩽ A.Upper_Bound ∧
      A.Id ∈ Ids
    initialization ensures
      A.Lower_Bound = 0 ∧ A.Upper_Bound = 0 ∧
      ¬A.Parts_In_Use ∧
      A.Id ∉ #Ids
  end

  operation Set_Bounds(
      restores LB: integer,
      restores UB: integer,
      updates A: Splittable_Array)
    requires LB ⩽ UB ∧ ¬A.Parts_In_Use
    ensures A.Lower_Bound = LB ∧ A.Upper_Bound = UB ∧
      ¬A.Parts_In_Use ∧
```

```
    A.Id ∉ #Ids ∧ #A.Id ∉ Ids

operation Set_Split_Point(
    restores i: integer,
    updates A: Splittable_Array)
  requires A.Lower_Bound ⩽ i ∧ i ⩽ A.Upper_Bound ∧
    ¬A.Parts_In_Use
  ensures A.Split_Point = i ∧
    [[everything else about A stays the same]]

operation Swap_Entry_At(
    restores i: integer,
    updates A: Splittable_Array,
    updates E: Entry)
  requires ¬A.Parts_In_Use ∧
    A.Lower_Bound ⩽ i < A.Upper_Bound
  ensures E = #A.Contents(i) ∧ A.Contents(i) = #E ∧
    [[everything else about A stays the same]]

operation Split(
    updates A: Splittable_Array,
    replaces L: Splittable_Array,
    replaces U: Splittable_Array)
  requires ¬A.Parts_In_Use
  ensures A.Parts_In_Use ∧
    L.Incl_Lower_Bound = A.Incl_Lower_Bound ∧
    L.Excl_Upper_Bound = A.Split_Point ∧
    U.Lower_Bound = A.Split_Point ∧
    U.Upper_Bound = A.Excl_Upper_Bound ∧
    L.Id = A.Id ∧ L.Contents = A.Contents ∧
    U.Id = A.Id ∧ U.Contents = A.Contents ∧
    ¬L.Parts_In_Use ∧ ¬U.Parts_In_Use ∧
    [[everything else about A stays the same]]

operation Combine(
    updates A: Splittable_Array,
    clears L: Splittable_Array,
    clears U: Splittable_Array)
  requires A.Parts_In_Use ∧
    ¬L.Parts_In_Use ∧ ¬U.Parts_In_Use ∧
    L.Incl_Lower_Bound = A.Incl_Lower_Bound ∧
    L.Excl_Upper_Bound = A.Split_Point ∧
    U.Incl_Lower_Bound = A.Split_Point ∧
    U.Excl_Upper_Bound = A.Excl_Upper_Bound ∧
    L.Id = A.Id ∧ U.Id = A.Id
```

```
    ensures ¬A.Parts_In_Use ∧
      ∀(i : ℤ)(
        (i < A.Split_Point) ⇒ (A.Contents(i) = #L.Contents(i)) ∧
        (i ⩾ A.Split_Point) ⇒ (A.Contents(i) = #U.Contents(i))) ∧
      [[everything else about A stays the same]]

  operation Lower_Bound(preserves A: Splittable_Array): integer
    ensures Lower_Bound = A.Lower_Bound

  operation Upper_Bound(preserves A: Splittable_Array): integer
    ensures Upper_Bound = A.Upper_Bound

  operation Split_Point(preserves A: Splittable_Array): integer
    ensures Split_Point = A.Split_Point

  operation Parts_Are_In_Use(preserves A: Splittable_Array):
    Boolean
    ensures Parts_Are_In_Use = A.Parts_In_Use

  operation Ids_Match(
      preserves A1: Splittable_Array,
      preserves A2: Splittable_Array): Boolean
    ensures Ids_Match = (A1.Id = A2.Id)

