Parallel Functional Arrays

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

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

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

Abstract

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

We embed sequences in a language that supports fork join parallelism. Because we support fork-join parallelism, the cost of programs is non-deterministic and depends on the order in which operations are interleaved. This is because the ordering of operations can affect whether a sequence is a leaf or interior. Consequently the concurrent costs of such programs are difficult to analyze. Our main result is the derivation of a deterministic evaluational dynamics which makes analyzing the costs much easier. Our theorems are not array-specific and can be applied to any data type with different costs for operating on interior and leaf versions.

We give a wait-free concurrent sequence implementation which requires constant work for accessing and updating leaf arrays and bounded work for operations on interior arrays. We sketch out our proof of correctness for the array implementation. The key advantages of our sequence 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 like 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 (pun not intended). It is easy to implement such sequences with constant time random access reads (using arrays stored as contiguous memory), or to implement with logarithmic time reads and updates using balanced trees, but it seems that getting both in constant time cannot be achieved without some form of 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, for example, use STMonads with an array to implement depth first search (DFS) in linear work (King and Launchbury 1995). 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 using these approaches is that they are basically implementing a second imperative 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 they 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 that at certain points in the code a count must be one, and therefore it is safe to replace an 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 effecting the value of other nodes. Dietz showed that arrays can be maintained fully persistently using $O(\log \log n)$ time per operation (read or write). Holmstrom, Hughes and others (see (Aasa et al. 1988)) suggested storing a version tree by keeping the full array at the root, and a each node represents an update. Chuang's approach (Chuang 1992) supports $O(1)$ accesses and insertions to the most recent version of arrays, however accessing old versions of the array takes $O(n)$ work which is often impractical. O'Neill and Warren (O'Neill 2000) describe various improvements. The problem with these approaches is that they can be very complicated, and only efficient in certain cases. Furthermore it can be hard to reason about the performance.

None of the existing approaches properly support concurrent operations on functional arrays. O' Neill 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.

Our Approach: Sequences

In this paper we present an approach for efficiently supporting functional arrays (sequences). It uses some ideas from the previous approaches but unlike the previous approaches it supplies a well defined cost dynamics, and supports parallelism, and without language extensions. More specifically the approach has the following important features.

1. It has fully functional value dynamics—i.e., when not considering costs, sequences act no differently than purely functional sequences.
2. It requires no changes in existing languages and no special types, syntactic extensions, etc. Programs can be efficient—i.e., constant time reads and writes—when using a completely standard functional programming style.
3. The approach supports nested parallelism—sequences can be passed to parallel calls and safely updated and read, again with a purely functional dynamics.
4. 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.
5. We describe a cost-bounded implementation which maps costs in the model to costs on either a sequential or parallel RAM (PRAM).
6. Although accessing old versions of a sequence is more expensive than accessing new versions (as defined in the cost dynamics), reading old versions never costs more than $O(\log n)$ work.

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

Although our value dynamics is purely functional (and compositional) our cost dynamics is not—they require passing a store and modeling costs based 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-value only differ in their costs, and a store is required to model the difference (Ariola et al. 1995).

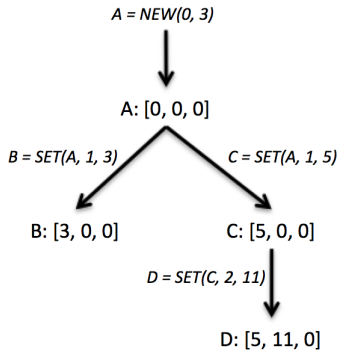

Figure 1. Example usage of sequences

In this paper 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 of A is replaced by v (in the following discussion n refers to the length of an 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 the figure, after all functions are applied, sequences A and C are *interior nodes*, and sequences B and D are *leaves* of the version tree. In our cost dynamics applying GET and SET to sequences at the leafs takes constant work, but applying them to interior nodes is more expensive.

We start with a simple pure call-by-value lamba calculus extended with sequences. To define abstract costs and show the relationship to concrete cost (time) on a parallel machine we use two levels of cost dynamics: a structural cost dynamics and a evaluation cost dynamics. Our structural cost dynamics is a single step dynamics and is closer to the implementation. The dynamics defines the work by summing the cost per transition across an arbitrary interleaving of the potential transitions (due to concurrency), and defines the span by allowing all potentially parallel transitions to proceed together. We prove a mapping of the cost in the structural dynamics to time on a P processor machine, assuming given costs for the sequence functions. The cost is non-deterministic due to different possible interleavings.

The evaluation dynamics is a higher-level big-step dynamics more appropriate for a programmer to reason about costs. In particular the costs in the evaluation dynamics are deterministic independent of interleaving. We prove that the costs in the evaluation dynamics gives a tight worst case upper bound on the cost of the underlying structural dynamics, allowing for any interleaving.

We describe the implementation in an impure language—our pure language extended with references and an atomic compare-and-swap on those references. To implement sequences, we maintain at each leaf of the version tree a mutable array of the most recent values. We also keep with each location a change-log that keeps track of values at interior nodes of the version tree. Applying GET to a leaf only requires a lookup in the array. Applying SET to a leaf only requires updating the array, and adding an element to the change log for the old version. Applying GET to an interior nodes requires a binary search on the change log, which can be bounded by $O(\log n)$ time. To ensure that change logs do not get too large, whenever the total size across all change logs reaches n, the array is copied. This requires $O(n)$ time, but can be amortized against

the updates. Applying SET to an interior node requires creating a new array, and copying values to it. This requires $O(n)$ time.

