High-Level Abstractions for Safe Parallelism

Robert L. Bocchino Jr.
Carnegie Mellon University
rbocchin@cs.cmu.edu

Hannes Mehnert
IT University of Copenhagen
hame@itu.dk

Jonathan Aldrich
Carnegie Mellon University
jonathan.aldrich@cs.cmu.edu

Abstract

Recent research efforts have developed sophisticated type systems for eliminating unwanted interference (i.e., read-write conflicts) from parallel code. While these systems are powerful, they suffer from potential barriers to adoption in that (1) they rely upon complex and/or restrictive features that may be difficult for programmers to understand and use; and (2) they impose a nontrivial annotation burden.

In this work we explore a different approach: instead of extending the type system to do all the work of proving noninterference, we rely upon *high-level abstractions* that capture important patterns of noninterfering parallelism — for example, performing a parallel divide-and-conquer update on an array, or updating different array cells in parallel while reading memory disjoint from the array. We show how, with suitably designed APIs, a few simple type system extensions can guarantee that user code is noninterfering, assuming the APIs are correctly implemented. Of course someone still must check the API implementation; but such checking (which can be done, e.g., with program logic) is hidden from the user of the API.

To illustrate the idea, we present a prototype implementation in Standard ML, including several parallel APIs and two realistic client programs. We sketch the typing annotations and verification methodology we have in mind. We pose several research questions raised by the prototype and suggest ideas for extending the work.

1. Introduction

Single-processor size and speed have hit a scaling wall, and commodity hardware is becoming more parallel. Therefore software is becoming more parallel as well. Parallel software, however, poses significant development and maintenance challenges. One important challenge is the possibility of *data races*, which occur when two concurrent tasks access the same memory without coordination, and when at least one of the accesses is a write. Data races can result in nondeterministic computation results and subtle errors.

Researchers have proposed different ways to avoid races and/or ensure deterministic execution, typically using types and related annotations to represent effects [17, 28] or permissions [19, 26, 38, 39]. Overall, the strategy of these approaches is to use program annotations to track where *interference* (i.e., parallel read-write conflicts) may potentially occur, and ensure correct synchronization. These systems can provide impressive guarantees, for example excluding all data races at compile time [18, 39], or even assuring that the program executes deterministically [15, 17, 28].

The cost of these guarantees, however, can be high. First, the burden of understanding and writing the annotations is nontrivial. Second, the type systems can impose awkward restrictions, such as disallowing common patterns of assignment (in the case of region types, such as DPJ [16]) or aliasing (in the case of uniqueness types [26, 38, 39]). Finally, the more esoteric aspects of these systems (for example wildcard regions in DPJ, or "borrowing" rules in uniqueness-based systems) can be intimidating for programmers.

In this work we explore a different approach: instead of extending the type system to do all the work of proving noninterference, we rely upon *high-level abstractions* that capture important patterns of noninterfering parallelism — for example, performing a parallel divide-and-conquer update on an array, or updating different array cells in parallel while reading memory disjoint from the array. We show how, with suitably designed APIs, a few simple type system extensions can guarantee that user code is noninterfering, assuming the APIs are correctly implemented. Of course someone still must check the API implementation; but such checking (which can be done, e.g., with program logic) is hidden from the user of the API.

Our main insight is that *verified parallel APIs* — plus an "ordinary" user type system augmented with a few simple extensions — can do the same work as the more complex type system extensions in previous work. The primary benefit is that the user experience should be more familiar: instead of mastering a complex type system, the user just has to understand and use the API. The annotation burden should also be less: for example, there are no uniqueness or effect annotations. Finally, there are no restrictions on assignment or aliasing, other than those imposed by the minimally extended type system. Of course the programmer is restricted to using available APIs, but if another API is needed, then (assuming its design is possible) it can be easily added. Overall, our approach is similar to the work on parallel frameworks in DPJ [16], but with more robust API design and far less user-side annotation.

We illustrate our idea by describing a prototype implementation in Standard ML. We describe several parallel APIs and two realistic client programs (a merge sort and an n-body simulation), including the typing annotations and verification methodology we have in mind. Then we discuss some research questions raised by our prototype. After that we discuss related work and ideas for future work.

2. Examples

In this section we illustrate our idea with two examples from our prototype implementation, written in Standard ML (SML). ML is well suited to this work because its type and module systems are elegant and powerful for expressing higher-order functional APIs; and yet it supports imperative computations with in-place updates, e.g., using *ref* and *array* types. However, we believe this choice is not essential; for example, we have written similar examples in Scala and F# (we discuss the F# implementation in Section 3). The full source code for our examples is available on GitHub [6].

2.1 Disjoint Array Slices

Our first example is an SML module *DisjointSlices*, which supports in-place divide-and-conquer operations on collections of disjoint array slices. By "array slice" we mean a sequence of index positions into an array. By "disjoint" we mean that any two slices in the collection represent non-overlapping memory, either because they index into different arrays, or because the index ranges do not

overlap. We show how to (1) write the *DisjointSlices* module so that it supports noninterfering divide-and-conquer parallelism on arrays and (2) use the module to write parallel merge sort.