end Splittable_Array_Template
```

The `Splittable_Array` component is modeled in part as a function from integers to entries (`Contents`). The `Incl_Lower_Bound` and `Excl_Upper_Bound` give the addressable range of indices. The operations allow the client to set a split point within the addressable range, and to split/combine the array into two subarrays, split at `A.Split_Point`. The `Parts_In_Use` flag indicates whether the array has been split into subparts, and controls access to those parts.

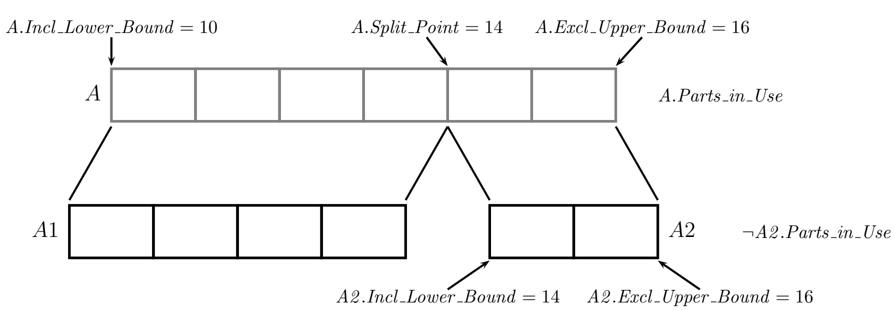

Fig. 3. The state after `Split(A, A1, A2)` is executed on a `Splittable_Array`.

Figure 3 visualizes how a splittable array is split up by the `Split` operation. Given three splittable arrays `A`, `A1`, and `A2`, the operation call `Split(A, A1, A2)` places the contents of `A` below the split point into `A1` and the contents of `A` at and above the split point in `A2`. After `A` is split, a client cannot access anything in `A` until `A1` and `A2` are combined back into `A`; that is, until $A.Parts\_In\_Use = \mathsf{true}$.

3.1 Divide-and-Conquer Client Code Using `Splittable_Array`

A natural application of `Splittable_Array` is in a parallel divide-and-conquer algorithm such as the `mutateTops` operation from Listing 2. Using the component in such a context dramatically simplifies the reasoning involved in formally verifying the correctness of such code. Listing 4 shows how such an algorithm might be implemented and specified using `Splittable_Array`.

Listing 4. A recursive, parallel divide-and-conquer solution using `Splittable_Array`

```
uses Stack with [[some interference contract]];
uses Splittable_Array with [[some interference contract]];
uses Entry with [[some interference contract]];
facility Stack_Fac is Stack(Entry);
facility Array_Fac is Splittable_Array(Stack_Fac.Stack);
operation Mutate_Tops(updates A: Splittable_Array);
  requires
    A.Incl_Lower_Bound < A.Excl_Upper_Bound and
    not A.Parts_In_Use;
  ensures [[the top of each stack in A has been mutated]];
  interference spec
    affects A@*;
recursive procedure
  decreasing A.Excl_Upper_Bound - A.Incl_Lower_Bound;

  if (Excl_Upper_Bound(A) - Incl_Lower_Bound(A) > 1) then
    var A1, A2: Splittable_Array
    var mid: Integer := (Incl_Lower_Bound(A) + Excl_Upper_Bound(A))
     / 2
    Set_Split_Point(mid, A)
    Split(A, A1, A2)
    cobegin
      Mutate_Tops(A1)
      Mutate_Tops(A2)
    end;
    Combine(A, A1, A2)
  else
    var stack: Stack
    var index: Integer := Incl_Lower_Bound(A)
    Swap_Entry_At(A, index, stack)
    if Length(stack) > 0 then
```