We give a wait-free concurrent implementation 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. All we require is that memory reads, writes and compare-and-swap are atomic. 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.

To evaluate our approach in practice we ran a variety of experiments on a preliminary concrete implementation. The experiments show that accesses 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.

Our results on cost dynamics can be easily 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 coming up with 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:

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

The constants c contains the usual arithmetic types, such as the natural numbers and numerical functions. The expression $(e_1||e_2)$ 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. $\text{NEW}(n, v)$ evaluates to a sequence of size n with the value v at each index. $\text{GET}(A, i)$ evaluates to the i^{th} element of sequence A. $\text{SET}(A, i, v)$ evaluates to a new sequence where the i^{th} element of A is substituted with v.

3. Structural Dynamics

Our goal is to bound the runtime cost of evaluating a program (in our language) on a p processor machine. For this purpose, we define two separate structural (small step) cost dynamics for our language: the *idealized parallel dynamics* and the *interleaved dynamics*.

The parallel dynamics defines the parallel *span* (or depth) of a computation, which is intuitively the number of parallel steps given an unbounded number of processors. It is a pure dynamics and allows all parallel expressions to proceed on the same step.

The interleaved dynamics defines the total *work* performed by a computation, which can be thought of as the number of instructions used by the computation (within constant factors). The dynamics is not pure, requiring a store to keep track 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. In Section 4 we give a deterministic cost dynamics and show that it provides an upper bound on the work over all possible interleavings.

The two dynamics are useful together because we can bound the runtime on any fixed number of processors based on the work and span.

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

$$\frac{e_1 \rightarrow_{par} e_1' \quad e_2 \rightarrow_{par} e_2'}{e_1||e_2 \rightarrow_{par} e_1'||e_2'} \text{ (fork)}$$

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

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

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

Figure 2. Example rules in the parallel dynamics for our language.

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:

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

Let $\overline{\text{GET}}(A, i)$ denote $A[i]$, $\overline{\text{SET}}(A, (i, v))$ denote $A[i \mapsto v]$ (which denotes the sequence obtained if the value at index i of A is replaced with v), and $\overline{\text{NEW}}((n, v))$ denote a sequence of size n with the value v at each index. Intuitively, $\overline{\text{GET}}$, $\overline{\text{SET}}$, and $\overline{\text{NEW}}$ represent abstract versions of GET, SET, and NEW.

We show the rules for get and fork-join in Figure 2. 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 into the store σ and V is a list of the elements in the sequence.

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

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

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

$$\frac{a \text{ val} \quad l \notin L(\sigma)}{\sigma, \text{NEW}(a) \to \sigma[l \mapsto +], \overline{\text{NEW}}(a), n(a)} \text{ (new)}$$

$$\frac{A = (l, V) \quad \sigma[l] = + \quad a \text{ val}}{\sigma, \text{GET}(A, a) \to \sigma, \overline{\text{GET}}(V, a), g_l(V)} \text{ (get-leaf)}$$

$$\frac{A = (l, V) \quad \sigma[l] = - \quad a \text{ val}}{\sigma, \text{GET}(A, a) \to \sigma, \overline{\text{GET}}(V, a), g_i(V)} \text{ (get-interior)}$$

$$\frac{A = (l, V) \quad \sigma[l] = + \quad l' \notin L(\sigma) \quad a \text{ val}}{\sigma, \text{SET}(A, a) \to \sigma[l \mapsto -, l' \mapsto +], (l', \overline{\text{SET}}(V, a)), s_l(V)} \text{ (set-leaf)}$$

$$\frac{A = (l, V) \quad \sigma[l] = - \quad l' \notin L(\sigma) \quad a \text{ val}}{\sigma, \text{SET}(A, a) \to \sigma[l' \mapsto +], (l', \overline{\text{SET}}(V, a)), s_i(V)} \text{ (set-interior)}$$

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

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

$$\frac{v_1, v_2 \text{ val}}{\sigma, (v_1 || v_2) \to \sigma, (v_1, v_2), 1} \text{ (join)}$$

Figure 3. The interleaved dynamics for our language. We omit rules that involve stepping the arguments to GET, SET, NEW, and other language constructs.

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 $g_l(V)$ and $g_i(V)$ be the work of GET applied to V if it is a leaf or interior sequence, respectively, and $s_l(V)$ and $s_i(V)$ be the work of SET applied to V if it is a leaf or interior sequence. Let $n(a)$ be the work of evaluating NEW(a). We assume that $g_l(V) \leq g_i(V)$ and $s_l(V) \leq s_i(V)$ (operating on leaf sequences is cheaper that operating on interior sequences).

The dynamics is given in figure 3. 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 for GET and SET depends on whether the sequence is a leaf or interior.

The rules for fork-join are non-deterministic—either side of the fork can take a step. 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.

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 $0 \leq i < n, \sigma_i, e_i \to \sigma_{i+1}, e_{i+1}, w_{i+1}$. We say that T takes σ_0, e_0 to σ_n, e_n and has length n and denote this by $\sigma_0, e_0 \to^n \sigma_n, e_n$.

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

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. In particular, a sequence can be used concurrently by multiple parallel expressions and the work could depend on the order in which the instructions are interleaved. Figure 4 shows 3 ways a leaf sequence A might be used. 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 $g_l(A)$ if and only if they execute before the call to SET on the right branch, otherwise they would have work $g_i(A)$.

