Parallel Functional Arrays

Ananya Kumar
Carnegie Mellon University, USA
ananyak@andrew.cmu.edu

Guy E. Blelloch
Carnegie Mellon University, USA
blelloch@cs.cmu.edu

Robert Harper
Carnegie Mellon University, USA
rwh@cs.cmu.edu

Abstract

The goal of this paper is to develop a form of functional arrays (sequences) that are as efficient as imperative arrays, can be used in parallel, and have well defined cost-semantics. The key idea is to consider sequences with functional value semantics but non-functional cost semantics. Because the value semantics is functional, "updating" a sequence returns a new sequence. We allow operations on "older" sequences (called interior sequences) to be more expensive than operations on the "most recent" sequences (called leaf sequences).

We embed sequences in a language supporting fork-join parallelism. Due to the parallelism, operations can be interleaved non-deterministically, and, in conjunction with the different cost for interior and leaf sequences, this can lead to non-deterministic costs for a program. Consequently the costs of programs can be difficult to analyze. The main result is the derivation of a deterministic cost dynamics which makes analyzing the costs easier. The theorems are not specific to sequences and can be applied to other data types with different costs for operating on interior and leaf versions.

We present a wait-free concurrent implementation of sequences that requires constant work for accessing and updating leaf sequences, and logarithmic work for accessing and linear work for updating interior sequences. We sketch a proof of correctness for the sequence implementation. The key advantages of the present approach compared to current approaches is that our implementation requires no changes to existing programming languages, supports nested parallelism, and has well defined cost semantics. At the same time, it allows for functional implementations of algorithms such as depth-first search with the same asymptotic complexity as imperative implementations.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords cost semantics, concurrency, parallel, functional data structures, persistence, arrays

1. Introduction

Supporting sequences (arrays) with efficient (constant time) accesses and updates has been a persistent problem in functional languages. It is easy to implement such sequences with constant time access (using arrays stored as contiguous memory), or to implement with logarithmic time accesses and updates using balanced trees, but it seems that getting both accesses and updates in constant time cannot be achieved without some form of language extension. This means that algorithms for many fundamental problems are a logarithmic factor slower in functional languages than in imperative languages. This includes algorithms for basic problems such as generating a random permutation, and for many important graph problems (e.g., shortest-unweighted-paths, connected components, biconnected components, topological sort, and cycle detection). Simple algorithms for these problems take linear time in the imperative setting, but an additional logarithmic factor in time in the functional setting, at least without extensions.

A variety of approaches have been suggested to alleviate this problem. Many of the approaches are based on the observation that if there is only a single reference to an array, it can be safely updated in place. The most common such approach is to use monads (Moggi 1989; Wadler 1995). Monads can force code to be single threaded and can thread mutable arrays (or other mutable state) so the only reference to them is the current one. Haskell supplies the ST monad for this purpose. King and Launchbery (1995), for example, use STMonads with an array to implement depth first search (DFS) in linear work. Guzman and Hudak's single-threaded lambda calculus also uses the idea of single threading the computation (Guzmán and Hudak 1990), and they motivate the approach based on allowing for fast array updates. The problem with these approaches is that they are basically implementing a second single-threaded language within a functional language. Using monads means using a new syntax not just for the operations themselves but for much of the code the operations are embedded in. Indeed King and Launchbery have to write completely different code for DFS, even though it is only for efficiency and not correctness (it is trivial to implement DFS in $O(m \log n)$ time purely functionally). Monads also do not allow keeping persistent versions of structures, exactly because the state can be changed. More importantly, however, monads force the program itself to be single threaded, inhibiting parallelism.

A second approach is to use a type system that enforces the single-threadedness of individual arrays rather than the program as a whole. This can be done with linear types (Girard 1987; Wadler 1990), which can be used to ensure that references are not duplicated. This is more flexible than monads, since it allows the threading of the data to be distinct from the threading of the program. However, such type systems are not available in most languages, and can be hard to reason about.

A third approach is to support fully functional arrays in a general functional language, and to check either statically or dynamically if they happen to be used in a single threaded manner. This can be done, for example, by using reference counting (Hudak and Bloss 1985; Hudak 1986). The idea is to keep track of how many references there are to an array and update in place if there is only one. Hudak describes techniques to statically analyze code to determine where the count must be one, making it safe to replace an

POPL'17, January 15–21, 2017, Paris, France
© 2017 ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009869

update with an in place update. The problem with this approach is that the efficiency of the code depends heavily on the effectiveness of the compiler and the specific way the code is written, and it can therefore be hard for the user to reason about efficiency.

The last approach is to fully support functional arrays, even with sharing, but using more sophisticated data structures. This is sometimes referred to as version tree arrays (Aasa et al. 1988) or fully persistent arrays (Driscoll et al. 1989). The idea is to maintain a version tree for an array (or more generally other data types), such that any node of the tree can be updated to create a new leaf below it with the modified value, without affecting the value of other nodes. Dietz (1989) shows that arrays can be maintained fully persistently using $O(\log \log n)$ time per operation (access or update). Holmstrom and Hughes (Aasa et al. 1988) suggest storing a version tree by keeping the full array at the root, and storing the update performed at each node. A related idea is trailer arrays (Bloss 1989), used by Haskell's DiffArrays, which store the full array for each leaf node, and a history of updates that were performed. Trailer arrays support $O(1)$ accesses and updates to the most recent version of an array, however accessing old versions of an array requires searching through the history and takes unbounded work. Chuang's approach (Chuang 1992) improves trailer arrays so that accessing old versions takes $O(n)$ work. O'Neill (2000) describes various improvements.

None of the existing approaches properly support concurrent operations on functional arrays. O'Neill (2000) suggests (in passing) having a lock for each element of the array. However, when many threads contend for the same element, this would serialize accesses and therefore they would not take constant time. Additionally, per-element locks add significant memory overhead.

1.1 Cost-Bounded Parallel Sequences

We present a new approach for efficiently supporting functional arrays (sequences). It uses some ideas from previous work, but unlike the previous approaches it supplies a well-defined cost dynamics (operational semantics), and supports parallelism, without language extensions. More specifically the approach has the following important features.

1. (Functional) It has fully functional value dynamics—i.e., when not considering costs, sequences act purely functionally. In particular, it requires no changes in existing languages and no special types, syntactic extensions, etc.
2. (Efficient) Accessing or updating the most recent versions of a sequence is constant time. Accessing old versions of a sequence requires at most $O(\log n)$ work and updating an old version requires at most $O(n)$ work, where n is the length of the sequence.
3. (Parallel) The approach supports fork-join (nested) parallelism—sequences can be passed to parallel calls and safely updated and read, again with a purely functional value dynamics. Our internal data-structures are wait-free (Herlihy 1988) so no process has to wait for another.
4. (Analysis) We supply a well defined cost dynamics, which can be used to formally analyze the cost of any program. The dynamics captures both sequential and parallel costs. We describe a cost-bounded implementation which maps costs in the model to costs on either a sequential or parallel RAM (PRAM).

We have implemented the approach and in this paper present some performance numbers.

We consider sequences with three functions: `new`, `get` (read), and `set` (write). `new`(n, v) creates a new sequence of length n with the value v at each index, `get`(A, i) returns the i^{th} element of A, and `set`(A, i, v) returns a "new" sequence where the i^{th} element

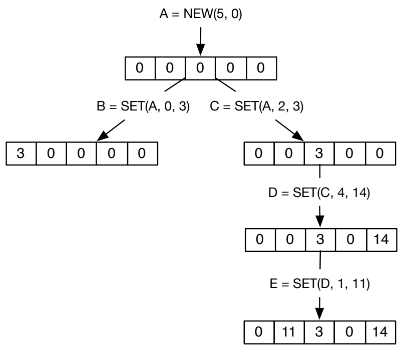

Figure 1. Example usage of sequences

of A is replaced by v. In the following discussion n refers to the length of a sequence. Figure 1 gives an example.

As in previous work (Aasa et al. 1988) the history of a sequence after it is created with `new` can be viewed as a version tree. A version tree has interior nodes and leaves. In Figure 1, after all functions are applied, sequences A, C, and D are *interior nodes*, and sequences B and E are *leaves* of the version tree. In our cost dynamics applying `get` and `set` to sequences at the leaves takes constant work, but applying them to interior nodes is more expensive.

The sequential implementation of sequences is straightforward. Each sequence has a version number and a reference to an instance of type ArrayData. Multiple sequences can reference the same ArrayData instance. However, exactly one of these sequences can be a leaf sequence. The other sequences must be interior, and are ancestors of the leaf sequence in the version tree. For example, in Figure 1, A and B might reference an ArrayData instance, and C, D, and E might reference another ArrayData instance.

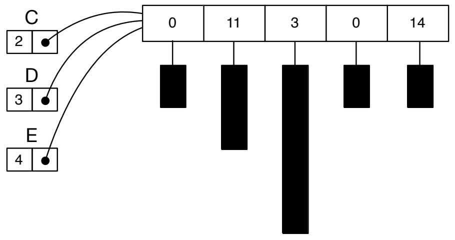

Figure 2. Simplified visualization of sequence implementation. E is the leaf node and has the highest version number. The version labeled change logs are shaded black.

In each ArrayData instance, we store a mutable Value array that contains the values in the leaf sequence. `get` on a leaf simply reads the appropriate index in the Value array. For each index in the Value array, we also store a change-log of values that were previously at that index. Entries in the change log are labeled by version numbers. `get` on an interior sequence involves binary searching the log at the appropriate index.

`set` on a leaf sequence involves modifying the Value array at the desired index and appending a change-log entry to store the value that was overwritten. `set` on an interior sequence involves extracting the values at each index using `get`, copying the values to a new ArrayData instance with empty logs, and overwriting the desired index in the new Value array. Note that `set` on an interior sequence leads to a new branch in the version tree.

To ensure that change logs do not become too large, whenever the total size summed across all change logs reaches n, the Value array is copied. This requires $O(n)$ work, but can be amortized against the updates. The copying ensures that `get` and `set` on interior nodes have $O(\log n)$ and $O(n)$ work respectively. The wait-free concurrent implementation is more involved, and we defer the description and pseudocode to section 6.