```
      var e: Entry
      Pop(stack, e)
      Mutate(e)
      Push(stack, e)
    end
    Swap_Entry_At(A, index, stack)
  end
end Mutate_Tops
```

As discussed in Section 2, keeping verification of this code relatively simple involves showing that the operations inside the `cobegin` block are *non-interfering* as defined in [8]. First, by the *interference spec* (a local, operation-level version of the component-level *interference contract*) of `Mutate_Tops`, we know that it *affects* all partitions of `A` (whatever those partitions may be—this particular interference spec is agnostic to the interference contract used with the array). If there were a shared array parameter between the two recursive calls, it would be impossible to show non-interference. Fortunately, however, the two calls to `Mutate_Tops` inside the `cobegin` statement operate on different array variables, so they are necessarily independent because of RESOLVE's clean semantics.

3.2 Reusability and Modifiability

The code in Listing 4 is highly reusable. A client can use any implementation of the Stack concept as long as it respects the interference contract defined in the solution. For example, if there is a cactus stack realization of Stack that has appropriate concurrency properties, the client may use it without having to reprove `Mutate_Tops` or write a new specification. In fact, this particular operation is entirely agnostic to any of the interference contracts that may be supplied.

It is also highly amenable to modifications, requiring only minimal proofs to be discharged in most cases. Consider an alternate approach to `Mutate_Tops` where the array is split into four parts instead of two. Now the parallel section of the code might look like Listing 5. Thanks to clean semantics, it is still a simple syntactic check to show that the four parallel calls are non-interfering. The one-time proof of disjointness in the intervals falls on the implementer of the `Splittable_Array` specification—but is trivial unless the implementer opts for a shared realization such one discussed in Section 3.3.

Listing 5. The parallel section of a divide-and-conquer solution which splits `A` into 4 parts via consecutive calls to `Split`.