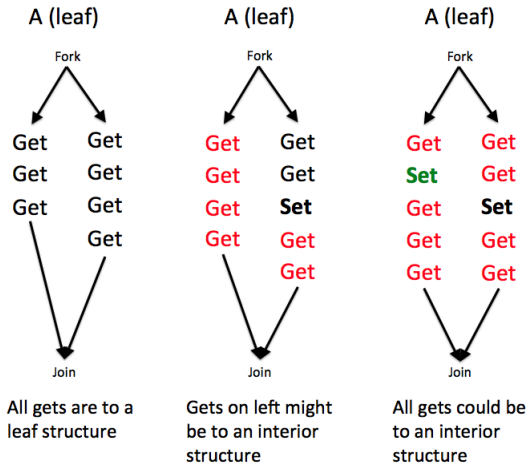

Figure 4. Different ways a sequence might be used in a fork join

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 $g_l(V)$, $g_i(V)$, $s_l(V)$, $s_i(V)$, and $n(a)$ are upper bounds on the number of instructions needed by the implementation for each of the corresponding functions. 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.

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.

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. 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*

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

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. Evaluational Cost Dynamics

The evaluational (big-step) 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 evaluational dynamics. In section 5, we prove that the costs in the evaluational dynamics are a tight upper bound for the costs in the structural dynamics. Our 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, we first compute the worst-case cost of each fork separately. Then for sequences that are used in both forks, we add additional work at the join point. To compute the additional work at the join point, we keep track of the number of GETs to leaf values on each side of the fork. If the value was SET on the other side of the fork, then the SET might have happened before the GETs, so we need to charge additional work for the GETs.

Our evaluational dynamics is defined by the following judgement, 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 into the store δ and V is a list of the elements in the sequence. The store is a mapping from labels to $(+/-, c)$, where c represents the number of leaf GETs on the value indexed by the label. $L(\delta)$ denotes the set of labels in the store δ.

The rules in the evaluational dynamics are given in figure 5. Note that GET costs $g_l(V)$ instead of $g_i(V)$ on a leaf value V but we increment the counter of leaf GETs in the store.

The fork-join rule is the most interesting and requires a few definitions. Suppose we have expression $(e_L||e_R)$ 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 GETs on one side of the fork remain leaf GETs iff there were no calls to SET on the other side of the fork.

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

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

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

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

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

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

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

$$\frac{A = (l, V) \quad a \text{ val} \quad \delta[l \mapsto (+, c)] \quad l' \notin L(\delta)}{\delta, \text{SET}(A, a) \Downarrow \delta[l \mapsto (-, c), l' \mapsto (+, 0)], (l', \overline{\text{SET}}(V, a)), s_l(V), 1} \text{ (set-leaf)}$$

$$\frac{A = (l, V) \quad a \text{ val} \quad \delta[l \mapsto (-, c)] \quad l' \notin L(\delta)}{\delta, \text{SET}(A, a) \Downarrow \delta[l' \mapsto (+, 0)], (l', \overline{\text{SET}}(V, a)), s_i(V), 1} \text{ (set-int)}$$

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

$$\frac{A = (l, V) \quad a \text{ val} \quad \delta[l \mapsto (+, c)]}{\delta, \text{GET}(A, a) \Downarrow \delta[l \mapsto (+, c + 1)], \overline{\text{GET}}(V, a), g_l(V), 1} \text{ (get-leaf)}$$

$$\frac{A = (l, V) \quad a \text{ val} \quad \delta[l \mapsto (-, c)]}{\delta, \text{GET}(A, a) \Downarrow \delta, \overline{\text{GET}}(V, a), g_i(V), 1} \text{ (get-interior)}$$

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

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

Figure 5. Rules for the evaluational dynamics. w' and δ' in the fork-join rule are defined in the paper.

EXTRAWORK gives the additional cost incurred if a sequence was modified on either side of a fork-join. If the value was interior before the fork-join, then all functions incurred their maximal cost and there is no additional cost. Otherwise, leaf GETs on one side of the fork are charged g_i work iff the other side of the fork called SET. Additionally, if both sides of the fork called SET, then one of the SETs came first and has work s_l and the subsequent SET has work s_i. We abuse notation so that g_l, g_i, s_l, s_i directly take in a label l instead of the corresponding sequence.

$$\begin{aligned}\text{EXTRAWORK}(l,(s,c),(s_L,c+c_L),(s_R,c+c_R)) = \\ ((\neg s_R)c_L + (\neg s_L)c_R)(g_i(l) - g_l(l)) + \\ (\neg s_R)(\neg s_L)(s_i(l) - s_l(l))\end{aligned}$$

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

Then, the fork-join cost dynamics are:

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

5. Cost Proofs

We show that the work computed by the evaluational 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).

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 $w_{i+1} = g_l(l)$. The number of cheap GETs on l in T is denoted by $SC_l(T)$.

Definition 5.2. Unlike the store in the evaluational dynamics, the store in the structural dynamics only stores whether each structure is $+$ (leaf) or $-$ (interior). $sign(\delta)$ 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 GETs in going from δ to δ' in the evaluational 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 length m that $\delta, e \Downarrow \delta', e', w, s$. Let R be an arbitrary labeling. Then there exists a derivation of length 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 evaluational 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 casing on each rule in the structural dynamics, and noting that transitions of expressions never depend on the values in the store (the dynamics are 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

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

The following hold:

1. *For some relabeling R of $L(\delta_E) \setminus L(\delta)$, $S_n = sign(R(\delta_E))$ 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:*

$$\begin{aligned}w_E + \sum_{l \in L(\delta_E)} EC_l(\delta, \delta_E)(g_i(l) - g_l(l)) = \\ W(T) + \sum_{l \in L(\delta_E)} SC_{R(l)}(T)(g_i(R(l)) - g_l(R(l)))\end{aligned}$$

Proof. By induction on the length of the shortest derivation in the evaluation dynamics. We prove the theorem for 2 of the rules in the evaluation dynamics: get-leaf and fork-join. The other cases follow from a similar line of reasoning.