Although our value dynamics is purely functional our cost dynamics is not. The cost dynamics requires passing a store and the costs depend on the order the store is threaded. To allow for parallelism we allow for arbitrary interleaving of the steps in the cost dynamics. We note that lazy languages also have a pure value dynamics but impure cost dynamics—call-by-name and call-by-need only differ in their costs, and a store is required to model the difference (Ariola et al. 1995).

Throughout the paper we use *work* to refer to the to total number of primitive instructions used by a computation. On a sequential machine this is equivalent to time. We use the term *span* (also called depth, or dependence depth) to refer to the number of parallel steps assuming an unbounded number of processors. Work and span are useful since together they can be used to bound the time on any fixed number of processors (Brent 1974; Blumofe and Leiserson 1999; Blelloch and Greiner 1995).

1.2 Overview of Paper

We use the general approach of cost-semantics and bounded cost implementations (Blelloch and Greiner 1995, 1996; Greiner and Blelloch 1999; Harper 2012; Blelloch and Harper 2013). The idea is to define a high-level cost dynamic semantics (dynamics) for a language and then prove that for a given implementation the costs (e.g. time or space) on the target machine can be bounded by the costs in the semantics using some relationship. This is usually done in two stages: first the high-level evaluation-based cost dynamics is mapped into an intermediate single-stepping structural dynamics (with costs), and then the structural dynamics is mapped onto a machine model. We follow a similar framework here.

We start with a simple pure call-by-value lambda calculus extended with sequences. The high-level cost dynamics is an extension of the standard evaluation dynamics for lambda calculus, and defines the work and span for every computation. The extension needs to include a mutable store to keep track of whether each sequence is a leaf or interior, so that the costs for each case can be properly accounted for. The tricky part is accounting for combining work at the join points of parallel calls.

We use two separate intermediate structural dynamics for the language: the *idealized parallel dynamics* and the *interleaved dynamics*. The idealized parallel dynamics is a pure dynamics and allows all parallel expressions to proceed on the same step. The number of steps gives the *span* of the computation. The interleaved dynamics defines the *work* of a computation. The dynamics is not pure, requiring, as with the evaluation semantics, a store to keep track of whether each sequence is a leaf or interior, so that different costs can be charged for `get` and `set` in the two cases. To account for parallelism, the dynamics allows for non-deterministic interleaving of steps on parallel expressions. In conjunction with the impure cost for `get` and `set`, this means the overall work is non-deterministic. At this level we still assume the `get` and `set` operations are atomic. We prove a tight relationship between the work and span given by the evaluation dynamics and the worst case for those given by the structural dynamics (over all possible interleavings).

We then map our sequence functions onto an impure target language in which the only atomic operations are reads, writes, and compare-and-swap instructions. We describe a wait-free concurrent implementation of sequences that uses careful synchronization. We assume that instructions in the target language can be interleaved in any way–i.e., at any given step many `gets` and `sets` can be in progress, with the individual instructions within their implementation interleaved. We show correctness with respect to any interleaving, and show that specific linearization points within the implementation define the relative order of `gets` and `sets` with respect to the structural dynamics.

By bounding the cost of the wait-free implementation we bound the overall work and span of the computation on the target machine based on that determined from the evaluation dynamics.

To evaluate our approach in practice we ran a variety of experiments on our preliminary concrete implementation. The experiments show that reads are 3% to 12% slower in leaf sequences than in regular Java arrays, and updates are 2.4 to 3.3 times slower. Considering the purity and additional functionality provided by our sequence implementation, this is a small slowdown. Furthermore, preliminary experiments back up our theoretical claim that threads can access our sequences with a high level of concurrency.

The results on cost dynamics can likely be extended to other data types where the cost of `get` and `set` are different for leaf and interior versions, even if there are multiple varieties of `get` and `set` functions. In particular, our sequence implementation can likely be extended to functional unordered sets implemented with hash tables, and our approach might be useful in developing more efficient implementations of functional disjoint sets (although a functional disjoint set implementation where all operations cost $\log n$ is known (Italiano and Sarnak 1991)).

2. Language

The notation we use for our language and dynamics is from (Harper 2016).

We use a standard applicative-order, call by value source language defined as follows:

$$e = x \mid c \mid \lambda(x, y).e \mid e_1 e_2 \mid (e_1, e_2) \mid (e_1 \parallel e_2) \mid \text{if } e_1\ e_2\ e_3$$

The constants c contains the usual arithmetic types, such as the natural numbers and numerical functions. The expression $(e_1 \parallel e_2)$ indicates that the two expressions can run in parallel, returning a pair (v_1, v_2) when both expressions are fully evaluated, while (e_1, e_2) generates a pair sequentially. Sequential and parallel pairs only differ in the cost dynamics—the value dynamics are identical. Our language can be augmented to add support for recursion using LETREC or a fixed point combinator.

Definition 2.1. The terminal (fully evaluated) values are lambda expressions, constants, and pairs of terminal values, and are denoted by the judgement VAL.

Our language also contains functions to work with sequences. $\texttt{new}(n, v)$ evaluates to a sequence of size n with the value v at each index. $\texttt{get}(A, i)$ evaluates to the i^{th} element of sequence A. $\texttt{set}(A, i, v)$ evaluates to a new sequence where the i^{th} element of A is substituted with v.

3. Structural Dynamics

As discussed in the introduction we use two separate structural dynamics for the language: the idealized parallel dynamics and the interleaved dynamics. The first captures the span and is deterministic, and the second captures the work and allows for non-deterministic interleaving. The work is measured based on the worst case interleaving.

In the dynamics we use $\text{GET}(A, i)$, $\text{SET}(A, i, v)$ and $\text{NEW}(n, v)$ to indicate the pure primitive versions of the operations, and `get`, `set`, and `new` as the corresponding operations in the language.

$$\frac{A, a \ \mathsf{val}}{\mathtt{get}(A, a) \rightarrow_{par} \mathrm{GET}(A, a)} \ \text{(get-eval)}$$

$$\frac{e_1 \rightarrow_{par} e_1' \quad e_2 \rightarrow_{par} e_2'}{e_1 \parallel e_2 \rightarrow_{par} e_1' \parallel e_2'} \ \text{(step-both)}$$

$$\frac{v_1 \ \mathsf{val} \quad e_2 \rightarrow_{par} e_2'}{v_1 \parallel e_2 \rightarrow_{par} v_1 \parallel e_2'} \ \text{(step-right)}$$

$$\frac{v_2 \ \mathsf{val} \quad e_1 \rightarrow_{par} e_1'}{e_1 \parallel v_2 \rightarrow_{par} e_1' \parallel v_2} \ \text{(step-left)}$$

$$\frac{v_1, v_2 \ \mathsf{val}}{(v_1 \parallel v_2) \rightarrow_{par} (v_1, v_2)} \ \text{(join)}$$

Figure 3. Example rules in the parallel dynamics.

3.1 Idealized Parallel Dynamics

The idealized parallel dynamics captures the evaluation steps taken on a machine with an unbounded number of processors and is given by the following judgement, where e is the expression and e' is the resulting expression:

$$e \rightarrow_{par} e'$$

We show the rules for get and fork-join in Figure 3. As usual, the judgement val describes terminal values. Notice that in fork-join, both sides of the fork-join take a step at the same time if possible. The rules for `set` and `new` are similar to the rules for `get`, and the rules for function application and if-then are as in a standard applicative order functional programming language.

The parallel structural dynamics, unlike the interleaved structural dynamics, is deterministic.

Definition 3.1. A *parallel transition sequence* T is a sequence $(e_0, ..., e_n)$ with $e_i \rightarrow_{par} e_{i+1}$ for all $0 \leq i < n$. We say that $e_0 \rightarrow_{par}^{n} e_n$ or $e_0 \rightarrow_{par}^{*} e_n$.

Definition 3.2. The *span* of a parallel transition sequence T, denoted by $SP(T)$, is its length n.

3.2 Interleaved Cost Dynamics

The interleaved dynamics is a concurrent dynamics and is given by the following judgement, where σ is the store, e is the expression, σ' is the resulting store, e' is the resulting expression after a step has been taken, and w is the work.

$$\sigma, e \rightarrow \sigma', e', w$$

Sequence values are represented by the pair (l, V) where l is a label that indexes the store σ and V is a list of the elements in the sequence. The store is a mapping from labels to either + (indicating a leaf sequence) or – (indicating an interior sequence). Let $L(\sigma)$ denote the set of labels in the store σ.

Let $gl(V)$ and $gi(V)$ be the work of `get` applied to V if it is a leaf or interior sequence respectively, and $sl(V)$ and $si(V)$ be the work of `set` applied to V if it is a leaf or interior sequence respectively. Let $n(a)$ be the work of evaluating `new`(a). We assume that $gl(V) \leq gi(V)$ and $sl(V) \leq si(V)$ (operating on leaf sequences is cheaper that operating on interior sequences).