```
cobegin
  Mutate_Tops(A1)
  Mutate_Tops(A2)
  Mutate_Tops(A3)
  Mutate_Tops(A4)
end
```

3.3 Efficiently Realizing `Splittable_Array`

A combination of clean semantics, careful component design, and robust specification has reduced the potentially complicated reasoning problem of showing non-interference in Listing 2 to a purely syntactic check in Listing 4. This is a clear advantage of our approach over the traditional one. Importantly, these reasoning advantages can be achieved *without* compromising performance.

Although clean semantics allows the two sub-arrays `A1` and `A2` to be reasoned about as if they were totally separate arrays, an efficient implementation of this concept would *not* make any copies of the array. The interface for this component was designed with a shared implementation in mind so that a realization could employ an underlying (traditional) array that is shared among all `Splittable_Array` instances with the same `Id`. This design choice manifests itself in the use of the `Split` and `Combine` operations as pseudo-synchronization points by flipping `Parts_In_Use` and preventing access to the array while it is split. Doing so ensures that at any time, there is only one array with each `Id` that can access any given index. In this way, a realization can share an underlying array among instances with the same `Id` without introducing any interference. Enabling such a shared implementation is important for preserving the performance benefits that programmers expect from parallel software; that is, the operations `Split`, `Combine`, and `Swap_Entry_At` can all be done in constant time.

4 A Hierarchy of Array Abstractions

4.1 Three Distinct Array Abstractions

The `Splittable_Array` abstraction presented here is one member of a hierarchy of concurrency-ready array abstractions that can be used in multiple contexts [6]. The most general abstraction in this family, `Index_Partitionable_Array`, may be partitioned on arbitrary indices rather than contiguous portions of the array. A third abstraction, `Distinguished_Index_Array`, allows a client to isolate a *distinguished entry* and operate on it separately from the rest of the array.

4.2 Layered Implementations

The `Index_Partitionable_Array` (and `Distinguished_Index_Array`) can be efficiently implemented in a similar manner to `Splittable_Array`; that is, by sharing a single underlying array amongst all instances with the same ID. Once an efficient realization for `Index_Partitionable_Array` is provided, however, more specialized realizations can be built by layering on top of it. For example, `Splittable_Array` can be realized with an underlying `Index_Partitionable_Array` and mapping the two sets of indices of the `Index_Partitionable_Array` to the *low* and *high* parts of the `Splittable_Array` and maintaining the invariant that each set of indices in the underlying `Index_Partitionable_Array` is contiguous (and corresponds to the appropriate indices for abstraction to a `Splittable_Array`).

5 Related Work

One central challenge in sequential, object-oriented software verification involves objects, aliasing, and properties about the heap [10, 11]. Separation logic is an extension of Hoare's logical rules to address these challenges. Examples of verification using separation logic in Coq include [12, 13] and in VeriFast to verify Java and C programs include [14, 15]. Automating verification with separation logic is the topic of [16–20]. Concurrent separation logics have attempted to expand the capabilities of separation logic by adding rules for reasoning about concurrent programs [21]. An important new direction for abstraction and simplification in concurrent separation logic is the focus of [22]. Our approach sidesteps the concerns of both separation logic and concurrent separation logic first by using a language which has clean semantics and then by abstracting away most implementation details to avoid reasoning directly about the state of the heap.

Other approaches have guaranteed determinism in the presence of parallelism using region logic or a variant thereof. In Deterministic Parallel Java [23] (DPJ), regions are defined explicitly by the programmer. These annotations allow a DPJ compiler to guarantee, syntactically, that two concurrent operations are non-interfering and thus will produce a deterministic result. ParaSail [24] is an extension of the Ada programming language that relies on value semantics to verify that concurrent statements are non-interfering. Both approaches are limited by what can be syntactically checked, and ParaSail in particular is limited by the fact that objects must be reasoned about as a whole and cannot be subdivided. By leveraging the full functional verification capabilities of RESOLVE, our approach can increase the expressiveness of method effects over that of DPJ by including conditional effects, and clean semantics let us preserve the simplicity afforded by ParaSail. Both DPJ and ParaSail offer examples of divide-and-conquer solutions similar to the one presented here and have verified them to be deterministic (their correctness is informally argued) [25, 26].