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

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

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

$$\delta_R, \text{GET}(e'_L, e'_R) \Downarrow \delta_E, e_E, g_l(l')$$

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

$$\delta, \text{GET}(e_L, e_R) \Downarrow \delta_E, e_E, w_E$$

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

Note that 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: Suppose that in $T_{0,m}$, for $0 \leq i \leq m$, $e_i = get(e_{L,i}, e_R)$. Consider a new projected transition sequence $T'_{0,m}$ with states $[(S_0, e_{L,0}), ..., (S_m, e_{L,m})]$ and costs the same as $T_{0,m}$: $[w_1, ..., w_m]$. It is easy to verify that $T'_{0,m}$ is a valid transition sequence starting at $(S_0, e_{L,0})$ and is maximal.

$S_0 = sign(\delta)$ and $e_{L,0} = e_L$ so we can apply the IH to $T'_{0,m}$. For some relabeling R of labels in $L(\delta_L)/\delta$, $s_m = sign(R(\delta_L))$ and $e_{L,m} = R(e'_L) \Rightarrow e'_m = (e_{L,m}, e_R) = (R(e'_L), e_R)$. WLOG suppose that δ_L was labeled such that the above hold (we can do this because of lemma 5.1). Note that in subsequent parts of the proof, we omit creating the projected transition sequence and apply the IH directly to $T_{0,m}$ when needed.

Using similar logic on $T_{m,n-1}$ we get that for some relabeling of δ_R/δ_L, applying that relabeling to δ_R and (e'_L, e'_R) gives us $e_{n-1} = (e'_L, e'_R)$ and $S_{n-1} = sign(\delta_R)$. WLOG suppose that δ_R was labeled so that the above holds.

By combining the relabelings of δ_L/δ and δ_R/δ_L and applying that to (e'_L, e'_R) and δ_R, we get that $e_{n-1} = get(e'_L, e'_R)$ and $S_{n-1} = sign(\delta_R)$. Note that get does not change the sign of any label in the store in either the structural or the evaluation dynamics, so $sign(\delta_E) = sign(\delta_R) = S_{n-1} = S_n$. Further, the structural and evaluational dynamics apply get in the same way, so $e_n = e_E$. Therefore the first part of this theorem holds. WLOG suppose that the stores and expressions were relabeled s.t. the first part holds.

Part 2: From lemma 5.1, the length of the shortest derivation for the relabeled evaluational dynamics does not change, so we can apply the inductive hypothesis even after the relabeling. Applying the IH to $T_{0,m}$, for all $l \in \delta_L$, $EC_l(\delta, \delta_L) \leq$

$SC_l(T'_{0,m}) = SC_l(T_{0,m})$. If $l \in \delta_R$ but $l \notin \delta_L$ then $EC_l(\delta, \delta_L) = SC_l(T_{0,m}) = 0$ (intuitively the label did not exist and so did not have any cheap gets), so in particular, for all $l \in \delta_R$, $EC_l(\delta, \delta_L) \leq SC_l(T_{0,m})$. Applying the IH to $T_{m,n-1}$ we get that for all $l \in \delta_R$, $EC_l(\delta_L, \delta_R) \leq SC(T_{m,n-1})$.

We note that EC and SC are additive, that is,

$$EC_l(\delta, \delta_R) = EC_l(\delta, \delta_L) + EC_l(\delta_L, \delta_R)$$

$$SC_l(T_{0,n-1}) = SC_l(T_{0,m}) + SC_l(T_{m,n-1})$$

This implies that for all $l \in \delta_R$, $EC_l(\delta, \delta_R) \leq SC_l(T_{0,n-1})$.

Now, let $e'_L = (l', V)$ with $l' \in \delta_R$. Then, $EC_{l'}(\delta, \delta_E) = EC_{l'}(\delta, \delta_R)+1$ and $SC_{l'}(T) = SC_{l'}(T_{0,n-1})+1$, so $EC_{l'}(\delta, \delta_E) \leq SC_{l'}(T)$. If $l \neq l'$ then $EC_l(\delta, \delta_E) = EC_l(\delta, \delta_R)$ and $SC_l(T) = SC_l(T_{0,n-1})$, so $EC_l(\delta, \delta_E) \leq SC_l(T)$. So the second part of the theorem holds.

Part 3: The third part of the theorem follows by similarly applying the IH on $T_{0,m}$ and $T_{m,n-1}$, and noting that $W(T) = W(T_{0,m}) + W(T_{m,n-1}) + g_l(l')$.

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

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

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

which gives us

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

with w' and δ' defined in terms of EXTRAWORK and COMBINE as given in the evaluational dynamics.

In T, for all $0 \leq i \leq n$ let $e_i = (e_i^L||e_i^R)$. Let I_L be the sequence of all $0 \leq i \leq n-1$ where transitioning from e_i to e_{i+1} involved stepping the left argument. Let m be the length of I_L. Similarly define I_R for the right argument, and let k be the length of I_R.

Let A be the sequence of e^Ls corresponding the indices in I_L, and add e_n^L at the end of A. Similarly, let B be the sequence of e^Rs corresponding the indices in I_R, adding e_n^R at the end.