Our module uses types *Array* and *ArraySlice* from the SML Standard Basis Library. An *Array* represents a type-polymorphic array with in-place update, and an *ArraySlice* represents an array slice as described above. In particular, creating an *ArraySlice* does not copy any array data; instead the slice stores a reference to the underlying array. That way several *ArraySlice* objects can read and write the same underlying array.

```
1  signature DISJOINT_SLICES =
2  sig
3
4    (* A list of disjoint array slices *)
5    mutable type 'a slices
6
7    (* A list of lists of disjoint array slices *)
8    mutable type 'a partitions
9
10   (* Create fresh array slices from
11      (length, initial value) pairs *)
12   val slices : (int * 'a) list -> 'a slices
13
14   (* Wrap an array in a singleton slice *)
15   val fromArray : 'a Array.array -> 'a slices
16
17   (* Add fresh array slices *)
18   val add : 'a slices * (int * 'a) list ->
19     'a slices
20
21   (* S -> I -> P splits each slice in S using the
22      corresponding index list in I *)
23   val split : 'a slices -> int list list ->
24     'a partitions
25
26   (* Transpose the list of lists of slices *)
27   val transpose : 'a partitions -> 'a partitions
28
29   (* Apply function in parallel to each element *)
30   val apply : ('a slices -> unit) ->
31     'a partitions -> unit
32
33   (* Get the list representation of the slices *)
34   val getList : 'a slices ->
35     'a ArraySlice.slice list
36
37   ...
38
39 end
```

Figure 1. Signature for the *DisjointSlices* module (partial).

Module signature. Figure 1 shows selected members of the signature for our *DisjointSlices* module. We have extended the SML syntax with a keyword *mutable* (highlighted in blue bold face in the figure, and discussed below), but otherwise this is plain SML.

In Figure 1 lines 4–8, the signature defines two abstract types, *slices* and *partitions*. Type *slices* represents a list of disjoint slices. Type *partitions* represents a list of lists of disjoint slices; a value of this type results from splitting one or more of the elements of a *slices* into sub-slices, to represent sub-computations on parts of the data. The user of this API expresses a divide-and-conquer parallel algorithm as a higher-order function that (1) takes a *slices* type as an argument; (2) splits the *slices* into *partitions*; and (3) applies itself to the *partitions*.

The keyword *mutable* appearing in lines 5 and 8 specifies that types *slices* and *partitions* represent values encapsulating references to mutable state (for example, an SML reference or array, or a record or tuple transitively containing a reference or array). On the other hand, the type variable *'a* must be an immutable value. To allow binding of mutable data to *'a* we would write *mutable 'a*. We envision a simple extension to the SML type system that enforces consistency with respect to these annotations. For example, given the type shown in Figure 1 line 5, it would be a type error (1) for the implementor to omit the *mutable* before the *type* keyword in the signature and then implement the type with references or mutable arrays; or (2) for the user to bind a mutable data type to a plain *'a* with no *mutable* keyword in the signature.

Lines 10–19 illustrate functions for creating and transforming *slices* types. Function *slices* takes a list of parameters (length and initial value) and uses them to populate a *slices* with fresh slices. Function *fromArray* accepts an existing array and wraps it in a *slices*. Function *add* adds fresh slices to an existing *slices*.

Lines 21–28 illustrate functions for creating and transforming *partitions* types. Function *split* takes a *slices* S and a list I of index lists, where I and S have equal length (if not, a runtime exception occurs). It produces a *partitions* type by splitting each of the slices in S according to the corresponding index list in I. For example, inputs $[A, B]$ and $[[m], [n]]$ yield $[[A_1, A_2], [B_1, B_2]]$, where A_1 represents the first m indices of A, and A_2 represents the rest, and similarly for B_1, B_2, and n. Function *transpose* performs a standard matrix transpose on a list of lists: for example, *transpose* $[[A_1, A_2], [B_1, B_2]]$ yields $[[A_1, B_1], [A_2, B_2]]$.

Function *apply* (line 30 and following) applies a higher-order function in parallel to each element in the list of *slices* types represented by the *partitions* input. Again we extend the SML type system slightly: we assume the function type *'a slices* $\rightarrow$ *unit* guarantees that (1) *'a* is an immutable value type, as before; and (2) calling the function does not touch any globally visible mutable state, such as a reference variable defined outside the function body. If a function does touch global mutable state, then its type must be annotated *global*. For example, the function

$$\mathit{fn}\, x \Rightarrow \mathit{let}\ \mathit{val}\, y = \mathit{ref}\, x\ \mathit{in}\ \mathit{fn}\, z \Rightarrow (y := !y + z; !y)$$

has type *int* $\rightarrow$ *global (int* $\rightarrow$ *int)*. The function itself is not *global* (there are no free variables in its definition), but the returned function is (variable y is free in its definition and has type *int ref*). In Figure 1 line 30 there is no *global* annotation, so we can infer from the type that the only mutable state entering *apply* is the *partitions* in the second argument; no such state may be "smuggled in" via the first argument. Note, however, that the user-defined function bound to the first argument can freely allocate and use its own (local) mutable state.