The dynamics is given in Figure 4. Note that `set`(A, a), where $A = (l, V)$ and $\sigma[l] = +$, creates a new label and value, extends the store to indicate the new value is a leaf, and updates the store at l to indicate that A is now interior. This is the impure aspect of the cost dynamics since there can be other references to l. Also note that the work for `get` and `set` depends on whether the sequence is a leaf or interior.

$$\frac{v_1, v_2 \ \mathsf{val}}{\sigma, (\lambda(x, y).e)(v_1, v_2) \rightarrow \sigma, [v_1/x][v_2/y]e, 1} \ \text{(func-app)}$$

$$\frac{}{\sigma, \mathtt{if}\ \mathtt{true}\ e_2\ e_3 \rightarrow \sigma, e_2, 1} \ \text{(if-true)}$$

$$\frac{}{\sigma, \mathtt{if}\ \mathtt{false}\ e_2\ e_3 \rightarrow \sigma, e_3, 1} \ \text{(if-false)}$$

$$\frac{a \ \mathsf{val} \quad l \notin L(\sigma)}{\sigma, \mathtt{new}(a) \rightarrow \sigma[l \mapsto +], \mathrm{NEW}(a), n(a)} \ \text{(new)}$$

$$\frac{A = (l, V) \quad \sigma[l] = + \quad a \ \mathsf{val}}{\sigma, \mathtt{get}(A, a) \rightarrow \sigma, \mathrm{GET}(V, a), gl(V)} \ \text{(get-leaf)}$$

$$\frac{A = (l, V) \quad \sigma[l] = - \quad a \ \mathsf{val}}{\sigma, \mathtt{get}(A, a) \rightarrow \sigma, \mathrm{GET}(V, a), gi(V)} \ \text{(get-interior)}$$

$$\frac{A = (l, V) \quad \sigma[l] = + \quad l' \notin L(\sigma) \quad a \ \mathsf{val}}{\begin{array}{c}\sigma, \mathtt{set}(A, a) \rightarrow \sigma[l \mapsto -, l' \mapsto +], \\ (l', \mathrm{SET}(V, a)), sl(V)\end{array}} \ \text{(set-leaf)}$$

$$\frac{A = (l, V) \quad \sigma[l] = - \quad l' \notin L(\sigma) \quad a \ \mathsf{val}}{\begin{array}{c}\sigma, \mathtt{set}(A, a) \rightarrow \sigma[l' \mapsto +], \\ (l', \mathrm{SET}(V, a)), si(V)\end{array}} \ \text{(set-interior)}$$

$$\frac{\sigma, e_1 \rightarrow \sigma', e_1', w}{\sigma, e_1 \parallel e_2 \rightarrow \sigma', e_1' \parallel e_2, w} \ \text{(step-left)}$$

$$\frac{\sigma, e_2 \rightarrow \sigma', e_2', w}{\sigma, e_1 \parallel e_2 \rightarrow \sigma', e_1 \parallel e_2', w} \ \text{(step-right)}$$

$$\frac{v_1, v_2 \ \mathsf{val}}{\sigma, (v_1 \parallel v_2) \rightarrow \sigma, (v_1, v_2), 1} \ \text{(join)}$$

Figure 4. The interleaved dynamics. Rules for stepping the arguments to `get`, `set`, `new`, and other language constructs are omitted.

The rules for fork-join are non-deterministic—either side of the fork can take a step (step-left or step-right), but not both. This allows for arbitrary interleaving of instructions on different sides of a fork-join. After both sides of a fork-join are fully evaluated the parallel pair is converted to a regular pair. Note that we assume that steps are themselves atomic. We relax this assumption in the implementation level (Section 6).

Definition 3.3. A *transition sequence* T is a sequence of states $[(\sigma_0, e_0), ..., (\sigma_n, e_n)]$ and per-step work $[w_1, ..., w_n]$ s.t. for all $0 \leq i < n$, $\sigma_i, e_i \rightarrow \sigma_{i+1}, e_{i+1}, w_{i+1}$. We say that T takes σ_0, e_0 to σ_n, e_n and has length n and denote this by $\sigma_0, e_0 \rightarrow^n \sigma_n, e_n$.

Definition 3.4. We say that T is *maximal* if there does not exist σ', e', w' s.t. $\sigma_n, e_n \rightarrow \sigma', e', w'$.

Definition 3.5. A *transition subsequence* $T_{i,j}$ of T is given by the sequence of states $[(\sigma_i, e_i), ..., (\sigma_j, e_j)]$ and per-step work $[w_{i+1}, ..., w_j]$. Note that $T = T_{0,n}$.

Definition 3.6. The *work of a transition sequence* T is given by:

$$W(T) = \sum_{i=1}^{n} w_i$$

Different transition sequences starting and ending at the same states may have different work—the work can depend on the order in which the instructions are interleaved. Figure 5 shows 3 ways a leaf sequence A might be used in two parallel calls. In particular, one side of the fork might call `get` on A while the other side calls `set` (see the center panel). Calls to `get` on the left branch would have work $gl(A)$ if and only if they execute before the call to `set` on the right branch, otherwise they would have work $gi(A)$.

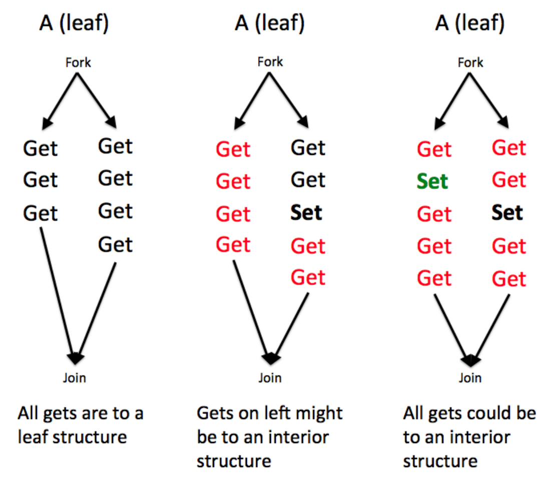

Figure 5. Some ways a sequence might be used in a fork join.

We cannot make any assumptions about how the instructions in different branches of a fork are interleaved, so we need to capture the worst case work over all possible interleavings.

Definition 3.7. Let w be the max of $W(T)$ over all transition sequences starting at σ_0, e_0. If $W(T)$ is unbounded then $w = \infty$. w is the *work of evaluating* σ_0, e_0.

3.3 Machine Costs

We now consider the cost of evaluating an expression on a shared memory machine with P processors. We assume that within a constant factor, the cost functions $gl(V)$, $gi(V)$, $sl(V)$, $si(V)$, and $n(a)$ are (possibly amortized) upper bounds on the number of instructions needed by the implementation for each of the array operations. In this case we say the cost functions have valid work bounds. In Section 6 we describe an implementation and prove work bounds for sequences.

The operations `new`, `get` and `set` can each have parallelism in their implementation, so in addition to the work we need to know the span for these functions (the number of time steps on an unbounded number of processors). Here we assume the span for `new`, `get`, and `set` is bounded by some $f(P)$ where P is the number of processors used. It is important that the bounds on span are not amortized.

Using the greedy scheduling theorem (Blumofe and Leiserson 1999) or Brent's theorem (Brent 1974) knowing the total work and total span (number of steps on an unbounded number of processors) is sufficient to bound the time on any finite number of processors. The analysis holds even if the work is amortized, as it is in the present case. We assume that standard instructions such as reading and writing to memory, arithmetic operations, etc. take constant time (and work). This leads to the following theorem.

Theorem 3.1. *Given cost functions with valid work bounds each with maximum span $f(p)$, then for any expression e with a constant number of variable names if w is the work of evaluating σ, e, and S is the span for the (unique) maximal transition sequence starting at e, then e can be evaluated using any greedy schedule on a P processor machine in time*

$$T \le c\left(\frac{w}{P} + Sf(P)\right)$$

for some constant c that is independent of the expression e.

Proof. (outline). This follows from the greedy scheduling theorem and previous work on bounded cost implementations. Bounding the number of variables is needed to bound the cost of evaluating each step of the program itself (Blelloch and Greiner 1995). It avoids issues of non-constant cost for looking up the values of variables in environments. □

4. Cost Dynamics

The cost dynamics gives an easy, deterministic way to compute the cost of a program without having to reason about multiple interleavings. In this section, we give the rules for the cost dynamics. In Section 5, we prove that the costs in the cost dynamics are a tight upper bound for the costs in the structural dynamics. The results can be easily extended for data types besides sequences (like unordered sets) even if there are multiple varieties of `get` and `set` functions.

To handle the non-determinism in fork-join, the cost dynamics first determines the worst-case cost of each branch in the fork independently starting at the state at the fork. Then for sequences that are used in both forks, it adds additional work at the join point. The semantics keeps track of the number of `get`s to leaf values on each side of the fork so that it can compute this additional work. If a value was `set` on one side of the fork, then the `set` might have happened before `get`s on the other side of the fork, so additional work needs to be charged for the `get`s.

The cost dynamics is defined by the following judgment, where δ is the store, e is the expression, δ' is the new store, v is the value that e evaluates to, w is the work, and s is the span.

$$\delta, e \Downarrow \delta', v, w, s$$

As in the previous section, sequences are represented by (l, V) where l is a label that indexes the store δ and V is a list of the elements in the sequence. The store is a mapping from labels to pairs $(+/-, c)$, where c represents the number of leaf `get`s on the value indexed by the label. $L(\delta)$ denotes the set of labels in the store δ.

The rules in the cost dynamics are given in figure 6. Note that `get` costs $gl(V)$ instead of $gi(V)$ on a leaf value V but we increment the counter of leaf `get`s in the store.

The fork-join rule is the most interesting and requires a few definitions. Suppose we have expression $(e_L \parallel e_R)$ with

$$\delta, e_L \Downarrow \delta_L, v_L, w_L, s_L$$

$$\delta, e_R \Downarrow \delta_R, v_R, w_R, s_R$$

Further, suppose that $(L(\delta_L) \setminus L(\delta)) \cap (L(\delta_R) \setminus L(\delta)) = \emptyset$ (the new labels produced on both sides of the fork-join do not conflict). Consider a sequence $A = (l, V)$ with $\delta[l] = (s, c)$, $\delta_L[l] = (s_L, c + c_L)$, $\delta_R[l] = (s_R, c + c_R)$. When multiplying two signs or multiplying a sign with an integer, consider + to be 1 and – to be 0.