Let $S_0^L = S_0 = sign(\delta)$. By inductively applying lemma 5.2, there exists $S_1^L, ...S_m^L$ s.t. $T_L = [(S_0^L, A_0), ..., (S_m^L, A_m)]$ is a transition sequence. T_L is maximal, because if A_m in T_L can take a step then e_n^L in T can take the corresponding step. Similarly, let $S_0^R = S_0 = sign(\delta)$. Then, there exists $S_1^R, ...S_k^R$ s.t. $T_R = [(S_0^R, B_0), ..., (S_k^R, B_k)]$ is a maximal transition sequence.

Applying the IH on T_L, we get a relabeling R_L on $L(\delta_L) \setminus L(\delta)$ s.t. $R_L(v_L) = A_m = e_n^L$ and $sign(R_L(\delta_L)) = S_m^L$. Similarly, we get a relabeling R_R on $L(\delta_R) \setminus L(\delta)$ s.t. $R_R(v_R) = B_k = e_n^R$ and $sign(R_R(\delta_R)) = S_k^R$.

By inducting on the product mk, we can show that the new labels produced in T_L and T_R do not overlap, that is: $(L(S_m^L) \setminus L(S_0)) \cap (L(S_k^R) \setminus L(S_0)) = \emptyset$. Compose the relabelings to get relabeling R.

Applying lemma 5.1 and the fork join rule, we get,

$$\delta, R(e_L) \Downarrow R(\delta_L), R(v_L), w_L$$

$$\delta, R(e_R) \Downarrow R(\delta_R), R(v_R), w_R$$

which gives us

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

with $R((v_1||v_2)) = (R(v_1)||R(v_2)) = (e_n^L||e_n^R)$. To reduce clutter, WLOG suppose that the stores and expressions in the evaluational dynamics were labeled as such. By lemma 5.1 we can apply the IH on the relabeled evaluational dynamics since the length of the shortest derivation is invariant under relabelings.

Lemma 5.4. *$l \in S_n$ if and only if either $l \in S_m^L$ or $l \in S_k^R$*

Proof. If $l \in S_0$ the claim holds trivially. Otherwise,

($\Rightarrow$) By considering the step in T that first introduced l, and examining the corresponding step in either T_L or T_R.

($\Leftarrow$) By considering the step in T_L (or T_R) that first introduced l and examining the corresponding step in T. □

Lemma 5.5. *$S_n[l] = -$ if and only if either $l \in S_m^L$ and $S_m^L[l] = -$ or $l \in S_k^R$ and $S_k^R[l] = -$*

Proof. Similar to lemma 5.4. □

Part 1: We claim that the set of labels in δ' and S_n are the same. From the evaluational dynamics, $L(\delta') = L(\delta_L) \cup L(\delta_R)$. Since $sign(\delta_L) = S_m^L$ and $sign(\delta_R) = S_k^R$, $L(\delta_L) = L(S_m^L)$ and $L(\delta_R) = L(S_k^R)$. From lemma 5.4, $L(S_n) = L(S_m^L) \cup L(S_k^R) = L(\delta')$ which proves the claim.

To show that $sign(\delta') = S_n$ we case on arbitrary $l \in L(\delta')$ and show that $sign(\delta')[l] = S_n[l]$:

Case $l \in L(\delta)$:

If $\delta[l] = (-, \_)$ then $S_0[l] = -$. The evaluational and structural dynamics do not change $-$ to $+$ so $\delta'[l] = (-, \_)$ and $S_n[l] = -$ so $sign(\delta')[l] = S_n[l]$.

Suppose $\delta[l] = (+, \_)$, in which case $S_0[l] = +$. If $\delta'[l] = (+, \_)$, then from the way combine works, $\delta_L[l] = (+, \_)$ and $\delta_R[l] = (+, \_)$. From the IH $S_m^L[l] = +$ and $S_k^R[l] = +$. From lemma 5.5 $S_n[l] = + = sign(\delta')[l]$.

On the other hand, if $\delta[l] = (+, \_)$ (in which case $S_0[l] = +$) but $\delta'[l] = (-, \_)$, then from the way combine works, either $\delta_L[l] = (-, \_)$ or $\delta_R[l] = (-, \_)$. WLOG suppose that $\delta_L[l] = (-, \_)$. From the IH, $S_m^L[l] = -$. From lemma 5.5 $S_n[l] = - = sign(\delta')[l]$.

Case $l \in L(\delta_L), l \notin L(\delta)$: Then $l \notin \delta_R$ so $\delta'[l] = \delta_L[l]$. From IH, $sign(\delta_L) = S_m^L$. Since $l \notin \delta_R$, $l \notin S_k^R$ so by applying lemma 5.5, $S_m^L[l] = S_n[l]$, as desired.

The case where $l \in L(\delta_R), l \notin L(\delta)$ is symmetric.

Finally, we note that $(v_1||v_2) = (e_n^L||e_n^R)$ from the way v_1, v_2 were relabeled.

Part 2:

We case on arbitrary $l \in \delta'$.

Case $l \in L(\delta)$:

If $\delta[l] = (-, c)$ then $EC_l(\delta, \delta') = 0 = SC_l(T)$.

Else suppose $\delta_L[l] = (+, c_1)$ and $\delta_R[l] = (+, c_2)$. Then $SC_l(T) = SC_l(T_L) + SC_l(T_R) \geq EC_l(\delta, \delta_L) + EC_l(\delta, \delta_R) = EC_l(\delta, \delta')$.

Else suppose $\delta_L[l] = (+, c_1)$ and $\delta_R[l] = (-, c_2)$. We can show that all the cheap GETs in T_R are cheap in T. So $SC_l(T) \geq SC_l(T_R) \geq EC_l(\delta, \delta_R) = EC_l(\delta, \delta)$.