DPJ additionally provides the `DPJArray` and `DPJPartition` [27, 28] families of classes to attack many of the same problems as the various partitionable arrays presented in this paper. A `DPJArray` allows the client to define *subarrays*, and operate on them as if they were their own array—without making any copies (shallow or otherwise). Because there are no disjointness requirements placed upon subarrays, the implementer of a divide-and-conquer algorithm in DPJ should prefer to use `DPJPartition` which splits an array into two disjoint, contiguous sections based upon a client-provided index. However, there is nothing to stop a client from accessing elements of a partitioned array while a parallel thread is accessing either of its sub-parts, potentially compromising determinism. In contrast, there is always exactly one instance of a `Splittable_Array` that can access any given element.

Other languages that provide array slicing or partition operations typically make shallow copies of the underlying array [29, 30]. This poses two immediate problems. First, it does not eliminate the potential for aliasing between elements of several arrays that have been sliced. Second, the operation has a runtime that is linear in the length of the slice.

Determinism guarantees in our research and others' amount to showing data race freedom, though our approach deals with high-level programming constructs. There is a large body of work on showing low-level race freedom, including both static approaches [31–36] and dynamic ones [37–40]. Like DPJ and ParaSail, these are limited to guaranteeing determinism and do not claim to formally verify full functional correctness.

6 Conclusions

Verifying the correctness of a parallel divide-and-conquer algorithm in a modular way using traditional arrays is a difficult problem for automated verification engines. To solve this problem, we develop a new abstraction, the `Splittable_Array`, with an interference contract that abstractly defines how data can be accessed. Combined with careful component design and clean semantics, this approach allows us to write software that is easy to reason about even in the presence of concurrency.

The `Splittable_Array` allows reusable verification of non-interference of array partitions in one place instead of for every client. The abstraction allows frame conditions to be captured in a novel way that reduces repeated verification costs, as demonstrated through the divide-and-conquer examples in this paper.

The development of this new data abstraction allows standard client verification machinery to be used, such as tools developed for the verification of RESOLVE programs [41, 42].

Acknowledgments

We thank the members of our research groups at Clemson, Ohio State, and other institutions who have contributed to the discussions on topics contained in this paper. This research is funded in part by US NSF grant DUE-1022941. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.

References

1. M. Sitaraman, B. Adcock, J. Avigad, D. Bronish, P. Bucci, D. Frazier, H. M. Friedman, H. Harton, W. Heym, J. Kirschenbaum, J. Krone, H. Smith, and B. W. Weide, "Building a push-button RESOLVE verifier: Progress and challenges," *Formal Aspects of Computing*, vol. 23, no. 5, pp. 607–626, 2011.
2. D. E. Harms and B. W. Weide, "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 424–435, may 1991.
3. G. Kulczycki, *Direct Reasoning.* Phd dissertation, Clemson University, School of Computing, 2004.

4. G. Kulczycki, M. Sitaraman, J. Krone, J. E. Hollingsworth, W. F. Ogden, B. W. Weide, P. Bucci, C. T. Cook, S. Drachova-Strang, B. Durkee, H. K. Harton, W. D. Heym, D. Hoffman, H. Smith, Y.-S. Sun, A. Tagore, N. Yasmin, and D. Zaccai, "A Language for Building Verified Software Components," in *Safe and Secure Software Reuse - 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Proceedings*, pp. 308–314, 2013.
5. A. Weide, P. A. G. Sivilotti, and M. Sitaraman, "Enabling Modular Verification with Abstract Interference Specifications for a Concurrent Queue," in *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers* (S. Blazy and M. Chechik, eds.), vol. 9971 of *Lecture Notes in Computer Science*, pp. 119–128, 2016.
6. A. Weide, P. A. G. Sivilotti, and M. Sitaraman, "Array Abstractions to Simplify Reasoning About Concurrent Client Code," Tech. Rep. RSRG-17-05, Clemson University - School of Computing, Clemson, SC 29634, nov 2017.
7. A. Weide, P. A. Sivilotti, and M. Sitaraman. Toward a Modular Proof Rule for Parallel Operation Calls. Tech. Report OSU-CISRC-3/20-TR01, The Ohio State University, Columbus, OH, March 2020.
8. A. Weide, P. A. G. Sivilotti, and M. Sitaraman, "Enabling Modular Verication of Concurrent Programs with Abstract Interference Contracts," Tech. Rep. RSRG-16-05, Clemson University - School of Computing, Dec 2016.
9. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer, "Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques," in *Software Engineering and Formal Methods* (G. Barthe, A. Pardo, and G. Schneider, eds.), (Berlin, Heidelberg), pp. 382–398, Springer Berlin Heidelberg, 2011.
10. G. T. Leavens, K. R. M. Leino, and P. Müller, "Specification and Verification Challenges for Sequential Object-oriented Programs," *Form. Asp. Comput.*, vol. 19, pp. 159–189, jun 2007.