COMBINE describes how to combine store values on both sides of a fork-join. A sequence is a leaf iff it is a leaf on both sides of the fork-join. Leaf `get`s on one side of the fork remain leaf `get`s iff there were no calls to `set` on the other side of the fork.

$$\text{COMBINE}((s, c), (s_L, c + c_L), (s_R, c + c_R)) = (s_L\, s_R, c + s_R\, c_L + s_L\, c_R)$$

$$\delta' = (\delta_L \backslash \delta) \cup (\delta_R \backslash \delta) \cup \bigcup_{l \in L(\delta)} [l \mapsto \text{COMBINE}(\delta[l], \delta_L[l], \delta_R[l])]$$

EXTRAWORK gives the additional cost incurred if a sequence was modified on either side of a fork-join. If the value was interior

$$\frac{}{\delta, c \Downarrow \delta, c, 1, 1}\ \text{(constants)}$$

$$\frac{\begin{array}{c}\delta, e_1 \Downarrow \delta', \lambda(x,y).e, w_1, s_1 \\ \delta', e_2 \Downarrow \delta'', (v_1, v_2), w_2, s_2 \\ \delta'', [v_1/x][v_2/y]e \Downarrow \delta''', v', w_3, s_3\end{array}}{\begin{array}{c}\delta, e_1\ e_2 \Downarrow \delta''', v', 1 + w_1 + w_2 + w_3, \\ 1 + s_1 + s_2 + s_3\end{array}}\ \text{(f-app)}$$

$$\frac{\delta, e_1 \Downarrow \delta', \texttt{true}, w_1, s_1 \quad \delta', e_2 \Downarrow \delta'', v, w_2, s_2}{\delta, \texttt{if}\, e_1\, e_2\, e_3 \Downarrow \delta'', v, 1 + w_1 + w_2, 1 + s_1 + s_2}\ \text{(if-true)}$$

$$\frac{\delta, e_1 \Downarrow \delta', \texttt{false}, w_1, s_1 \quad \delta', e_3 \Downarrow \delta'', v, w_2, s_2}{\delta, \texttt{if}\, e_1\, e_2\, e_3 \Downarrow \delta'', v, 1 + w_1 + w_2, 1 + s_1 + s_2}\ \text{(if-false)}$$

$$\frac{\delta, e \Downarrow \delta', a, w, s \quad l \notin L(\delta')}{\delta, \texttt{new}(e) \Downarrow \delta'[l \mapsto (\mathbf{+}, 0)], \textsc{new}(a), w + n(a), s + 1}\ \text{(new)}$$

$$\frac{A = (l, V) \quad a\,\mathsf{val} \quad \delta[l \mapsto (\mathbf{+}, c)] \quad l' \notin L(\delta)}{\begin{array}{c}\delta, \texttt{set}(A, a) \Downarrow \delta[l \mapsto (\mathbf{-}, c), l' \mapsto (\mathbf{+}, 0)], \\ (l', \textsc{set}(V, a)), sl(V), 1\end{array}}\ \text{(set-leaf)}$$

$$\frac{A = (l, V) \quad a\,\mathsf{val} \quad \delta[l \mapsto (\mathbf{-}, c)] \quad l' \notin L(\delta)}{\begin{array}{c}\delta, \texttt{set}(A, a) \Downarrow \delta[l' \mapsto (\mathbf{+}, 0)], \\ (l', \textsc{set}(V, a)), si(V), 1\end{array}}\ \text{(set-int)}$$

$$\frac{\begin{array}{c}\delta; e_1 \Downarrow \delta_1, A, w_1, s_1 \quad \delta_1, e_2 \Downarrow \delta_2, a, w_2, s_2 \\ \delta_2, \texttt{set}(A, a) \Downarrow \delta', A', w', s'\end{array}}{\delta, \texttt{set}(e_1, e_2) \Downarrow \delta', A', w_1 + w_2 + w', s_1 + s_2 + s'}\ \text{(set-eval)}$$

$$\frac{A = (l, V) \quad a\,\mathsf{val} \quad \delta[l \mapsto (\mathbf{+}, c)]}{\begin{array}{c}\delta, \texttt{get}(A, a) \Downarrow \delta[l \mapsto (\mathbf{+}, c + 1)], \\ \textsc{get}(V, a), gl(V), 1\end{array}}\ \text{(get-leaf)}$$

$$\frac{A = (l, V) \quad a\,\mathsf{val} \quad \delta[l \mapsto (\mathbf{-}, c)]}{\delta, \texttt{get}(A, a) \Downarrow \delta, \textsc{get}(V, a), gi(V), 1}\ \text{(get-interior)}$$

$$\frac{\begin{array}{c}\delta, e_1 \Downarrow \delta_1, A, w_1, s_1 \quad \delta_1, e_2 \Downarrow \delta_2, a, w_2, s_2 \\ \delta_2, \texttt{get}(A, a) \Downarrow \delta', v', w', s'\end{array}}{\delta, \texttt{get}(e_1, e_2) \Downarrow \delta', v', w_1 + w_2 + w', s_1 + s_2 + s'}\ \text{(get-eval)}$$

$$\frac{\begin{array}{c}\delta, e_L \Downarrow \delta_L, v_L, w_L, s_L \quad \delta, e_R \Downarrow \delta_R, v_R, w_R, s_R \\ (L(\delta_L) \setminus L(\delta)) \cap (L(\delta_R) \setminus L(\delta)) = \emptyset\end{array}}{\begin{array}{c}\delta, (e_L \parallel e_R) \Downarrow \delta', (v_L, v_R), 1 + w_L + w_R + w', \\ 1 + \max(s_L, s_R)\end{array}}\ \text{(fj)}$$

Figure 6. Rules for the cost dynamics. w' and δ' in the fork-join rule are defined in the text.

before the fork-join, then all functions incurred their maximal cost and there is no additional cost. Otherwise, leaf `get`s on one side of the fork are charged gi work iff the other side of the fork called `set`. Additionally, if both sides of the fork called `set`, then one of the `set`s came first and has work s_L and the subsequent `set` has work si. We abuse notation so that gl, gi, sl, si directly take in a label l instead of the corresponding sequence.

$$\begin{array}{l}\textsc{extrawork}(l, (s, c), (s_L, c + c_L), (s_R, c + c_R)) = \\ \quad ((\neg s_R)\, c_L + (\neg s_L)\, c_R)(gi(l) - gl(l)) + \\ \quad (\neg s_R)(\neg s_L)(si(l) - sl(l))\end{array}$$

$$w' = \sum_{l \in L(\delta), \delta[l \mapsto (\mathbf{+}, c)]} \textsc{extrawork}(l, \delta[l], \delta_L[l], \delta_R[l])$$

Then, the fork-join cost dynamics are:

$$\delta; (e_L \parallel e_R) \Downarrow \delta'; (v_L \parallel v_R); 1 + w_L + w_R + w'; 1 + \max(s_L, s_R)$$

5. Cost Validation

We show that the work computed by the cost dynamics is a tight upper bound for the work in the interleaved structural dynamics. Proofs for the span bounds are omitted because they are standard (the parallel structural dynamics is deterministic, with no interleaving).

Definition 5.1. Consider a transition sequence T. We say that $S_i, e_i \rightarrow S_{i+1}, e_{i+1}, w_{i+1}$ is a `get` on l if the step involves evaluating the `get` function on structure (l, V). We say it is a *cheap* `get` on l if $w_{i+1} = gl(l)$. The number of cheap `get`s on l in T is denoted by $SC_l(T)$.

Definition 5.2. Unlike the store in the cost dynamics, the store in the structural dynamics only stores whether each structure is **+** (leaf) or **–** (interior). *sign*(δ) takes a store that maps $l \mapsto (s, c)$ and returns a store which maps $l \mapsto s$.

Definition 5.3. Suppose that $\delta, e \Downarrow \delta', e', w', s'$. We define the number of cheap `get`s in going from δ to δ' in the cost dynamics as follows. Suppose $l \mapsto (s', c') \in \delta'$. If $l \mapsto (s, c) \in \delta$ then $EC_l(\delta, \delta') = c' - c$ and if $l \notin L(\delta)$ then $EC_l(\delta, \delta') = c'$. If $l \notin L(\delta')$ then $EC_l(\delta, \delta') = 0$.

Definition 5.4. A *relabeling* of L_1 is a bijective function R from label sets $L_1 \rightarrow L_2$. R can be used to relabel stores and expressions. $R(\delta) = \{R(l) \mapsto e \mid l \mapsto e \in \delta \wedge l \in L_1\} \cup \{l \mapsto e \mid l \mapsto e \in \delta \wedge l \notin L_1\}$. In other words, R relabels some of the labels in δ. Similarly, $R(e)$ returns an expression e' where each occurrence of a label $l \in L_1$ in e is substituted by $R(l)$.

Lemma 5.1. *Suppose that there exists a derivation of depth m that $\delta, e \Downarrow \delta', e', w, s$. Let R be an arbitrary labeling. Then there exists a derivation of depth m that $R(\delta), R(e) \Downarrow R(\delta'), R(e'), w, s$.*

Proof. By applying the relabeling to each step of the derivation, and noting that all rules in the cost dynamics hold under relabelings. ☐

Lemma 5.2. *If $S, e \rightarrow S', e', w$ and U is a store s.t. for all $l \in e$, $l \in U$, then there exists U', w' s.t. $U, e \rightarrow U', e', w'$*

Proof. By a case analysis on the structural dynamics, noting that transitions of expressions never depend on the values in the store (the dynamics is purely functional). ☐

Theorem 5.3. *Suppose that*

$$\delta, e_0 \Downarrow \delta_E, e_E, w_E, s_E$$

and consider arbitrary maximal transition sequence T taking

$$S_0 = \textit{sign}(\delta), e_0 \rightarrow^n S_n, e_n$$

The following hold:

1. *For some relabeling R of $L(\delta_E) \setminus L(\delta)$, $S_n = \textit{sign}(R(\delta_E))$ and $e_n = R(e_E)$.*

2. *For all $l \in \delta_E$, $EC_l(\delta, \delta_E) \leq SC_{R(l)}(T)$.*
3. *The sum of work and cheap* gets *is conserved:*

$$w_E + \sum_{l \in L(\delta_E)} EC_l(\delta, \delta_E)(gi(l) - gl(l)) =$$
$$W(T) + \sum_{l \in L(\delta_E)} SC_{R(l)}(T)(gi(R(l)) - gl(R(l)))$$

Proof. By induction on the length of the shortest derivation in the cost dynamics. We sketch the theorem for one of the rules in the cost dynamics: get-leaf. The other cases follow from a similar line of reasoning.

Case get-eval-leaf: Suppose $e_0 = \mathtt{get}(e_L, e_R)$ and

$$\delta, e_L \Downarrow \delta_L, e'_L, w_L$$

$$\delta_L, e_R \Downarrow \delta_R, e'_R, w_R$$

$$\delta_R, \mathtt{get}(e'_L, e'_R) \Downarrow \delta_E, e_E, gl(l')$$

where $e'_L = (l', V)$ for some V, which gives us

$$\delta, \mathtt{get}(e_L, e_R) \Downarrow \delta_E, e_E, w_E$$

with $w_E = w_L + w_R + gl(l')$.

The structural dynamics first step the left argument to get, then the right argument, and finally evaluate the get. So there exists m such that $T_{0,m}$ involves stepping the left argument, $T_{m,n-1}$ involves stepping the right argument, and $T_{n-1,n}$ involves evaluating the get.