Merge sort. Figure 2 shows how to use the *DisjointSlices* module to implement parallel merge sort with a four-way recursive split. This code is based on the merge sort program in the DPJ benchmarks [3]. Calls to functions declared in *DISJOINT_SLICES* are set off in green bold face.

Function *sort* (lines 33 and following) accepts an *int array* to sort. It wraps the array in a *slices*, adds a fresh array to the *slices*, and passes the result to the helper function *sortSlices*. Function *sortSlices* (lines 8 and following) accepts a *slices* that wraps disjoint slices A and B. A is the input to be sorted, and B is an auxiliary array required by the sorting algorithm. At the end of a call to *sortSlices*, A is sorted in place.

If A is smaller than a predetermined size, then *sortSlices* applies a sequential quicksort to A. Otherwise, it (1) divides A into quarters and sorts each one in parallel; (2) in parallel merges each pair of quarters of A into a half of B; and (3) merges the halves of B back into A. The quarters and halves (lines 19–22) are created by splitting and then transposing, as discussed above. Function *splitFirst* (lines 23–24) splits slice A only: it applies *split* to transform $[A, B]$ into $[[A_1, A_2], [B]]$, and then it applies *flatten* to transform that into $[A_1, A_2, B]$.

Function *merge* (lines 3 and following) accepts a *slices* type containing slices $[A_1, A_2, B]$; it merges A_1 and A_2 into B in

```
1  open DisjointSlices
2
3  fun merge (sls : int slices) : unit =
4    case getList sls of
5      [A1,A2,B] => (* Merge A1, A2 into B *)
6    | _         => raise BadArgument
7
8  fun sortSlices (sls : int slices) : unit =
9    case getList sls of
10     [A,B] =>
11       let
12         val len = ArraySlice.length A
13       in
14         if len <= QUICK_SIZE then
15           quickSort A
16         else let
17           val q = len div 4
18           val quarterIdxs = [q,2*q,3*q]
19           val quarters = transpose (split sls
20             [quarterIdxs,quarterIdxs])
21           val halves =
22             transpose (split sls [[2*q],[2*q]])
23           fun splitFirst idx sls =
24             flatten (split sls [[idx],[]])
25         in
26           (apply sortSlices quarters;
27            apply (merge o (splitFirst q)) halves;
28            merge (splitFirst (2*q) (rev sls)))
29         end
30       end
31   | _ => raise BadArgument
32
33 fun sort (arr : int Array.array) : unit =
34   sortSlices
35     (add (fromArray arr,[(Array.length arr,0)]))
```

Figure 2. Merge sort implementation using *DisjointSlices*.

parallel. We omit most of the code for this function, which is similar to the code shown for *sort*.

Module implementation. Figure 3 shows a possible implementation of the *DisjointSlices* module. In line 34, we assume the existence of a function *ParallelList.apply* that applies the function f in parallel over the elements of *ps*.

To ensure the safety of this module, we must prove that the parallel application in the implementation of *apply* (line 34) is safe, *regardless of the values of f and ps provided by the user*. As stated in the introduction, in contrast to approaches like DPJ [17], we do not extend the type system so that it is powerful enough to carry out this proof; instead, the extended type system just provides enough information so the proof is possible looking only at the API implementation code. We imagine that the proof would be done either manually or with an automatic or semi-automatic theorem prover. Below we sketch how the proof might go.

By the semantics of the parallel map, and the semantics of function types discussed above, it suffices to prove that any *ps* passed to *apply* represents a disjoint partition, i.e., a list of lists of slices such that for each pair of slices, the arrays are different or the index sets are disjoint. To prove this fact, we must examine the API and enumerate all the ways that a *partitions* can be produced.