The case where $\delta_L[l] = (-, c_1)$ and $\delta_R[l] = (+, c_2)$ follows by symmetry.

Finally, if $\delta_L[l] = (-, c_1)$ and $\delta_R[l] = (-, c_2)$, then $SC_l(T) \geq 0 = EC_l(\delta, \delta')$.

Case $l \in L(\delta_L), l \notin L(\delta)$ Then $EC_l(\delta, \delta') = EC_l(\delta, \delta_L)$ since $l \notin \delta_R$. By IH, $EC_l(\delta, \delta_L) \leq SC_l(S_0, S_m^L)$. Since $l \notin S_k^R$, $SC_l(S_0, S_m^L) = SC_l(S_0, S_n)$ and so the claim holds.

The case where $l \in L(\delta_R), l \notin L(\delta)$ is similar.

Part 3: We omit the formal proof of this part because it contains tedious algebraic details but the key ideas are described below.

In the conservation expression, each GET on each sequence (l, V) in both the evaluational and structural dynamics is charged $g_l(V)$ work (considering the sum from both terms). In particular, note that the formula adds an additional $g_l(V) - g_l(V)$ cost for cheap GETs. Since the number of GETs is the same, the total cost of GETs is the same in both dynamics.

The number of SETs that are charged $s_l(V)$ is the same for both the evaluational and structural dynamics. This can be shown by

casing on the signs of the sequence on both sides of the fork. If and only if SET was called on a sequence on both sides of the fork, one of the SETs will cost $s_i(V)$. Since the number of SETs is the same, the total cost of SETs is the same in both dynamics. □

Theorem 5.6. *(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 $\sum_{l \in \delta_E} EC_l(\delta, \delta_E)(g_i(l) - g_l(l)) \leq \sum_{l \in \delta_E} SC_l(T)(g_i(l) - g_l(l))$. But then from the conservation of sum of work and cheap gets, $w_E \geq W(T)$. □

Theorem 5.7. *(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* $2W(T) \geq w_E$.

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

Case fork-join: Suppose $e_0 = (e_L||e_R)$ 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

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

From the IH, 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 sequence 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).

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

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

The function DOUBLE-SETS computes whether a sequence 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 $2W(T) \geq w_E$.

WLOG suppose $c_R \geq c_L$. In the evaluational 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 5.8. *(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 evaluational dynamics. □

When evaluating an expression e in our language, the stores are initially empty. In particular, the signs of the store in the evaluational 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 computed by the evaluational 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 evaluational dynamics gives an upper bound for the execution time of the expression.

6. Implementation

In this section we give our implementation of functional arrays (sequences). We show the correctness of our implementation. We show that GET and SET on leaf sequences require constant work, and GET and SET on interior sequences require bounded work. As such, our implementation matches the structural dynamics of a versioned data structure, and we can use the evaluational dynamics we derived to analyze costs of parallel programs involving sequences.

We give SML-like code for our implementation. In our implementation we construct and manipulate mutable arrays (henceforth called arrays). We assume that an array of size n can be created with n work on a single processor machine and $\log n$ work on a machine with an unbounded number of processors. new(n, v) creates a new array of size n with v at each index. 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 with constant work. update(A, i, v) mutates index i of array A to have value v with constant work.

Our implementation assumes an atomic compare and swap function CMPSWAP. CMPSWAP$(a$: int ref, v : int, v' : int) : bool is a single atomic operation. 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 our proofs of correctness, we assume a sequentially consistent memory model. When analyzing costs, we assume that all processors are synchronized with respect to a global clock. At each time step, as many processors as possible execute a single pending instruction.

6.1 Pusharray Implementation

We use an auxiliary data structure, pusharrays, to store log entries. Conceptually, pusharrays are lists that support 3 functions. Suppose we have a pusharray A. PUSH(A, e) inserts entry e, SIZE(A) returns the number of entries inserted, GET(A, i) returns the i^{th} entry inserted. Note that GET is only defined between indices 0 and SIZE$(A) - 1$. All operations have amortized constant work.

Pusharrays can be used semi-concurrently. At most one thread can execute instructions in PUSH at any time, but multiple threads can call SIZE and GET. The type definition for pusharrays is given below.

```
1 type capacity = int
2 type size = int
3 type 'a pusharray = (capacity × size
4                                × 'a array) ref
```

The pusharray's array stores the entries inserted into the pusharray. Size represents the number of entries inserted and capacity represents the size of the array. If the pusharray does not have the capacity for more insertions, it copies all entries to an array that is twice as large. The implementation for PUSH is given below.

```
1  val push : 'a pusharray × 'a → unit
2  fun push(A as ref(c, s, D), v) =
3    if c = s then
4      let val c' = 2 × c
5          val D' = Array.new(c', v)
6      in copyArray(D, D');
7          update(D', s, v);
8          A := (c', s + 1, D')
9      end
10   else
11     update(D, s, v);
12     A := (c, s + 1, D)
```

SIZE simply returns the size attribute of the pusharray, and GET(A, i) returns the i^{th} index in the pusharray's data array. Intuitively, the reason that pusharrays are needed for our sequence implementation is that they are linearizable (Herlihy and Wing 1990) given that different calls to PUSH (on the same pusharray) do not overlap.

6.2 Sequence Implementation

Our sequence implementation keeps a value array of the most recent values, which represents the values at a leaf node of the version tree. For each index of the array we store a change-log that keeps track of values at interior nodes of the version tree. The type definition for sequences is given below.

```
1  type version = int
2  type 'a logs = ('a pusharray) array
3  type 'a arraydata = version ref × 'a array
4                                  × 'a logs
5  type 'a sequence = version × 'a arraydata
```

Note that multiple sequences can reference the same arraydata. Figure 6 visualizes a newly created sequence A.