Part 1: We show the first part of the theorem holds. Essentially, we can apply the induction hypothesis to $T_{0,m}$ to get a relabeling R of labels in $L(\delta_L)/\delta$ s.t. $S_m = sign(R(\delta_L))$ and $e_m = \mathtt{get}(R(e'_L), e_R)$. Similarly, we can apply the inductive hypothesis to $T_{m,n-1}$ [1]. We then note that get evaluates to the same value, and doesn't modify the sign of the stores, in both the interleaved and cost dynamics. By composing the relabelings, the first part of this theorem holds. Without loss of generality, suppose that the stores and expressions were relabeled so that the first part holds.

Part 2: From lemma 5.1, the length of the shortest derivation for the relabeled cost dynamics does not change, so we can apply the inductive hypothesis even after the relabeling. We apply the inductive hypothesis to $T_{0,m}$ and $T_{m,n-1}$. This tells us the number of cheap gets in the structural dynamics is bounded by the number of cheap gets in the cost dynamics.

Part 3: Apply the IH on $T_{0,m}$ and $T_{m,n-1}$. Then we get, $W(T) = W(T_{0,m}) + W(T_{m,n-1}) + gl(l') = w_L + w_R + gl(l')$. ☐

Theorem 5.4. *(Work Bound) Given the conditions in theorem 5.3, $w_E \geq W(T)$*

Proof. We assume that δ_E has been relabeled as described in part 1 of theorem 5.3. For all $l \in \delta_E$, $EC_l(\delta, \delta_E) \leq SC_l(T)$. This implies that $\sum_{l \in \delta_E} EC_l(\delta, \delta_E)(gi(l) - gl(l)) \leq \sum_{l \in \delta_E} SC_l(T)(gi(l) - gl(l))$. But then from the conservation of sum of work and cheap gets, $w_E \geq W(T)$. ☐

Theorem 5.5. *(Tightness) Suppose that $\delta, e_0 \Downarrow \delta_E, e_E, w_E, s_E$. Then, for some n, there exists a transition sequence T from $s_0 = sign(\delta), e_0 \rightarrow^n s_n, e_n$ with $2W(T) \geq w_E$.*

[1] Technical note: we construct projected sequences for $T_{0,m}$ and $T_{m,n-1}$, and appeal to lemmas 5.1 and 5.2, before applying the inductive hypothesis.

Proof. By induction on the rules of the cost dynamics. We present the construction for the most interesting case, fork-join.

Case fork-join: Suppose $e_0 = (e_L \parallel e_R)$ and

$$\delta, e_L \Downarrow \delta_L, v_L, w_L, s_L$$

$$\delta, e_R \Downarrow \delta_R, v_R, w_R, s_R$$

which gives us

$$\delta, (e_L \parallel e_R) \Downarrow \delta', (v_1 \parallel v_2), w_L + w_R + w'$$

From the inductive hypothesis, there exists transition sequence T_L taking $s_0^L, e_0^L \rightarrow^m s_m^L, e_m^L$ with $s_0^L = sign(\delta)$, $e_0^L = e_L$, $e_m^L = v_L$ and $2W(T_L) \geq w_L$. Similarly, there exists transition sequence T_R taking $s_0^R, e_0^R \rightarrow^n s_n^R, e_n^R$ with $s_0^R = sign(\delta)$, $e_0^R = e_R$, $e_n^R = v_R$ and $2W(T_R) \geq w_R$.

The function CHEAP-GETS computes the number of gets to a seq on a particular side of the fork.

$$\text{CHEAP-GETS}((s, c), (s_{\text{my}}, c_{\text{my}}), (s_{\text{oth}}, c_{\text{oth}})) = s_{\text{oth}}(c_{\text{my}} - c)$$

c_L represents the extra costs of gets on the left fork that become expensive (because of sets on the right fork). Similarly, c_R represents the extra costs of gets on the right fork that become expensive (because of sets on the left fork).

$$c_L = \sum_{l \in L(\delta), \delta[l \mapsto (+,c)]} \text{CHEAP-GETS}(\delta[l], \delta_L[l], \delta_R[l])(gi(l) - gl(l))$$

$$c_R = \sum_{l \in L(\delta), \delta[l \mapsto (+,c)]} \text{CHEAP-GETS}(\delta[l], \delta_R[l], \delta_L[l])(gi(l) - gl(l))$$

The function DOUBLE-SETS computes whether a seq was set on both sides of the fork-join.

$$\text{DOUBLE-SETS}((s_L, c_L), (s_R, c_R)) = s_L s_R$$

$$ss = \sum_{l \in L(\delta), \delta[l \mapsto (+,c)]} \text{DOUBLE-SETS}(\delta_L[l], \delta_R[l]) s_i(l)$$

If $c_R \geq c_L$ then we step the left side of the fork to completion before the right side of the fork. If $c_L > c_R$ then we step the right side of the fork to completion before the left side. We can then show that the resulting transition sequence T satisfies $2W(T) \geq w_E$.

Without loss of generality we may assume that $c_R \geq c_L$. In the cost dynamics, $w' = c_L + c_R + ss$. Because we step the left side of the fork before the right side, and $c_R \geq c_L$, $2W(T) \geq 2(W(T_L) + W(T_R) + c_R + ss) \geq 2W(T_L) + 2W(T_R) + c_R + c_L + ss \geq w_L + w_R + c_L + c_R + ss \geq w_L + w_R + w'$. ☐

Theorem 5.6. *(Conditional termination) Suppose that $\delta, e_0 \Downarrow \delta_E, e_E, w_E, s_E$. Then there exists n s.t. all transition sequences starting at $sign(\delta), e_0$ have length $\leq n$.*

Proof. By induction on the rules of the cost dynamics. ☐

Suppose that we use the cost dynamics to derive that the work of evaluating an expression e is w. When evaluating e, the stores in the cost dynamics and structural dynamics are initially empty. In particular, the signs of the store in the cost dynamics and structural dynamics are the same. From the tightness theorem, we know there exists a transition sequence T starting at e. From the conditional termination theorem, all transition sequences starting at e have bounded length. Then, from the work bound theorem, the work w computed by the cost dynamics is at least the work of any transition sequence. Since the structural dynamics captures the (non-deterministic) execution time of the expression, this means that the cost dynamics gives an upper bound for the execution time of the expression.

6. Implementation

In this section we present an implementation of functional arrays (sequences) and prove its correctness. We show that $gl(V) = sl(V) = O(1)$ (that is, `get` and `set` on leaf sequences require constant work), $gi(V) = O(\log n)$, and $si(V) = O(n)$, where n is the size of V (that is, `get` and `set` on interior sequences require bounded work) are valid amortized work bounds for this implementation. As such, we can use the cost dynamics we derived to analyze costs of parallel programs involving sequences.

We give SML-like code for the implementation. In it we construct and manipulate mutable arrays (henceforth called arrays) using three functions: `tabulate`(n, f) creates a new array of size n which for all i contains $f(i)$ at the i^{th} index; `sub`(A, i) evaluates to the i^{th} element of array A; and `update`(A, i, v) mutates index i of array A to have value v. `sub` and `update` take constant work, and `tabulate` takes $O(n + W)$ work and $O(\max(\log n, S))$ span where W is the sum of the work of all the function calls to f, and S is the maximum of the span of the function calls to f.

We also use ML-style reference (`ref`) cells. In addition to reads (`!a`) and writes (`a := v`) to `ref`s we assumes an atomic compare and swap function `cmpswap`(a:`int ref`, v:`int`, v':`int`):`bool`. It compares the value at a to v, and if and only if they are the same sets the value at a to v' and returns true, otherwise it returns false.

We assume a p-processor parallel machine model. For the correctness proofs, we assume a sequentially consistent memory model. When analyzing costs, we assume that all processors are synchronized with respect to a global clock (not needed for correctness), and a greedy schedule, such that if there are l instructions available at a time step we will run $\min(p, l)$ of them.

As described in the introduction we use an ArrayData structure that keeps for each index the most recent version as well as a log of older versions. However, the concurrent version needs to be much more careful of how the data structures are updated and accessed.

6.1 Log Implementation

We use a Log data structure to store the logs. Logs supports three functions: `new`() creates a new empty log, `push`$(l, (V, a))$ inserts a new version into the log l with version number V and value a, and `get_version`(l, V) accesses the value corresponding to smallest (earliest) version in the long that is greater than or equal to version V. Version numbers are represented as integers. All functions have amortized constant work. The definition for Logs is given in Figure 7. The Log's array stores the entries and has a given size and capacity. The `push` function tries to add to the end of the array if there is capacity, and if not, it copies all entries to a new array that is twice as large, doubling the capacity. The copy can be done in $O(n)$ work and $O(\log n)$ span for capacity n using `update`. Because it is only applied every n steps this gives the amortized work bounds. The `get_version` function searches the log using binary search for the appropriate version. The function can return `NONE` if there is no such version.

The functions on Logs can be used semi-concurrently. In particular at most one thread can execute a `push` at a time, but multiple threads can call `size` and `get_version`. If used in this way then Logs are linearizable (Herlihy and Wing 1990).

6.2 Sequence Implementation

The sequence implementation keeps a Value array of the most recent values for each index, which represents the values at a leaf node of the version tree. For each index it also stores a change-log that keeps track of values at interior nodes of the version tree. The definition for sequences is given in Figure 8. Note that multiple sequences can reference the same ArrayData. Figure 9 visualizes a newly created sequence A, and Figure 10 visualizes `set`s on interior and leaf sequences.

```
type capacity = int
type size = int
type version = int
type 'a entry = version × 'a
type 'a Log = (capacity × size ×
               ('a entry) option array) ref

val push : 'a Log × 'a entry → unit
fun push($A$ as ref($c, s, D$), $v$) =
  if $c = s$ then
    let val $c' = 2 \times c$
        val $D'$ = Array.tabulate($c'$, fn _ ⇒ NONE)
    in copyArray($D, D'$);
       Array.update($D', s$, SOME($v$));
       $l$ := ($c', s+1, D'$)
    end
  else
    Array.update($D, s$, SOME($v$));
    $l$ := ($c, s+1, D$)

val new() : unit → Log
fun new() = (1,0,Array.tabulate(1, fn _ ⇒ NONE))

val get_version : 'a Log × version → 'a option
fun get_version($l, V$) =
  let val($c, s, D$) = !$l$
  in if $s$ = 0 then NONE
     else let val SOME($V'$,_) = Array.sub($D$,$s - 1$)
     in if $V' < V$ then NONE
        else SOME ‘binary search for smallest version
                   $\geq V$, and get its value’
     end
  end
```

Figure 7. Code for Logs.