Here the ML module system helps us. Because of the opaque constraint :> in Figure 3 line 1, together with the abstract type in Figure 1 line 8, the representation of the type *partitions* as a list of lists is hidden outside the module implementation. Therefore the only way the user can get a *partitions* is by splitting a *slices* or applying a transformation such as *transpose* on an existing *partitions*. Similarly, to obtain a *slices* the user must create one using a constructor provided by the module signature, or transform one to another, or flatten a *partitions* into a *slices*. Thus it suffices to prove two facts: (1) any constructor for *slices* creates a disjoint

```
1  structure DisjointSlices :> DISJOINT_SLICES =
2  struct
3
4  mutable type 'a slices =
5    'a ArraySlice.slice list
6  mutable type 'a partitions =
7    'a ArraySlice.slice list list
8
9  fun slices specs =
10   List.map (ArraySlice.full o Array.array) specs
11
12 fun fromArray a = [ArraySlice.full a]
13
14 fun add (a,specs) = a @ (slices specs)
15
16 fun splitOne (slice,is) =
17   let
18     val starts = 0 :: is
19     val ends = is @ [ArraySlice.length slice]
20     val lens = ListPair.map (op -) (ends,starts)
21     fun makeSlice (start,len) =
22       ArraySlice.subslice (slice,start,SOME len)
23   in
24     ListPair.map makeSlice (starts,lens)
25   end
26
27 fun split sliceList isList =
28   ListPair.map splitOne (sliceList,isList)
29
30 fun transpose ([]::_) = []
31   | transpose rows    = map hd rows ::
32                         transpose (map tl rows)
33
34 fun apply f ps = ParallelList.apply f ps
35
36 fun getList slices = slices
37
38 ...
39
40 end
```

Figure 3. Implementation of the *DisjointSlices* module.

slices; and (2) any transformation that maps one *slices* or *partitions* to another preserves disjointness.

As an example of checking fact 1, notice that the API shown in Figure 1 supports constructing a *slices* from fresh arrays (function *slices* in line 12) or wrapping a single array in a *slices* (function *fromArray*). In the first case, the fresh arrays are disjoint by the semantics of the SML array operations used in line 10 of Figure 3, while in the second case the single array is trivially disjoint. Significantly, because of the hidden representation, the user may not simply take an arbitrary list of slices (which might not be disjoint) and wrap them in a *slices*. As an example of checking fact 2, consider function *split* (Figure 3 line 27), which maps a *slices* to a *partitions*. Because *split* partitions the components of the *slices* into disjoint pieces, it should be straightforward to prove from its implementation that if the *sliceList* input is disjoint, then the *partitions* output is disjoint as well.

Finally, notice that the function *getList* (Figure 3 line 36) exposes the list representation of the *slices* type. This exposure is necessary so that the client can access the elements of the *slices* (for example, in Figure 2 lines 4 and 9). This exposure does not present a problem for the correctness argument sketched above. It *would* be a problem if the user could go the other way, i.e., could construct an arbitrary list of slices and make it into a *partitions* object. However, that is not allowed by the API.

2.2 Spatial Region Tree

Our second example is an SML module *RegionTree*, which represents a spatial region tree. This structure stores data (usually rep-

resenting physical objects in space) in its leaves, while the inner nodes of the tree represent partitions of space. Such trees appear, for example, in physics simulations (to simulate particle interactions) and graphics computations (for ray tracing and collision detection).

```
1  signature REGION_TREE =
2  sig
3
4    (* A region tree with read/write privileges *)
5    mutable type 'a tree
6
7    (* A region tree with read-only privileges *)
8    readonly type 'a readOnlyTree
9    readonly type 'a readOnlyNode
10
11   ...
12
13   (* Construct a new empty tree with given number
14      of dimensions and index function *)
15   val empty : int -> 'a indexFn -> 'a tree
16
17   (* Insert a value into the tree *)
18   val insert : 'a tree -> 'a -> unit
19
20   (* Apply a reduction to the tree in parallel,
21      updating the nodes in place *)
22   val reduce : 'a tree -> 'a reduction ->
23     'a option
24
25   (* Obtain a read-only alias to a tree *)
26   val readOnly : 'a tree -> 'a readOnlyTree
27
28   (* Get the root node out of a tree *)
29   val getRoot : 'a readOnlyTree ->
30     'a readOnlyNode option
31
32   (* Get the children of a node *)
33   val getChildren : 'a readOnlyNode ->
34     'a readOnlyNode option array option
35
36   (* Get the data out of a node *)
37   val getData : 'a readOnlyNode option ->
38     'a option
39
40   ...
41
42 end
```

Figure 4. Signature for the *RegionTree* module (partial).

API design. Figure 4 shows selected members of our *RegionTree* API. It is similar to the API described in [16], but it uses the techniques introduced here instead of region and effect annotations for safe parallelism.

Line 5 declares an abstract *mutable* type *'a tree* that represents a region tree carrying data of type *'a*. The API provides three kinds of operations on the type. First, the user may build a region tree by inserting elements repeatedly from the root. Each node stores its children in a mutable array, and the build occurs by updating the child arrays in place. Second, the user may perform a parallel reduction on the tree. This operation starts at the leaves; at each node it reduces the results produced by the node's children into a single result for the node. It also modifies the node in place by storing the result into the node's data field (implemented with a *ref* type). Third, the user may obtain references to the tree nodes in order to write custom read-only traversals.

To use the API, the user must first create a fresh region tree by applying *empty* (line 15) to two arguments: (1) the dimension of space that the tree represents; and (2) a user-defined "index function" that specifies how to traverse the tree when inserting an element. As in [16], the index function maps a tree level and data element to the index of the child to visit next. To add nodes to a tree, the user passes a tree and a data element to *insert* (line 18), which uses the index function stored in the tree to add a node containing the data.