11. J. C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, (Washington, DC, USA), pp. 55–74, IEEE Computer Society, 2002.
12. A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky, "Effective Interactive Proofs for Higher-order Imperative Programs," in *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, (New York, NY, USA), pp. 79–90, ACM, 2009.
13. H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using Crash Hoare Logic for Certifying the FSCQ File System," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 18–37, ACM, 2015.
14. B. Jacobs, J. Smans, and F. Piessens, "Verifying the composite pattern using separation logic," in *Proceedings of the 7th International Workshop on Specification and Verification of Component-Based Systems*, SAVCBS'08, pp. 83–88, 2008.
15. B. Jacobs, J. Smans, and F. Piessens, "A quick tour of the VeriFast program verifier," in *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, (Berlin, Heidelberg), pp. 304–311, Springer-Verlag, 2010.
16. F. Bobot and J.-C. Filliâtre, "Separation Predicates: A Taste of Separation Logic in First-Order Logic," in *Formal Methods and Software Engineering* (T. Aoki and K. Taguchi, eds.), vol. 7635 of *Lecture Notes in Computer Science*, pp. 167–181, Springer Berlin Heidelberg, 2012.

17. M. Botinčan, M. Parkinson, and W. Schulte, "Separation Logic Verification of C Programs with an SMT Solver," *Electron. Notes Theor. Comput. Sci.*, vol. 254, pp. 5–23, oct 2009.
18. R. Piskac, T. Wies, and D. Zufferey, "Automating Separation Logic Using SMT," in *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, (Berlin, Heidelberg), pp. 773–789, Springer-Verlag, 2013.
19. R. Piskac, T. Wies, and D. Zufferey, "Automating Separation Logic with Trees and Data," in *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, (New York, NY, USA), pp. 711–728, Springer-Verlag New York, Inc., 2014.
20. C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard, "Using First-Order Theorem Provers in the Jahob Data Structure Verification System," in *Verification, Model Checking, and Abstract Interpretation* (B. Cook and A. Podelski, eds.), vol. 4349 of *Lecture Notes in Computer Science*, pp. 74–88, Springer Berlin / Heidelberg, 2007.
21. S. Brookes, "A Semantics for Concurrent Separation Logic," in *CONCUR 2004 - Concurrency Theory* (P. Gardner and N. Yoshida, eds.), (Berlin, Heidelberg), pp. 16–34, Springer Berlin Heidelberg, 2004.
22. R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal, "The Essence of Higher-Order Concurrent Separation Logic," in *Programming Languages and Systems* (H. Yang, ed.), (Berlin, Heidelberg), pp. 696–723, Springer Berlin Heidelberg, 2017.
23. R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A Type and Effect System for Deterministic Parallel Java," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, (New York, NY, USA), pp. 97–116, ACM, 2009.
24. T. Taft and J. Hendrick, "Designing ParaSail, a new programming language." http://parasail-programming-language.blogspot.com/, 2017.
25. "The Deterministic Parallel Java Tutorial, Version 1.0." http://dpj.cs.illinois.edu/DPJ/Download_files/DPJTutorial.html, 2010.
26. T. Taft, "Tutorial: Multicore Programming using Divide-and-Conquer and Work Stealing." https://drive.google.com/file/d/0B6Vq5QaY4U7uUktTcUpySmVjaW8/edit, 2013.
27. "Class DPJArray." https://dpj.cs.illinois.edu/DPJ/Download_files/DPJRuntime/DPJArray.html.
28. "Class DPJPartition." https://dpj.cs.illinois.edu/DPJ/Download_files/DPJRuntime/DPJPartition.html.
29. "Array.prototype.slice()." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice.
30. "An informal introduction to python." https://docs.python.org/3/tutorial/introduction.html#lists.
31. D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 237–252, ACM, 2003.
32. M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for Java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp. 308–319, ACM, 2006.
33. S. N. Freund and S. Qadeer, "Checking concise specifications for multithreaded software," *Journal of Object Technology*, vol. 3, no. 6, pp. 81–101, 2004.

34. C. Flanagan and S. Qadeer, "A Type and Effect System for Atomicity," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, (New York, NY, USA), pp. 338–349, ACM, 2003.
35. M. Naik and A. Aiken, "Conditional Must Not Aliasing for Static Race Detection," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, (New York, NY, USA), pp. 327–338, ACM, 2007.
36. M. Kawaguchi, P. Rondon, A. Bakst, and R. Jhala, "Deterministic Parallelism via Liquid Effects," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 45–54, ACM, 2012.
37. E. Pozniansky and A. Schuster, "MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, pp. 327–340, mar 2007.
38. E. Pozniansky and A. Schuster, "Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs," in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, (New York, NY, USA), pp. 179–190, ACM, 2003.
39. C. Flanagan and S. N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 121–133, ACM, 2009.
40. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multi-threaded Programs," in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, (New York, NY, USA), pp. 27–37, ACM, 1997.
41. "ResolveOnline." http://resolveonline.cse.ohio-state.edu/.
42. "RESOLVE Web IDE." https://resolve.cs.clemson.edu/teaching.