The `set` operation uses a compare and swap to increment the ArrayData's version so that only one thread can modify an ArrayData at any point in time. If the compare and swap fails (which means the sequence is an interior sequence), the values in the sequence are copied over to produce a new ArrayData, and an empty Log is created at each index. Note that the values are not directly copied into the new ArrayData from the Value array, instead `get` is called at each index so that the correct version is used. If the compare and swap is successful (which means the sequence is a leaf sequence), a log entry is inserted and the Value array is mutated.

The ordering of the instructions in `set` is critical. If the Value array is modified before the log entry is inserted then a `get` evaluated between the 2 instructions could evaluate to the wrong value.

The `get`(A, i) operation first loads the value v at the i^{th} index of the Value array. If the sequence's version and corresponding ArrayData's version match (meaning that the sequence is a leaf sequence), then `get` evaluates to v. Otherwise `get` looks up the value in the Log.

As in `set`, the ordering of the instructions in `get` is important. In particular, index i of the Value array must be loaded before the versions are compared or the logs are examined. If the versions are compared first and a `set` is evaluated between the 2 instructions, the `set` might modify the Value array and cause the `get` to evaluate to the wrong value.

```
31 type 'a logs = ('a Log) array
32 type 'a ArrayData = version ref × 'a array
33                                  × 'a logs
34 type 'a sequence = version × 'a ArrayData

35 val new : int × 'a → 'a sequence
36 fun new(size, init) =
37   (1, (ref 1, Array.tabulate(size, fn i ⇒ init),
38     Array.tabulate(size, fn _ ⇒ Log.new())))

39 val set : 'a sequence × int × 'a → 'a sequence
40 fun set(S as (V, (V_r, A, L)), i, v) =
41   if not(cmpswap(V_r, V, (V + 1))) orelse
42       !V_r = Array.length(A) then
43     let val n = Array.length(A)
44         val A' = Array.tabulate(n, fn i ⇒ get(S, i))
45         val L' = Array.tabulate(n, fn _ ⇒ Log.new())
46     in Array.update(A', i, v); (1, (ref 1, A', L'))
47     end
48   else
49     Log.push(Array.sub(L, i), (V, Array.sub(A, i)));
50     Array.update(A, i, v);
51     (V + 1, (V_r, A, L))

52 val get : 'a sequence × i → 'a
53 fun get((V, (V_r, A, L)), i) =
54   let val guess = Array.sub(A, i)
55       val l = Array.sub(L, i)
56   in if V = !V_r then guess
57      else case Log.get_version(l, V) of
58             NONE ⇒ guess
59           | SOME(v) ⇒ v
60   end
```

Figure 8. Code for sequences.

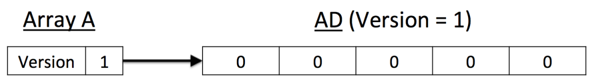

Figure 9. A = new(5, 0)

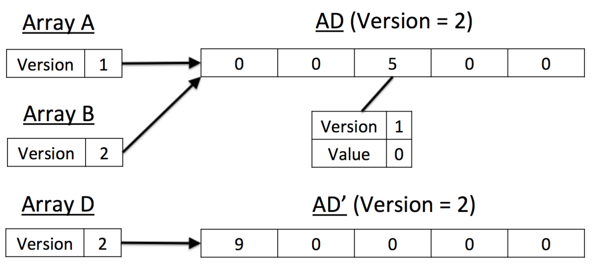

Figure 10. $B = \texttt{set}(A, 2, 5)$ changes the Value array and adds a log entry. Then $D = \texttt{set}(A, 0, 9)$ creates AD' because A is interior.

6.3 Target Language

For the sake of the proofs we give a formal definition of the target language in which we implement sequences.[2] The target language extends the source language to support mutable references and compare and swap. The structural dynamics of the target language is given by the following judgement, where (π, μ) is a memory store and (π', μ') is the memory after the step has been taken,

$$(\pi, \mu), e \rightarrow_T (\pi', \mu'), e'.$$

For simplicity we assume all sequences and ArrayDatas are int sequences and int ArrayDatas. To avoid dealing with overflow issues, we assume that all integers in the target language are unbounded. Sequences are represented as &l where the label l indexes into the store π.

The memory contains an immutable store π that maps labels to $(\lambda, *k)$ where λ is an integer version number and k is an index into μ. Intuitively, π contains all the sequences created so far in the evaluation of the expression. New key-value pairs can be inserted into π but existing mappings cannot be modified. The memory also contains a mutable store μ that maps labels to values of type int ArrayData.

For concision we do not give the full dynamics of our target language. As examples, we give the rules for creating a new ArrayData, compare and swap, and pushing a log entry. Let a_n denote an array of n as and let $\{\}$ denote an empty (int $*$ int) Log.

$$\frac{n \text{ int val} \quad k \notin L(\mu)}{(\pi, \mu), \text{NEWAD}(n) \rightarrow_T (\pi, \mu[k \mapsto (0, 0_n, \{\}_n)]), *k} \text{(new-ad)}$$

$$\frac{V, V' \text{ int val} \quad \mu[k] = (V, A, L)}{\begin{array}{c}(\pi, \mu), \text{CMPSWAP}(*k, V, V') \rightarrow_T \\ (\pi, \mu[k \mapsto (V', A, L)]), \mathit{true}\end{array}} \text{(cas-true)}$$

$$\frac{V, V' \text{ int val} \quad \mu[k] \neq (V, A, L)}{(\pi, \mu), \text{CMPSWAP}(*k, V, V') \rightarrow_T (\pi, \mu), \mathit{false}} \text{(cas-false)}$$

$$\frac{\begin{array}{c}\mu[k] = (V, A, L) \quad i, V', v \text{ int val} \\ L[i] = (p_1, ..., p_n) \quad L' = L[i \mapsto (p_1, ..., p_n, (V', v))]\end{array}}{(\pi, \mu), \text{PUSH}(*k, i, (V', v)) \rightarrow (\pi, \mu[k \mapsto (V, A, L')]), *k} \text{(push)}$$

The dynamics for `push` given above suggests that `push` is atomic, whereas in the implementation we use a compare-and-swap at line 41 to ensure that calls to push do not overlap. In the correctness proof given below we take this exclusion property as given for the sake of concision; a complete proof would require ensuring that exclusion is maintained by the use of compare-and-swap in the actual implementation.

In the source language, evaluating `new`, `get`, and `set` involves a single step. In the target language, `new`, `get`, and `set` are function calls to our implementation. Evaluating the function calls requires multiple steps. Let $C(\texttt{new})$, $C(\texttt{get})$, $C(\texttt{set})$ be the implementation (that is, the lambda expression) for `new`, `get`, and `set` respectively.

The following rules show the evaluation of `get` and `set` in the target. A source `get` is substituted with its implementation. For `set`, we use a "wrapper," called `set'`, so that the sequence resulting from evaluation of `set` can be added to the store before completion. We keep track of the sequences created so we can show invariants on sequences in the proof of correctness.

$$\frac{i \text{ int val}}{(\pi, \mu), \texttt{get}(\&l, i) \rightarrow_T (\pi, \mu), C(\texttt{get})(\&l, i)} \text{(get)}$$

[2] The target language is a simplification of the SML-like code used in the samples, but suffices for the proofs.

$$\frac{i, v \text{ int val}}{(\pi,\mu), \texttt{set}(\&l, i, v) \rightarrow_T (\pi,\mu), \texttt{set}'(C(\texttt{set})(\&l, i, v))} \text{ (start-set)}$$

$$\frac{(\pi,\mu), e \rightarrow_T (\pi',\mu'), e'}{(\pi,\mu), \texttt{set}'(e) \rightarrow_T (\pi',\mu'), \texttt{set}'(e')} \text{ (eval-set)}$$

$$\frac{l \notin L(\pi)}{(\pi,\mu), \texttt{set}'((V, *k)) \rightarrow_T (\pi[l \mapsto (V, *k)], \mu), \&l} \text{ (end-set)}$$

The rules for `new` are similar to `set`. As in the source structural dynamics, either side of a fork-join expression in the target can take a step. This makes the target structural dynamics non-deterministic. However, because `get` and `set` involve multiple steps, the target has a greater number of possible interleavings than the source.

Definition 6.1. A transition sequence in the target language is a sequence of memory stores $[(\pi_0,\mu_0), ..., (\pi_n,\mu_n)]$ and expressions $[e_0, ..., e_n]$ such that for all $0 \leq i < n$, $(\pi_i,\mu_i), e_i \rightarrow_T (\pi_{i+1},\mu_{i+1}), e_{i+1}$. We say that $(\pi_0,\mu_0), e_0 \rightarrow_T^* (\pi_n,\mu_n), e_n$ or $(\pi_0,\mu_0), e_0, \rightarrow_T^n (\pi_n,\mu_n), e_n$.

An expression e in the source language is compiled to an expression e' in the target and then executed according to the target's dynamics. Since the source language is a subset of the target language, $e = e'$. In other words an expression e in the source language can be viewed as an expression in the target.

We show that our implementation of sequences in the target correctly implements the source. The non-atomicity of `set` and `get` means that we cannot appeal to standard parametricity theorems for the proofs.

6.4 Correctness of the Implementation

The correctness proof proceeds along the lines of (Birkedal et al. 2012) and (Turon et al. 2013). We concentrate here on the critical parts, involving `new`, `get`, and `set`. It takes the form of a rely-guarantee argument (Jones 1983) for the correctness of the implementation integer sequences. To avoid complications we omit the proof of atomicity of the push operation mentioned earlier. To account for would require a more elaborate invariant that tracks the state of the interlock used by the compare-and-swap operation.