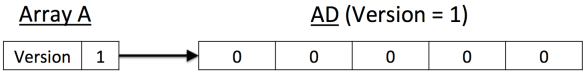

Figure 6. A = NEW(5, 0)

The implementation of NEW is given below.

```
1  val new : int × 'a → 'a sequence
2  fun new(size, init) =
3    (1, (ref 1, Array.new(size, init),
4      tabulate(size,
5        fn _ ⇒ Pusharray.new(0, init))))
```

SET 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 is successful (which means the sequence is a leaf sequence), a log entry is inserted and the value array is mutated. Otherwise, the values in the sequence are copied over to produce a new arraydata. The new arraydata will have empty logs at each index. Note that the values are not directly copied into the new arraydata from the value array, we need to call GET at each index of the sequence when getting the values.

The ordering of the instructions in SET is critical. If the vals array is modified before the log entry is inserted then a GET evaluated between the 2 instructions could evaluate to the wrong value.

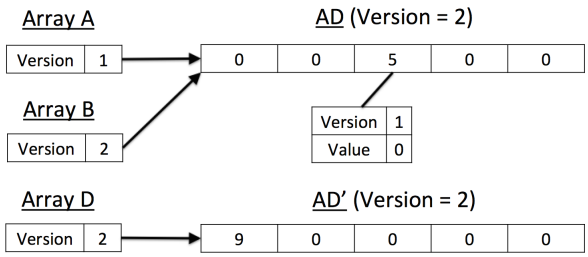

Figure 7. B = SET$(A, 2, 5)$ changes the value array and adds a log entry. Then D = SET$(A, 0, 9)$ creates AD' because A is interior.

Figure 7 visualizes SETs on interior and leaf sequences.
The implementation of SET is given below.

```
1  val set : 'a sequence × int × 'a → 'a sequence
2  fun set((V, (Vr, A, L)), i, v) =
3    if not(cmpswap(Vr, V, (V + 1))) orelse
4       (!Vr mod length(A) = 0) then
5      let val A' = getFarrayValues((V, (Vr, A, L)))
6          val L' = emptyLogs(size(A))
7      in update(A', i, v); (1, (ref 1, A', L')
8      end
9    else
10     Pusharray.push(sub(L, i), (V, sub(A, i)));
11     update(A, i, v);
12     (V + 1, (Vr, A, L))
```

GET(A, i) first loads the value v at the i^{th} index of the value array. If the sequences' version and corresponding arraydata's version match (meaning that the sequence is a leaf sequence), then GET evaluates to v. If the log at index i is empty or all versions in the log at index i are less than the sequence version, then GET evaluates to v. Otherwise, GET binary searches the log at index i for the value corresponding to the least version $\geq$ the sequence's version.

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.

The implementation of GET is given below.

```
1  val get : 'a sequence × i → 'a
2  fun get((V, (Vr, A, L)), i) =
3  let val guess = sub(A, i)
4      val l = sub(L, i)
5      val n = Pusharray.size(l)
6  in if V = !Vr then guess
7      else if n = 0 then guess
8      else
9          let val (V', _) = Pusharray.get(l, n − 1)
10         in if V' < V then guess
11           else valueAtLeastVersionGeq(l, V)
12         end
```

6.3 Target Language

For our proofs we give a more formal definition of the target language in which we implement sequences[1]. The target language

[1] Our target language is different from the SML-like code used in the code samples. This for clarity of our code samples. It is easy to translate our code samples to our 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 the memory 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 (V, A, L) 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, pushing a log entry, and compare and swap. Let a_n denote an array of n as and let $\{\}$ denote an empty (int $*$ int) pusharray.

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

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

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

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

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 $C(\text{NEW}), C(\text{GET}), C(\text{SET})$ 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. GET is simply substituted with its implementation. For SET, we use a SET' "wrapper" so that the sequence SET evaluates to can be added to the store π in the end-set rule. We keep track of the sequences created so we can show invariants on sequences in our proof of correctness.

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

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

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

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

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 our proofs.

6.4 Proof of Correctness

The full correctness proof follows 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. We give a rely-guarantee style argument (Jones 1983) for the correctness of our implementation (for integer sequences).

We show 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 follow from 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. (Outline) 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 $(\{\}, \{\})$.

Our IH is that that the theorem holds for all transition sequences of length n. We then consider an arbitrary transtion sequence of length $n + 1$. From the IH, the first n steps leads to a valid memory state. We then case on all possible modifications that the $n + 1^{\text{th}}$ 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 10 to 12 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) Consider $(\pi, \mu), (\pi', \mu')$ *valid* [2]. Consider arbitrary label l with $l \in L(\pi)$ and $l \in L(\pi')$. Suppose that $\pi[l] = (\lambda, k)$ and $\pi'[l] = (\lambda', k')$. We say that $(\pi, \mu) \leq (\pi, \mu)$ at l if the following hold.