To perform a parallel reduction, the user calls *reduce* (line 22), passing in a standard reduction function of type *'a reduction*, defined to be

$$\textit{'a option} \rightarrow \textit{'a option list} \rightarrow \textit{'a option}.$$

The reduction function takes a current value and a list of child values and reduces them to a single updated value. We use an *option* type so that a node may have an empty data field. As in the *DisjointSlices* API, *'a* is an immutable value, and any mutable state accessed by a function of type *'a reduction* must be local to the function definition.

To support read-only operations on the tree, we introduce an annotation *readonly*, indicating a type that provides a reference to mutable data but may be used only for reading, and not writing, the data. Figure 4 lines 8–9 define two *readonly* types, *readOnlyTree* and *readOnlyNode*, which provide read-only access to a tree or a node respectively. Function *readOnly* (line 26) converts a *tree* into a *readOnlyTree*; its implementation is the identity function, as only the types are significant. As shown in lines 28 and following, the API also provides functions for obtaining a *readonly* reference to the root of a *readOnlyTree*, obtaining *readonly* references to the children of a node, and reading data out of a node.

As with the *mutable* annotation discussed in Section 2.1, the compiler enforces that *readonly* types are consistently used (for example, that a *mutable* type is never bound to a *readonly* type parameter). However, *readonly* and non-*readonly* aliases to the same object may freely coexist: for example, applying *readOnly* to a variable *tree* does not prohibit or restrict the subsequent use of *tree*, as it would in systems based on access permissions [26, 38, 39]. Further, unlike previous systems incorporating immutable types, the compiler does not actually prohibit writes from occurring through references of *readonly* type. Instead, the *readonly* annotation regulates the use of the API, and the actual invariant is provided by the API design and implementation. For example, a correct *RegionTree* implementation must ensure that no operation on a *readOnlyTree* modifies the tree. This allocation of responsibility keeps the user-side type system very simple and minimally restrictive.

Barnes-Hut simulation. We have used the *RegionTree* API to write the Barnes-Hut n-body simulation (BH) [17, 40]. BH simulates the interaction between a number of massive bodies (for example, stars or planets) in a series of time steps. At each time step, the algorithm (1) constructs a region tree containing the bodies at the leaves; (2) performs a bottom-up reduction on the tree to fill in the center-of-mass coordinates for the inner nodes; (3) uses the region tree to compute the forces on the bodies; and (4) uses the forces to update the body positions. In our implementation, steps 2 and 3 are parallel. Step 1 could also be parallelized (by adding a parallel tree build to our API), but we have not done that. The most time-consuming part of the computation — and the best opportunity for parallel speedup — occurs in step 3.

Figure 5 illustrates, in ML-like pseudocode, one time step of our implementation. Function *timeStep* (line 15) accepts an array of *body* objects and computes a new array with the updated positions for that step. Lines 16–17 use the *RegionTree* API to insert the bodies into the tree and fill in the center-of-mass coordinates. Lines 18–19 obtain read-only references to the tree and the array. Line 20 passes the read-only tree and array references to *computeForces*, which returns a new array containing bodies with updated forces. Lines 21–22 update the positions in place in that array and return the array.

Lines 4 and following show the *computeForces* function. This function accepts a pair *(tree,bodies)* of a read-only tree and a

```
1  (* body RegionTree.readOnlyTree *
2     body option Array.readOnlyArray ->
3     body option Array.array *)
4  fun computeForces (tree, bodies) =
5    let f = (* function taking (tree, bodies)
6               and index i to new body
7               with updated force *)
8    let m = ArrayModifier.modifier
9      (Array.length bodies, NONE) (tree,bodies)
10   ArrayModifier.modifyi m f
11   ArrayModifier.getArray m
12
13 (* body option Array.array ->
14    body option Array.array *)
15 fun timeStep bodies =
16   let tree = (* insert bodies into fresh tree *)
17   computeCofM tree
18   let tree' = RegionTree.readOnly tree
19   let bodies' = Array.readOnly bodies
20   let bodies = computeForces (tree',bodies')
21   updatePositions (tree',bodies)
22   bodies
```

Figure 5. Pseudocode for one time step of Barnes-Hut.

```
1  signature ARRAY_MODIFIER =
2  sig
3
4    mutable type ('a, readonly 'b) modifier
5
6    (* Create a new modifier from a fresh array
7       and read-only state *)
8    val modifier : (int * 'a) -> 'b ->
9      ('a,'b) modifier
10
11   (* Apply a modify function in parallel
12      to the array *)
13   val modifyi : ('a,'b) modifier ->
14     ('a,'b) modifyiFn -> unit
15
16   (* Get the array out of the modifier *)
17   val getArray : ('a,'b) modifier ->
18     'a Array.array
19
20   ...
21
22 end
```

Figure 6. Signature for the *ArrayModifier* module (partial).

read-only array. It constructs a function f that reads the tree and array and computes a new body for each index position i. Because the incoming *tree* and *array* types are read-only, this function is constrained to call API functions that accept a *readOnlyTree* or *readOnlyArray* as input. In particular, by the design of the *RegionTree* API, there is no way for f to insert an element into the tree. However, f can obtain read-only references to the tree nodes and their children, to traverse the tree and read its data.