We state invariants on the memory store and constraints on the possible ways that the memory store can evolve in a valid target program. Given these invariants and constraints, we show that `new`, `get`, and `set` behave the same way in the source and target. The tricky part of the proof is setting up the invariants and theorems—the proofs involve extensive but straightforward casework.

Definition 6.2. (Valid memories) Consider memory store (π,μ). Consider arbitrary label $l \in L(\pi)$ and suppose $\pi[l] = (\lambda, k)$. We say (π,μ) is valid at l if the following hold.

1. $k \in L(\mu)$. Assuming this is true, let $\mu[k] = (V, A, L)$.
2. $0 \leq \lambda \leq V$
3. For all i and $0 \leq j < \text{LEN}(L[i])$ if $L[i][j] = (V', v)$ then $V' < V$
4. For all i and $0 \leq j < k < \text{LEN}(L[i])$, if $L[i][j] = (V_1, v_1)$ and $L[i][k] = (V_2, v_2)$ then $V_1 < V_2$

We say (π,μ) *valid* if for all $l \in L(\pi)$, (π,μ) is valid at l.

Theorem 6.1. *Let e be an expression in the source language. Suppose that $(\{\},\{\}), e \rightarrow_T^* (\pi,\mu), e'$. Then (π,μ) valid.*

Proof. (Sketch) By induction on the length of the transition sequence taking $(\{\},\{\}), e$ to $(\pi,\mu), e'$. For the base case, verify that each condition holds for $(\{\},\{\})$.

The inductive hypothesis is that that the theorem holds for all transition sequences of length n. We then consider an arbitrary transition sequence of length $n+1$. From the hypothesis, the first n steps leads to a valid memory state. We then case on all possible modifications that the $n+1$th step can make to memory. Showing properties (1) to (3) is straightforward. To prove property (4), we need to appeal to a lemma that different executions of lines 49 to 51 in the implementation of `set` on the same ArrayData cannot overlap. The lemma holds because of property (2) and the compare and swap. ☐

Definition 6.3. (Memory transformations) Assume that (π,μ) and (π',μ') are valid.[3]. Consider arbitrary label l with $l \in L(\pi)$ and $l \in L(\pi')$, and suppose that $\pi[l] = (\lambda, k)$ and $\pi'[l] = (\lambda', k')$. We say that $(\pi,\mu) \leq (\pi,\mu)$ at l if the following conditions hold:

1. $\lambda = \lambda'$ and $k = k'$ (in other words, values in π are immutable). Assuming this is true, let $\mu[k] = (V, A, L)$ and $\mu'[k] = (V', A', L')$.
2. $V \leq V'$. In other words, the versions are non-decreasing.
3. For all i, $\texttt{len}(L[i]) \leq \texttt{len}(L'[i])$ and for all i, j with $0 \leq j < \texttt{len}(L[i])$, $L[i][j] = L'[i][j]$. Intuitively, this means that logs can only be extended, existing entries cannot be overwritten.
4. If $\texttt{len}(L[i]) < \texttt{len}(L'[i])$ then $L'[\texttt{len}(L[i])] = (V'', A[i])$ where $V \leq V''$.
5. For all i, if $A[i] \neq A'[i]$ then $\lambda < V'$ and exists $0 \leq j < \texttt{len}(L'[i])$ with $L'[i][j] = (V'', A[i])$ and $\lambda \leq V''$.

We say that $(\pi,\mu) \leq (\pi',\mu')$ if $L(\pi) \subseteq L(\pi')$ and for all l such that $l \in L(\pi)$ and $l \in L(\pi')$, $(\pi,\mu) \leq (\pi',\mu')$ at l.

Lemma 6.2. *(Reflexivity) If* (π,μ) valid *then* $(\pi,\mu) \leq (\pi,\mu)$.

Lemma 6.3. *(Transitivity) If $(\pi_0,\mu_0) \leq (\pi_1,\mu_1)$ and $(\pi_1,\mu_1) \leq (\pi_2,\mu_2)$ then $(\pi_0,\mu_0) \leq (\pi_2,\mu_2)$.*

Theorem 6.4. *Let e be an expression in the source language. Suppose that $(\{\},\{\}), e \rightarrow_T^* (\pi,\mu), e'$ and $(\pi,\mu), e' \rightarrow_T^* (\pi',\mu'), e''$. Then $(\pi,\mu) \leq (\pi',\mu')$.*

Proof. (Sketch) By induction on the length of the transition sequence taking $(\pi,\mu), e'$ to $(\pi',\mu'), e''$. The base case is trivial. For the inductive step case on the possible modifications the implementation can make to memory and then use transitivity of $\leq$. ☐

Definition 6.4. Suppose A_S, A_T val. We say that $\sigma, A_S \sim (\pi,\mu), A_T$ if A_S, A_T have the same length and for all valid i, $\sigma, \texttt{get}(A_S, i) \rightarrow^* \sigma', v$ and $(\pi,\mu), \texttt{get}(A_T, i) \rightarrow_T^* (\pi',\mu'), v$ for some v int val.

Lemma 6.5. *(Monotonicity) If $\sigma, A_S \sim (\pi,\mu), A_T$ and $(\pi,\mu) \leq (\pi',\mu')$ then $\sigma, A_S \sim (\pi',\mu'), A_T$.*

Because the target language supports concurrency, the memory store might be modified (by other evaluations of `new` and `set`) while `new`, `get`, and `set` are being evaluated. To account for this we define concurrent transition sequences.

Definition 6.5. A *concurrent transition sequence* τ in the target is a left sequence of memories $[(\pi_0,\mu_0), ..., (\pi_{n-1},\mu_{n-1})]$, right sequence of memories $[(\pi'_0,\mu'_0), ..., (\pi'_{n-1},\mu'_{n-1})]$ and expressions $[e_0, ..., e_n]$ with $(\pi_i,\mu_i), e_i \rightarrow_T (\pi'_i,\mu'_i), e_{i+1}$ for all i, and $(\pi'_i,\mu'_i) \leq (\pi_{i+1},\mu_{i+1})$ for all $0 \leq i < n-1$. We say that τ takes $(\pi_0,\mu_0), e_0$ to $(\pi'_{n-1},\mu'_{n-1}), e'_{n-1}$.

Theorem 6.6. *(new) Suppose that for some A_S, A_T val, $\sigma, \mathit{new}(n, v) \rightarrow^* \sigma', A_S$ and $(\pi,\mu), \mathit{new}(n, v) \rightarrow_T^* (\pi',\mu'), A_T$. Then $\sigma', A_S \sim (\pi',\mu'), A_T$.*

[3] Note that the relation is only defined for valid memory states.

Proof. Because the logs are empty, `get` on every index of A_T evaluates to 0. Similarly, `get` on every index of A_S evaluates to 0. □

Theorem 6.7. *(`get`) Suppose that* $\sigma, A_S \sim (\pi, \mu), A_T$. *Fix* i int val. *Suppose that* $\sigma, \mathtt{get}(A_S, i) \to^* \sigma, v$ *where* v int val. *Let* $e_0 = \mathtt{get}(A_T, i)$ *and suppose that* $(\pi, \mu) \leq (\pi_0, \mu_0)$. *Consider a concurrent transition sequence from* $(\pi_0, \mu_0), e_0$ *to* $(\pi'_{n-1}, \mu'_{n-1}), e_n$ *with* e_n int val. *Then* $e_n = v$.

Proof. (Sketch) We case on the line where the implementation of `get` returns. There are 4 cases: lines 56, 24, 26, and 28. In each case, we show that e_n is the same as the value $(\pi, \mu), \mathtt{get}(A_T, i)$ evaluates to. Since $\sigma, A_S \sim (\pi, \mu), A_T$, this is the same value that $\sigma, \mathtt{get}(A_S, i)$ evaluates to, which implies $e_n = v$.

The first case is on line 56 in the implementation of `get`, where guess is returned if the version of the sequence and ArrayData are the same. Since sequence versions are immutable, ArrayData versions are non-decreasing, and sequence versions are $\leq$ array data versions, the sequence and ArrayData versions are also the same on line 54. From the implementation of `get`, if the entire `get` was evaluated atomically on line 54, it would evaluate to *guess*. By lemma 6.5 this means that guess and v are the same.

The other cases on lines 24 and 26 are similar. In the last case, `get` involves a binary search on the logs. The theorem follows for the binary-search case from property (3) in memory transformations (logs can only be extended) and property (4) in valid memories (versions in a log are strictly increasing). □

Theorem 6.8. *(`set`) Suppose that* $\sigma, A_S \sim (\pi, \mu), A_T$. *Fix* i, v int val. *Suppose that* $\sigma, \mathtt{set}(A_S, i) \to^* \sigma', A'_S$ *where* A'_S val. *Let* $e_0 = \mathtt{set}(A_T, i, v)$ *and suppose that* $(\pi, \mu) \leq (\pi_0, \mu_0)$. *Consider a concurrent transition sequence from* $(\pi_0, \mu_0), e_0$ *to* $(\pi'_{n-1}, \mu'_{n-1}), e_n$ *with* e_n val. *Then* $\sigma', A'_S \sim (\pi'_{n-1}, \mu'_{n-1}), e_n$.

Proof. (Sketch) Case on whether the compare and swap in `set` evaluates to true or false.

If it evaluates to false, then the implementation calls `get` at each index of A_T and creates a new ArrayData in μ. In this case $\sigma', A'_S \sim (\pi'_{n-1}, \mu'_{n-1}), e_n$ follows from theorem 6.7.

If it evaluates to true, then from lemma 6.5 A_S and A_T are related when the compare and swap evaluates. In the source, $\mathtt{set}(A_S, i, v)$ evaluates to $A_S[i \mapsto v]$. In the target, suppose that $A_T = \&l$, $\pi[l] = (\lambda, k)$ and $\mu[k] = (V, A, L)$. As argued before, different executions of lines 49 to 51 in the implementation of `set` on the same ArrayData cannot overlap. So in the target, the implementation sets $\mu[k] = (V+1, A[i \mapsto v], L')$ for some L' and returns $\&l'$ such that $\pi[l'] = (V+1, k)$. Going through the definition of $\sim$ this gives us $\sigma', A'_S \sim (\pi'_{n-1}, \mu'_{n-1}), \&l'$. □

A sequence implementation that never terminates would satisfy all the theorems above, but would not correctly implement the sequence specification. As such, the proof of correctness requires proving that the sequence operations have bounded cost.