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

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, $\text{LEN}(L[i]) \leq \text{LEN}(L'[i])$ and for all i, j with $0 \leq j < \text{LEN}(L[i])$, $L[i][j] = L'[i][j]$. Intuitively, this means that logs can only be extended, existing entries cannot be overwritten.
4. If $\text{LEN}(L[i]) < \text{LEN}(L'[i])$ then $L'[\text{LEN}(L[i])] = (V'', A[i])$ where $V \leq V''$.
5. For all i, if $A[i] \neq A'[i]$ then $\lambda < V'$ and exists $0 \leq j < \text{LEN}(L'[i])$ 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 s.t. $l \in L(\pi)$ 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. (Outline) 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, $\sigma, \text{GET}(A_S, i) \rightarrow^* \sigma', v$ and $(\pi, \mu), \text{GET}(A_T, i) \rightarrow_T^* (\pi', \mu'), v$ 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, $\sigma, \text{NEW}(n, v) \rightarrow^* \sigma', A_S$ *and* $(\pi, \mu), \text{NEW}(n, v) \rightarrow_T^* (\pi', \mu'), A_T$. *Then* $\sigma', A_S \sim (\pi', \mu'), A_T$.

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* $\sigma, \text{GET}(A_S, i) \rightarrow^* \sigma, v$ *where* v int val. *Let* $e_0 = \text{GET}(A_T, i)$ *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. (Outline) We case on the line that the implementation of GET returns. There are 4 cases: lines 6, 7, 10, and 11. In each case, we show that e_n is the same as the value $(\pi, \mu), \text{GET}(A_T, i)$ evaluates to. Since $\sigma, A_S \sim (\pi, \mu), A_T$, this is the same value that $\sigma, \text{GET}(A_S, i)$ evaluates to, which implies $e_n = v$.

The first case is on line 6 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 3. From the implementation of GET, if the entire GET was evaluated atomically on line 3, it would evaluate to *guess*. By lemma 6.5 this means that *guess* and v are the same.

The other cases on lines 7 and 10 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* $\sigma, \text{SET}(A_S, i) \rightarrow^* \sigma', A'_S$ *where* A'_S val. *Let* $e_0 = \text{SET}(A_T, i, v)$ *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. (Outline) 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, $\text{SET}(A_S, i, v)$ 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 10 to 12 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 $\&l'$ s.t. $\pi[l'] = (V + 1, k)$. Going through the definition of $\sim$ this gives us $\sigma', A'_S \sim (\pi'_{n-1}, \mu'_{n-1}), \&l'$.

□

Theorem 6.9. *(Bounded cost) There exists* k *s.t. if* A_T *has size* n *then there does not exist* (π, μ) valid *and a concurrent transition sequence of length* $> kn$ *starting at either* $(\pi, \mu), \text{GET}(A_T, i)$ *or* $(\pi, \mu), \text{SET}(A_T, i, v)$ *for all* i, v int val.

Proof. (Outline) 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 6 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 $\log n$ [3].

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

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

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 3 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, where 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 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 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. 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.

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 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 evaluational dynamics to compute the work W and span S of evaluating an expression e in our 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 we ensured that the entire arrays cannot 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 took 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

1. We do not have formal proofs for the cost bounds of our sequence implementation. Current techniques for analyzing concurrent data structures focus on proving correctness. Coming up with a framework to prove the costs of a concurrent implementation is an exciting opportunity. Note that our main results still hold without these proofs, because we present a general way to analyze data types where operating on leaf values and interior values have different costs.
2. Extending the approach of allowing operations on interior data values to be more expensive than for leaf data values to other data types (for example unordered sets or functional disjoint sets) is a possible future direction.
3. We conjecture that it is not possible for a sequence implementation to have constant (worst case) costs for all operations, which would ground the need for an implementation where costs for leaf and interior versions are different.

4. Our cost dynamics could be extended to situations that generalize beyond separate costs for interior and leaf versions. In particular, our interleaved structural dynamics does not give a tight bound for the cost of our sequence implementation. In our implementation, accessing a value in an interior version could involve constant work if there are no (or a constant number of) log entries at that index.

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. doi: 10.4230/LIPIcs.CSL.2012.107. URL `http://dx.doi.org/10.4230/LIPIcs.CSL.2012.107`.

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.

R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999. ISSN 0004-5411. doi: 10.1145/324133.324234. URL `http://doi.acm.org/10.1145/324133.324234`.

R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, Apr. 1974. ISSN 0004-5411. doi: 10.1145/321812.321815. URL `http://doi.acm.org/10.1145/321812.321815`.

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. ISBN 0-387-55253-7. URL `http://dl.acm.org/citation.cfm?id=145055.145076`.

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

R. Harper. *Practical Foundations for Programming Languages, 2nd edition*. Cambridge University Press, 2016. ISBN 9781107150300.

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. ISSN 0164-0925. doi: 10.1145/78969.78972. URL `http://doi.acm.org/10.1145/78969.78972`.

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. ISBN 0-89791-200-4. doi: 10.1145/319838.319876. URL `http://doi.acm.org/10.1145/319838.319876`.

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, Berlin, Heidelberg, 1991. ISBN 978-3-540-47566-8. doi: 10.1007/BFb0028283. URL `http://dx.doi.org/10.1007/BFb0028283`.

C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, Oct. 1983. ISSN 0164-0925. doi: 10.1145/69575.69577. URL `http://doi.acm.org/10.1145/69575.69577`.

D. J. King and J. Launchbury. Structuring depth-first search algorithms in haskell. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 344–354, 1995.

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

M. E. O'Neill. *Version Stamps for Functional Arrays and Determinacy Checking: Two Applications of Ordered Lists for Advanced Programming Languages*. PhD thesis, Simon Fraser University, 2000.

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. doi: 10.1145/2429069.2429111. URL `http://doi.acm.org/10.1145/2429069.2429111`.

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. ISBN 3-540-59451-5. URL `http://dl.acm.org/citation.cfm?id=647698.734146`.