Lines 8–9 construct an *ArrayModifier* for use in generating the new body array. This API, shown in relevant part in Figure 6, encapsulates the pattern of modifying an array in place in parallel while reading disjoint state. After Figure 5 line 9, m stores a reference to an *ArrayModifier.modifier* containing a fresh body array of the same length as *bodies*, and storing the read-only state *(tree,bodies)*.

In Figure 5 line 10, the call to *ArrayModifier.modifyi* uses f to modify m's array in place in parallel. As shown in Figure 6 lines 13–14, function *modifyi* has type

$$\textit{('a,'b) modifier} \rightarrow \textit{('a,'b) modifyiFn} \rightarrow \textit{unit}.$$

The type *modifyiFn* is defined as follows:

$$\textit{type ('a, readonly 'b) modifyiFn} = \textit{'b} \rightarrow \textit{int} \rightarrow \textit{'a}.$$

Here *'a* is the type of an array element, and *'b* is the type of the state being read during the computation of *'a* at each index position of the array. Notice that the *readonly* annotation on type variable *'b* ensures that only read-only state can go into the *modifyiFn*. Thus the type system ensures that the only "modifying" here is done by the modifier itself. Also, notice that the function f in Figure 5 line 5 satisfies this constraint, because the pair *(tree,bodies)* is a pair of read-only types. Finally, Figure 5 line 11 gets the modified array out of the *ArrayModifier* object and returns it.

Correctness argument. Again, we use the type system not to make a complete correctness argument, but to provide enough information so that a correctness argument is possible without seeing any client code. In the BH example, we have used three APIs that incorporate parallelism and/or type annotations: *RegionTree*, *Array*, and *ArrayModifier*. For each API, we must check that (1) the *readonly* type annotations are correctly placed; and (2) the parallel constructs are noninterfering.

To perform the first check, we must ensure that for any API function accepting a *readonly* type, the function does not modify any data reachable from that type. While this kind of check can be hard for a general shared-memory program, it seems quite tractable given the closed-world assumption of our parallel APIs. For example, in our *RegionTree* implementation, the operations that take a *readOnlyTree* don't write any memory at all; they just read data out of arrays and *ref* fields. In the *ArrayModifier* implementation, *modifyi* does accept read-only state and modify an array. However, since the array is created inside the *ArrayModifier* implementation, it cannot alias with any read-only state passed in by the user.

The second check must be done for each API function that is internally parallel; it is similar to the verification of the *DisjointSlices* API discussed in Section 2.1. For example, to verify the parallel reduction provided by *RegionTree*, we could use a technique such as separation logic [37] or regional logic [11] to verify that the tree build indeed produces a tree; and then we could use the tree shape to prove disjointness for the parallel updates. For *ArrayModifier.modifyi*, the verification should be easy, since the only parallel modification is to write values into array cells with disjoint indices.

3. Research Questions

In this section we pose some questions for further research suggested by the examples discussed in Section 2.

How to formalize the type system. We would like to formalize the semantics of the type annotations *mutable* and *readonly*. Specifically, we would like to write down a core calculus and work out the rules for ensuring consistency (1) between type definitions in signatures and modules and (2) between type variables and their bindings. The type system we have in mind is very simple, so we believe this formalization should be straightforward.

Whether the approach is sufficiently general. The approach we have described is feasible only if we can design a set of parallel APIs that is general enough to cover a broad range of parallel algorithms. We believe we can do at least as well as type systems such as DPJ, because for each parallel pattern that DPJ can express (such as divide-and-conquer array updates) we can design a corresponding API. However, to answer this question we must study further examples.

How to verify the API implementations. Verifying the API implementations poses several research questions. First, can we formalize the informal verification arguments sketched in Section 2? Second, can the verification be partially or totally automated, for

example using an SMT solver? Automatic or semi-automatic proof, where possible, can greatly lower the barrier to adoption of a verification technology. Third, how will the verification scale?

Whether the parallel performance is acceptable. We would like to understand the performance impact of this approach compared to approaches that rely on more powerful user type systems. We can think of two potential impacts. First, since we are providing high-level APIs, we are giving the user less control over exactly how a parallel algorithm is constructed than if we were to provide more fundamental constructs, such as parallel loops and direct memory access. We believe with a suitably designed set of APIs this problem should not be too severe.

Second, our approach does rely on immutability more than some other approaches, such as DPJ. While our merge sort example (Section 2.1) closely tracks the DPJ merge sort benchmark, our Barnes Hut implementation (Section 2.2) relies on slightly more copying of immutable values, instead of in-place updates. For example, in the Java version, the force computation modifies the fields of *body* objects in place, whereas the implementation shown here generates a new array of bodies. In general, greater use of immutable values simplifies the analysis, but by introducing more copies it can also stress the allocator and garbage collector and increase working set sizes in the cache.

To make a preliminary investigation into this question, we ported the SML examples described in Section 2 to F#. We did this because F# contains a subset that is close to SML, and it has a parallel runtime, whereas SML is sequential. We ran the F# code on a virtualized Windows XP platform (running in VMWare 3.1.4 on top of OS X). For merge sort on an array of size 2^{27} we saw a speedup of 1.5x on two cores and 2x on four cores. For Barnes-Hut we measured each of the parallelized force computation, the parallelized center-of-mass computation, and the entire computation. With an input size of 6400 bodies we saw a speedup of 1.2x on two cores for each of the three measurements. When we increased the input size to 64000 we saw no speedup.

The merge sort results are respectable, but not as good as the speedups reported for the DPJ benchmarks [17]. The Barnes-Hut results are disappointing. Further investigation is needed here. In particular, it is not clear whether the reduced performance is inherent in the API approach, or in some tuning issue in our code unrelated to our APIs (for example, the performance impact of stack-allocated structs vs. heap-allocated records in F#), or in some inherent limitation of the F# runtime versus the Java runtime. To explore this issue further, we plan to re-implement the APIs and benchmarks in either Java or Scala (both of which run on the JVM). This should provide a more direct point of comparison with DPJ, and give us a way to isolate and eliminate performance bottlenecks.

If it turns out that more in-place mutability is required for good performance, then there are at least two approaches we could take. The first one is simply to add more patterns. For example, instead of an *ArrayModifier* that writes values into an array, we could support an array with elements of type *('a,'b)*, where *'a* represents the fields being updated, and *'b* represents the unmodified fields. This would be similar to assigning different regions to different fields of the same object in DPJ. The second approach would be to selectively add uniqueness types (for example, an array of unique references) to support additional in-place updates.

4. Related Work

Languages such as ML [36], OCaml [5], F# [4], C# [1], and Scala [33] already enable the general style of programming we explore here, by supporting both higher-order functional abstractions and imperative code. OCaml and F# in particular have a *mutable* keyword for distinguishing mutable from immutable object fields (Scala's *val* and *var* are similar). However, these languages don't support the checking of safe parallelism, because they allow unrestricted use of aliases to mutable objects. Lime [10] is a Java-based language that uses value types similar to ours; however, it is specialized to streaming and dataflow computations, whereas we aim to capture more general patterns via APIs.

The monadic capabilities of Haskell [35] are similar to the ML type system extensions we explore here: imperative computations in Haskell must occur "inside a monad," and this prevents mutable state from entering a computation where it is not supposed to. Haskell monads have been used to write elegant concurrent APIs [30, 31]. However, Haskell monads, while powerful, are less familiar to programmers than straightforward imperative updates of data structures such as trees, arrays, and hash maps.

Languages such as Æminium [38] and HJp [39] provide similar safety guarantees to ours using types that express *permissions* or *capabilities* such as uniqueness and/or immutability. Haller and Odersky [27] have designed a simple capability system for guaranteeing race-safety in actor-based concurrency. Recent work by Gordon et al. [26] is similar, but with a focus on parallelism. ParaSail [7] requires all references to globally visible mutable objects to be unique, so (for example) references cannot be used to construct cyclic data structures.

Uniqueness types are powerful, but they restrict aliasing of mutable objects. They also require the programmer to understand sometimes subtle rules about how permissions are split, joined, consumed, borrowed, etc. One of our goals here is to avoid explicit uniqueness types in user code, although uniqueness invariants might be helpful in verifying API implementations. As a point of comparison, lines 18–19 of Figure 5, which convert *mutable* to *readonly* references, are reminiscent of the splitting rules in permission-based systems [32]. But in our approach such splitting is done with ordinary function calls; there are no extra typing rules for splitting.

Effect systems such as FX [25], DPJ [17], and Liquid Effects [28] use effect annotations to achieve similar guarantees to ours. However, in those systems the user has to write and understand the effect annotations. For example, compare Figure 2 with the DPJ implementation of merge sort [3], which uses region parameters, region constraints, and effect summaries to establish the required disjointness and noninterference properties. In Kawaguchi et al.'s system [28] many of the annotations are inferred, but the overhead of writing and understanding the annotations still seems nontrivial.

There has recently been much work on compiler and runtime mechanisms for ensuring race freedom [9] and determinism [12, 13, 21, 22, 34] in parallel programs. These mechanisms are attractive because they generally require little or no programmer annotation. However, dynamic checks can add runtime overhead, and they often provide a weaker guarantee than static checks (e.g., throwing an exception when a race or determinism violation is discovered). Further, they can be brittle (e.g., providing a semantics that varies with small changes to the program). Compiler-based techniques can also require complex and possibly obscure analysis, leading to problems if programmers need to understand what is going on in order to tune their code. In this work we use mostly-functional APIs to obtain the transparency and strong guarantees of simple, type-based static checking with annotation overhead that is not much greater than the compiler and runtime approaches.

Finally, the idea of using abstractions to encapsulate parallel patterns is of course not new. For example, Clojure [2] and Galois [29] provide APIs that encapsulate transactional operations, and Cilk++ [24] has *hyperobjects* that support patterns such as reduction operations. However, to our knowledge we are the first to explore the idea of *verified parallel APIs* for establishing a noninterference property with a minimally-extended type system.

5. Future Work

In addition to addressing the questions posed in Section 3, we would like to extend the work by developing APIs for different kinds of parallel abstractions. The abstractions presented in this paper focus on noninterfering parallelism, where concurrent access to memory is either disjoint or read-only. However, we believe that the idea is much more general. For example, one could easily add abstractions representing atomic and commutative operations on shared state [17]; atomic (not necessarily commutative) operations on shared state in the manner of transactions [18, 29]; futures [23]; pipelines [16]; or actors [8]. We believe that recent ideas in parallel programming such as concurrent revisions [20] and deterministic reservations [14] can also be adapted to work with our approach.

While the details of these abstractions still have to be worked out, the unifying idea is that *all interactions between parallel tasks should occur through parallel APIs*. For example, in a language with mutable objects o and arrays a, there should never be direct reads or writes to $o.f$ or $a[i]$ for any o or a that is accessible by multiple tasks. Task-local operations on $o.f$ and $a[i]$ are still supported, as are localized cyclic data structures (e.g., a circular list created in a single task). However, any inter-task communication, or global data structures designed to be operated on in parallel, must be managed by a safe parallel API. This is in contrast to an approach like DPJ, which allows direct access to shared mutable data, but requires region and effect annotations to prove safety.

Typically (as in the examples studied here) the API user would write a higher-order function that operates on local state and pass it into an abstraction; then the abstraction would orchestrate the application of the function to global state. That way, all parallel interaction through shared memory must be done in a way that the API implementor can "see" and verify as safe.

Acknowledgments

We thank Joshua Sunshine, Alex Potanin, and the WoDet reviewers for helpful comments. This work was supported by NSF grant #CCF-1116907 and CMU|Portugal grant CMU-PT/SE/0038/2008.

References

[1] *http://msdn.microsoft.com/en-us/library/618ayhy6.aspx*.

[2] *http://clojure.org*.

[3] *https://github.com/dpj/DPJ/*.

[4] *http://msdn.microsoft.com/en-us/library/dd233181.aspx*.

[5] *http://caml.inria.fr/pub/docs/manual-ocaml/*.

[6] *https://github.com/bocchino/ParAbs/*, .

[7] *http://parasail-programming-language.blogspot.com*, .

[8] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[9] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. In *PLDI*, 2008.

[10] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. *OOPSLA*, 2010.

[11] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, 2008.

[12] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[13] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[14] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *PPOPP*, 2012.

[15] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *HotPar*, 2009.

[16] R. L. Bocchino and V. S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *ECOOP*, 2011.

[17] 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 *OOPSLA*, 2009.

[18] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

[19] J. Boyland. Checking interference with fractional permissions. In *SAS*, 2003.

[20] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, 2010.

[21] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[22] Y. h. Eom, S. Yang, J. C. Jenista, and B. Demsky. DOJ: Dynamically parallelizing object-oriented programs. In *PPOPP*, PPoPP '12, 2012.

[23] C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

[24] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA*, 2009.

[25] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole. Report on the FX-91 programming language. Technical Report MIT/LCS/TR-531, 1992.

[26] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA*, 2012.

[27] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, 2010.

[28] M. Kawaguchi, P. Rondon, A. Bakst, and R. Jhala. Deterministic parallelism via liquid effects. In *PLDI*, 2012.

[29] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[30] D. Leijen, M. Fahndrich, and S. Burckhardt. Prettier concurrency: purely functional concurrent revisions. In *Haskell Symposium*, 2011.

[31] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Haskell Symposium*, 2011.

[32] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. In *POPL*, 2012.

[33] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Inc., 2008.

[34] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[35] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[36] L. C. Paulson. *ML for the working programmer (2nd ed.)*. Cambridge University Press, New York, NY, USA, 1996.

[37] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *IEEE Symposium on Logic in Computer Science*, 2002.

[38] S. Stork, P. Marques, and J. Aldrich. Concurrency by default: Using permissions to express dataflow in stateful programs. In *Onward!*, 2009.

[39] E. Westbrook, J. Zhao, Z. Budimlíc, and V. Sarkar. Practical permisssions for race-free parallelism. In *ECOOP*, 2012.

[40] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.