Theorem 6.9. *(Bounded Cost) There exists* k *such that if a target sequence* A_T *has size* n *then there does not exist* (π, μ) valid *and a concurrent transition sequence of length* $> kn$ *starting at either* $(\pi, \mu), \mathtt{get}(A_T, i)$ *or* $(\pi, \mu), \mathtt{set}(A_T, i, v)$ *for all* i, v int val.

Proof. (Sketch) Because we copy the contents of a sequence after n updates, where n is the size of the sequence, for some k independent of n, `set` involves at most kn steps, and `get` involves at most $k \log n$ steps. □

6.5 Interleaved Cost Bounds

We want to show that work done in the interleaved structural dynamics is an upper bound for the work done in the implementation. We first sketch a linearizability-type result (Herlihy and Wing 1990).

Consider the execution E of a program. We assume a sequentially consistent model of computation. Consider a sequential execution S of instructions in E that produces the same result as E.

Consider any `get` in the sequential execution S. From the previous subsection, the result that the program evaluates to does not depend on how the instructions in the `get` are interleaved. Consider line 56 in `get` where the version of the array is compared with its ArrayData. If the versions match then `get` involves a constant amount of work. Otherwise, in the worst case `get` involves a binary search on n elements, where n is the size of the array. This takes work proportional to $\log n$.

So the execution S is upper bounded (in cost) by an execution where the entire `get` is evaluated atomically at the instruction where the versions of the array and ArrayData are compared, where the `get` has constant work if the version check succeeds and $\log n$ work if the check fails.

Similarly, consider any `set` in the sequential execution S. From the previous subsection, the result that the program evaluates to does not depend on how the instructions in the `set` are interleaved. Consider line 41 in `set` which involves a compare and swap on the ArrayData's version. If the compare and swap is successful then `set` involves a constant amount of work. Otherwise, in the worst case `set` involves copying over n elements, where n is the size of the array. This takes work proportional to n.

So the execution S is upper bounded (in cost) by an execution where the entire `set` is evaluated atomically at the compare and swap instruction. The `set` has constant work if the compare and swap succeeds and n work if the compare and swap fails.

We can trivially assume that `new` is evaluated atomically since none of the effects of `new` are observable until it finishes evaluating, and `new` always takes time proportional to the size of the array being created.

We can then define a relation between sequences in the source and target. σ, A_S in the source and $(\pi, \mu), A_T$ in the target are related if `get` evaluates to the same value at all indices, and A_S is a leaf sequence if and only if $\lambda = V$ where $A_T = \&l$, $\pi[l] = (\lambda, k)$, and $\mu[k] = (V, A, L)$. We can show that the relation is preserved by `new` and `set` and that costs of operations in the source are an upper bound for operations in the target for related sequences.[4] Since the execution was transformed so that `new`, `get`, and `set` evaluate atomically, we can then prove and apply a parametricity theorem to show that programs in the source and target evaluate to the same value.

Converting this argument into a formal proof requires significant work beyond the scope of the paper.

6.6 Parallel Cost Bounds

Next, we sketch the case where we have an unbounded number of processors. In the worst case, `get` involves a binary search on n elements, where n is the size of the array. This takes $\log n$ time on a machine with an unbounded number of processors. In the worst case, `set` involves copying n array elements and n log entries where n is the size of the array. In a machine with an unbounded number of processors, the copying can be done in $\log n$ time. Similarly, `new` can execute in $\log n$ time on a machine with an unbounded number of processors. Note that the implementation

[4] `set` also copies the sequence data once every n times the version number is increased, but the copying is amortized constant time.

of `new`, `get`, and `set` are wait-free so these costs are independent of what other threads are doing.

It follows that we can use the cost dynamics to compute the work W and span S of evaluating an expression e in the source language. We can then use theorem 3.1 to get that the cost of evaluating e on a p-processor machine is $\leq c(\frac{W}{P} + S \log P)$ for some constant c independent of e.

7. Test Results

Besides implementing our functional arrays in SML, we implemented our arrays in Java and compared the performance of functional arrays with regular Java arrays. Java has a relaxed memory consistency model (not a sequentially consistent memory model), so we added memory fences in suitable locations to prevent the compiler and machine from reordering instructions.

We compared the performance of leaf functional arrays and regular Java arrays of size 3,000,000 on a dual-core machine. The arrays occupied 12mb of space, and the machine had 3mb of shared L3 cache, so the entire arrays could not be cached. Since many of our tests involved looping and accessing elements in the array, without performing any useful computations, we disabled compiler optimizations (which optimize away the loops).

We ran each test 5 times, and computed the average time. We do not show standard errors because we were interested in orders of magnitude, however all timings differed from the mean by at most 15% of the mean time.

	Regular	Functional	Slowdown
Seq. reading array	1.35s	1.50s	10.8%
Rand. reading array	8.88s	9.10s	3.4%
Seq. writing to array	2.78s	9.14s	3.3×
Rand. writing to array	5.57s	13.2s	2.4×

Table 1. Speed of leaf functional arrays vs. regular arrays in Java

In the sequential read test, after creating an empty array, we sequentially get elements at indices $0, 1, 2, ...$, starting over at 0 when we reach the end of the array. In the random read test, we repeatedly generate a random number r and get the r^{th} element of the array. We performed 15,000,000 accesses in both of these tests.

In the sequential write test, starting from an empty array, we sequentially set the elements at indices $0, 1, 2, ...$, starting over at 0 when we reach the end of the array. In the random write test, we repeatedly generate a random number r and set the r^{th} element of the array. We performed 5,000,000 writes in both of these tests.

The results suggest that operations on leaf functional arrays are almost as efficient as regular arrays. The additional 2-3 times slowdown in `set` is expected because we incur an additional cache miss when we insert a log entry. Note that similar benchmarks on alternative implementations of functional arrays, for example persistent binary search trees, are likely to be slower by a much larger factor.

Additionally, we compared the time taken to access elements in leaf arrays and interior arrays in a specific benchmark. We wrote 20,000,000 values into random indices of an array of size 2,100,000. We then read 5,000,000 values from random indices of the leaf array and the interior array. Reading from the interior array was $4.5\times$ slower, which is not too much of a slowdown.

We also performed two simple tests to profile multi-threaded accesses in our functional array implementation. In the first test, 2 threads simultaneously accessed 500,000 random elements in an array. The total time taken was $1.77\times$ less than a single thread accessing 1,000,000 random elements in the array. In the second test, 2 threads simultaneously accessed the same element 500,000 times. The total time taken was $1.76\times$ less than a single thread accessing the element 1,000,000 times. The results of the second test are particularly good. We cannot expect a $2\times$ speedup because the element will keep moving between the L1 caches of the two cores. However, the speedup of $1.76\times$ means that the accesses are not serialized (as they would be with a per-element lock).

8. Future Work

As mentioned, the correctness argument omits explicit proof that push operations are never executed concurrently, which is guaranteed by the use of compare-and-swap to ensure exclusive access. Extending the argument to account for the state of the exclusion lock would require a substantial extension to the proof, in particular enriching the invariant to account for the state of the lock. It would be useful to extend the argument to account for exclusion, and to mechanize the proof using verification methods such as those considered in Birkedal et al. (2012); Turon et al. (2013).

We do not at present have formal proofs for the cost bounds of the sequence implementation. Current techniques for analyzing concurrent data structures focus on proving correctness. Enhancing these logics to account for cost is an important direction for future research on the correctness of concurrent implementations of parallel algorithms.

The cost dynamics gives a tight upper bound on the costs in the interleaved structural dynamics. However, the interleaved structural dynamics does not give a tight bound for the cost of the sequence implementation. In the implementation, accessing a value in an interior version involves only constant work if there are a constant number of log entries at that index, but the interleaved structural dynamics charges n work. Extending the cost dynamics to situations that generalize beyond simply separate costs for interior and leaf versions is a possible future direction.

It would also be interesting to extend our methods to consider other data structures, such as unordered sets or disjoint sets, which are often used in parallel programs. Preliminary investigations suggest that similar methods would apply, but it remains to investigate these cases further.

Acknowledgements

We thank Joe Tassarotti for his helpful advice on the correctness proof and Danny Sleator for ideas on how to prove a lower bound on costs for functional arrays. This research is partially supported by the National Science Foundation under grant numbers CCF-1314590, CCF-1408940, CCF-1533858, and the Intel Science and Technology Center for Cloud Computing.

The third author gratefully acknowledges the support of the Air Force Office of Scientific Research through MURI grant FA9550-15-1-0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

References

A. Aasa, S. Holmström, and C. Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988.

Z. M. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler. A call-by-need lambda calculus. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 233–246, 1995.

L. Birkedal, F. Sieczkowski, and J. Thamsborg. A concurrent logical relation. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, pages 107–121, 2012.

G. Blelloch and J. Greiner. Parallelism in sequential functional languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 226–237, 1995.

G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of nesl. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, 1996.

G. E. Blelloch and R. Harper. Cache and i/o efficent functional algorithms. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 39–50, 2013.

A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 26–38, New York, NY, USA, 1989. ACM.

R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, Apr. 1974.

T.-R. Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In *Symposium Proceedings on 4th European Symposium on Programming*, ESOP'92, pages 110–129, London, UK, UK, 1992. Springer-Verlag.

J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1), 1989.

J. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Trans. Program. Lang. Syst.*, 21(2): 240–285, Mar. 1999.

J. C. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proc. Symposium on Logic in Computer Science (LICS)*, pages 333–343, 1990.

P. R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012.

R. Harper. *Practical Foundations for Programming Languages, 2nd edition*. Cambridge University Press, 2016.

M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 276–290, 1988.

M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 351–363, New York, NY, USA, 1986. ACM.

P. Hudak and A. G. Bloss. The aggregate update problem in functional programming systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 300–314, 1985.

G. F. Italiano and N. Sarnak. *Fully persistent data structures for disjoint set union problems*, pages 449–460. Springer Berlin Heidelberg, 1991.

C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, Oct. 1983.

E. Moggi. Computational lambda-calculus and monads. In *Proc. Symposium on Logic in Computer Science (LICS)*, pages 14–23, 1989.

A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 343–356, 2013.

P. Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag.