Extensible Sparse Functional Arrays with Circuit Parallelism

John T. O'Donnell

School of Computing Science, University of Glasgow, Glasgow G12 8QQ, United Kingdom

Abstract

A longstanding open question in algorithms and data structures is the time and space complexity of pure functional arrays. Imperative arrays provide update and lookup operations that require constant time in the Random Access Machine (RAM) theoretical model, but it is conjectured that there does not exist a RAM algorithm that achieves the same complexity for functional arrays, unless restrictions are placed on the operations. The main result of this paper is an algorithm that does achieve optimal unit time and space complexity for update and lookup on functional arrays. This algorithm does not run on a RAM, but instead it exploits the massive parallelism inherent in digital circuits. The algorithm also provides unit time operations that support storage management, as well as sparse and extensible arrays. The main idea behind the algorithm is to replace a RAM memory by a tree circuit that is more powerful than the RAM yet has the same asymptotic complexity in time (gate delays) and size (number of components). The algorithm uses an array representation that allows elements to be shared between many arrays with only a small constant factor penalty in space and time. This system exemplifies circuit parallelism, which exploits large numbers of transistors per chip in order to speed up key algorithms. Extensible Sparse Functional Arrays (ESFA) can be used with both functional and imperative programming languages. The system comprises a set of algorithms and a circuit specification, and it has been implemented on a GPGPU.

Keywords: functional array, sparse array, extensible array, functional programming, circuit parallelism

1. Introduction

A longstanding problem in algorithms and data structures is the complexity of operations on pure functional arrays. This question has both theoretical and practical significance, because arrays are fundamental to much software and the complexity of their operations affects the complexity of many algorithms.

An imperative array supports two operations: fetching an array element $a[i]$ and modifying an array element $a[i] := v$. Both operations require $O(1)$ time, according to common cost models, and they do not require any space beyond the memory originally allocated for the array. After an element of an imperative array is modified the previous content of that element is destroyed.

A pure functional program defines new values but does not perform side effects, such as modifying an existing value. Thus a pure functional array allows new arrays to be constructed but does not allow old ones to be changed. When a functional array is updated, the result is a new array that differs from the old array at one index, but the old array is still accessible. This paper generalizes functional arrays to handle sparse and extensible operations as well, and the data structure is called extensible sparse functional arrays (ESFA). Such an array maps indices to values, and is related, though not identical, to finite maps, hash tables and hash maps.

Imperative arrays are trivial to implement, as they rely on basic machine instructions and addressing modes. In contrast, straightforward implementations of functional arrays are inefficient in space or time, and the most efficient implementations are complex and still asymptotically slower than imperative arrays. Recopying an array being updated gives $O(1)$ access time but causes each update to take $O(n)$ space and time, where n is the array size. Algorithms that maintain balanced binary trees require $O(\log n)$ time for the operations.

Since unrestricted access to functional arrays is inefficient, there has been relatively little exploration of algorithms that rely on them. Nevertheless, functional arrays remain interesting in their own right, and they do have practical applications.

Functional arrays are not simply arrays used in a functional language. Imperative and functional arrays are distinct data structures that support different operations. Both data structures can be used in both imperative and functional languages (imperative array operations can be expressed in a pure functional language using monads or unique types). The choice be-

tween imperative and functional arrays should be based on the needs of the algorithm using them, not on the programming language used to express the algorithm.

This paper discusses the relationship between imperative and functional arrays, and conjectures that that it is impossible to implement functional arrays with $O(1)$ access time using a Random Access Machine (a theoretical model of computation). However, the main result of the paper is a system that does indeed implement functional arrays with the same time and space complexity as imperative arrays, as long as the complexities are compared fairly using the same cost models. This result appears to contradict the conjecture—but the new algorithm runs on a different model of computation which we call *circuit parallelism*.

The essential idea is that the Random Access Machine model—as well as conventional computers—makes inefficient use of the digital logic components that make up the memory. The time (or space) complexity of a system depends on the time (or space) complexity of the underlying machine model as well as of the algorithm. A memory contains an address decoder that takes $O(\log N)$ time for a memory of size N, but the decoder performs no other useful computation. It is customary to calculate the time complexity of an algorithm by counting the number of Random Access Machine steps that it performs, while assuming that each RAM step takes unit time. The system presented here transfers some of the computation from the algorithm to the hardware, where it takes place alongside the address decoder *without increasing the time complexity (gate delay) or the size complexity (gate count) of the machine*. By redesigning the hardware as well as the software (called "hardware/software co-design") we can sometimes beat lower bounds on complexity of RAM algorithms.

Parts of the ESFA system were presented in 1993 [1]. This is an expanded version of a paper that appeared in PPDP 2013 [2], which extended the 1993 work by discussing the operations for sparse and extensible access, analyzing the algorithm and hardware complexity, and giving correctness proofs of key parts of the system. This paper defines both a pure functional interface and a stateful interface, introduces invariants on the representation, defines the implementation using parallel combinators, proves its correctness using equational reasoning, and gives extended examples.

Every one of the ESFA operations takes a small fixed number of clock cycles. No iteration is used in any of the operations; the execution time is constant, and does not depend on the past history of updates or deletions

that led to the current state of the machine. No restrictions are placed on the operations that can be performed in order to achieve these tight time bounds. This constant time performance does not come at the cost of increased hardware complexity: the clock speed of the hardware (as a function of the gate delay) has the same time complexity as the clock for an ordinary addressable memory, and the hardware complexity in terms of number of logic gates and flip flops is also the same.

The algorithms presented here use *circuit parallelism*. This approach originated in associative processors [3] and active data structures [4]; other examples include priority queues [5], systems with chunks of memory organized as trees [6], smart memories for multicore processors [7], and associative searching [8]. Circuit parallelism is also the target platform for compilation of a declarative committed-choice rule language [9]. The idea is to bring the parallelism inherent in digital circuits to bear directly on the computations required by an algorithm, rather than organizing the circuit into conventional processors. In circuit parallelism, the computation is melded into the memory at the level of individual words, allowing algorithms that perform a computation in parallel on every word in the memory; it differs from data parallelism, where an operation performs a computation on every word of a data structure (rather than every word in the machine). Current hardware trends will make this approach increasingly productive, as the number of transistors per chip continues to increase.

The algorithms presented here are fine-grain and massively parallel, and they require suitable hardware in order to be usable. They do not run efficiently on a sequential computer that lacks a hardware accelerator. The fastest platform for ESFA is a direct VLSI implementation of the underlying parallel circuit, but an FPGA or GPU chip could also be used (see Section 8).

The ESFA system has been implemented and tested in several ways. First, this paper contains an executable specification written in Haskell. Second, it is implemented using a digital circuit that is specified and simulated using the Hydra hardware description language [10]. The design has not been fabricated as a physical chip, but the Hydra specification is precise down to the level of flip flops and logic gates (it is "synthesizable"), and the simulation is accurate in clock cycles and also in gate delays within a cycle. Third, the system is implemented as a program, written in C and CUDA [11], that runs on a general purpose GPU [12]. The GPU implementation gives reasonable performance, and there is extremely low variation in execution time of the operations, making it especially valuable for real-time applications and a good

platform for research. An FPGA implementation would be faster, but the GPU program is more portable and can run on many consumer computers.

The software is available on the web [13], including the sequential simulator, the parallel GPU program, a random test data generator, and sample test data files. The programs have been tested using a combination of small hand-written test cases, an SECD machine interpreter [14] that uses ESFA for the environment, and large-scale randomly generated test data. The simulator has run sequences of 725,000 operations, and the GPU has run tens of millions of operations, without error.

Section 2 introduces the operations on extensible sparse functional arrays. Section 3 gives an overview of the algorithm, which contains three layers. Section 4 describes the parallel circuit generator that constitutes the lower level, and Section 5 uses the circuit to implement a family of map, fold, and scan combinators. Section 6 implements the array operations using the combinators and proves their correctness. Related work on arrays in functional languages is described in Section 7, and the time and space complexity of ESFA are analyzed. Section 8 discusses the implementation of ESFA on standard parallel platforms, and Section 9 concludes.

2. Operations on ESF arrays

This section introduces the notation used for ESF arrays and presents two distinct application programming interfaces (APIs). First a high level interface is defined, where each operation is a pure function. Then a lower level interface is defined that makes the machine state visible and supports side effects such as deleting an array. The lower level could be used in an imperative language, in a functional language using monads or similar, or it could be used to implement the higher level API while hiding the side effects. Haskell notation is used throughout.

An *array element* maps an index $i :: Idx$ to a value $v :: Val$. An element is written $i \mapsto v$ and has type $Idx \mapsto Val$. The Idx type is a bounded integer such as Int; a digital circuit must be able to compare two indices. The value type Val is arbitrary. In Haskell notation the $(\mapsto)$ operator is a data constructor that builds the element.

An array $a :: Array$ is a set of elements with disjoint indices; that is, if $(i \mapsto x) \in a$ and $(j \mapsto y) \in a$, then $i \neq j$.

A value of type $Array$ is a *handle* providing access to the array. The representation is opaque: the only way to access to the contents of an array

empty	$:: Array$
update	$:: Array \to (Idx \mapsto Val) \to Array$
lookup	$:: Array \to Idx \to Maybe\ Val$
minDef, *maxDef*	$:: Array \to Maybe\ (Idx \mapsto Val)$
nextDef, *prevDef*	$:: Array \to Idx \to Maybe\ (Idx \mapsto Val)$

Table 1: Array operations: pure functional API

is by using the API. Two arrays can be compared for equality (so *Array* is in the *Eq* typeclass), but no other useful operation can be performed on the value of a handle. In particular, a handle does not point to a data structure, such as a list or tree, that can be traversed by the user program, and the location of an element is not found by doing arithmetic on a handle. This differs from C and related languages, where an array handle is the address of the first element of the array, so useful calculations can be performed on a handle: the address of $a[i]$ is $a + k \times i$, where k is the size of an element.

Arrays are immutable: once created, they can never be modified. There is a pre-defined constant $empty :: Array$ that contains no elements. This is unique and indestructible: the programmer can neither create nor delete it. The user program can create new arrays by updating an existing one, and arrays can be accessed using the other API functions.

If $a :: Array$ is not empty, then its lower (upper) bound is the smallest (largest) index for which it has a defined element. The bounds of *empty* are undefined. If an array a contains an element with index i for every i such that $lowerBound\ a \leqslant i \leqslant upperBound\ a$, the array is *dense*. An array that is not dense is *sparse*: it has one or more indices where there is no value.

2.1. Pure functional interface

The higher level API for ESF arrays consists of a set of pure functions that create and access arrays (Table 1). There is no need for the user program to sequence the array operations using a monad or similar technique; the array functions satisfy referential transparency and can be used just like any other function. The functions are strict in the *Array* and *Idx* operands.

An *update* takes an existing array, index, and value, and returns a new array with the given value at that index. For example,

$$b = update\ a\ (i \mapsto x)$$

creates a new array b that is exactly like a, except that b has value x at index i. The old array a is unchanged. Thus *update* a $(i \mapsto x)$ corresponds roughly to the imperative notation `a[i] := x`, but there is a crucial difference: *update* creates a new array without modifying the old one, which is still accessible.

The result of *update* is undefined if the memory is full; semantically it is $\bot$. Normally this would terminate the execution of the program. The implementation can detect this situation and provide an informative error message, just as with other data constructors like (:) in Haskell or *cons* in Scheme.

Array elements may be accessed using *lookup*, which takes an array and an index and returns the array element defined at that index, if one exists. With the definition of b above, *lookup* b i evaluates to *Just* x. If the array does not contain an element at the specified index, then *lookup* returns *Nothing*. Thus *lookup* a i corresponds to the imperative notation `a[i]`.

Two laws state the relationship between *lookup* and *update*. The first law says that an empty array contains no elements.

Law 2.1 (Empty array)**.** *For all* $i :: Idx$*,*

$$lookup\ empty\ i = Nothing$$

The second law says that an *update* to an existing array a with an element $j \mapsto v$ gives a new array that is identical to a except at index j, where it has the new value v.

Law 2.2 (Nonempty array)**.** *For all* $a :: Array$*, element* $(j \mapsto v) :: (Idx \mapsto Val)$ *and index* $i :: Idx$*,*

$$\begin{aligned} &lookup\ (update\ a\ (j \mapsto v))\ i \\ &\quad \mid i \equiv j = Just\ v \\ &\quad \mid i \not\equiv j = lookup\ a\ i \end{aligned}$$

The programmer does not declare the size of an array or allocate space for it, and an array of undefined elements cannot be allocated all at once, as in Fortran. Instead, elements are added one by one using *update*, and the

system allocates space automatically as needed. The programmer cannot modify an existing array; *update* creates a completely new array and leaves the old one unchanged. Every array is built incrementally through a sequence of updates, starting ultimately from *empty*. For example, *a3* is constructed using three updates:

$$a1 = update\ empty\ (1 \mapsto 101)$$
$$a2 = update\ a1\ (2 \mapsto 102)$$
$$a3 = update\ a2\ (3 \mapsto 103)$$

The values of the resulting arrays are:

$$a1 = \{1 \mapsto 101\}$$
$$a2 = \{1 \mapsto 101, 2 \mapsto 102\}$$
$$a3 = \{1 \mapsto 101, 2 \mapsto 102, 3 \mapsto 103\}$$

In the previous example, each *update* extends the most recently created array, resulting in a linear sequence of *chained updates*. However, *update* is not restricted to this pattern. Any array can be updated at any time, allowing for a tree-structured set of relationships among arrays. Consider the following definitions, with the previous definitions still in scope:

$$a4 = update\ a2\ (4 \mapsto 104)$$
$$a5 = update\ a1\ (5 \mapsto 105)$$

The values of *a1* and *a2* remain unchanged, and the new arrays are

$$a4 = \{1 \mapsto 101, 2 \mapsto 102, 4 \mapsto 104\}$$
$$a5 = \{1 \mapsto 101, 5 \mapsto 105\}$$

If an update gives a new value to an index that has already been defined, the old value is *shadowed*: it does not appear in the new array but is still present in the old one.

$$a6 = update\ a4\ (2 \mapsto 999)$$

The updated array is unchanged, as always, and the new array has a different value at index 2.

$$a4 = \{1 \mapsto 101, 2 \mapsto 102, 4 \mapsto 104\}$$
$$a6 = \{1 \mapsto 101, 2 \mapsto 999, 4 \mapsto 104\}$$

These examples illustrate the chief characteristic of functional arrays: *update* produces a new array, but does not change the old one. They also show why implementation of the operations in $O(1)$ time and space is difficult: it is essential to share each element among any number of arrays without using linked data structures to find them.

An *extensible array* is one whose minimum and maximum bound can be changed at any time. The *update* operation gives extensibility for free, as there is no restriction on the value of the index that is provided. For example, we could define $a7 = update\ a2\ (1000000 \mapsto 7)$. Naturally this can leave many indices where no element is defined: the result of $lookup\ a7\ 100$ is *Nothing*.

In a sparse array, many elements may have a default value d (often 0). The representation should use memory only for the non-default elements, saving space. A sparse lookup can be defined that returns the default at an index that has not been defined with *update*.

In addition to saving space, sparse arrays save time by making it possible to traverse the non-default elements without having to iterate over all the indices. ESF arrays support sparse traversal using the *minDef*, *maxDef*, *nextDef* and *prevDef* operations. Suppose we wish to iterate over all the non-default elements of an array, from the lowest to highest index. The starting point of the iteration is determined using *minDef* to find the lowest index with a non-default value, and the iteration repeatedly applies *nextDef* to the current index to find the next one.

2.2. Stateful interface

The pure functional interface hides the state of the ESFA memory. However, the memory state needs to be made explicit in order to implement storage management. Therefore a lower level API is defined that treats each operation as an effect, and this stateful API can be used to implement the pure functional API. Alternatively, it can be used in an imperative language, or in a functional language with monads, giving the user program control over storage management.

The parallel combinators defined in Section 5 maintain the hidden state using a monad *SystemState*. For the array operations, this type is specialized using a memory *Cell* state type (defined in Section 6.1) and an auxiliary state (needed for other features not discussed in this paper). An array operation that performs a computation which produces a value of type a has type

updateS	$:: Array \to (Idx \mapsto Val) \to Bool$
	$\to EsfaState\ (Result\ Array)$
lookupS	$:: Array \to Idx \to EsfaState\ (Result\ Val)$
minDefS, *maxDefS*	$:: Array \to EsfaState\ (Result\ (Idx \mapsto Val))$
nextDefS, *prevDefS*	$:: Array \to Idx \to EsfaState\ (Result\ (Idx \mapsto Val))$
deleteS	$:: Array \to EsfaState\ ()$
killZombieS	$:: EsfaState\ (Maybe\ Val)$

Table 2: Array operations in the stateful API. The *deleteS* and *killZombieS* operations support memory management, and the others correspond to functions in the pure API in Table 1.

EsfaState a. Operations that might fail return a value of type *Result a*, which is either an error message code or a successful result *a*.

type *SystemState s e a* = *StateT* (*CircuitState s*, *e*) *IO a*
type *EsfaState a* = *SystemState Cell AuxState a*
type *Result a* = *Either ErrMsg a*

Table 2 shows the types of the operations in the stateful interface. These operate in the *EsfaState* monad. The operation *updateS a* $(i \mapsto v)$ *nfy* is similar to the pure *update a* $(i \mapsto v)$, but there are several differences. The result has type *Result Array* because a stateful update may fail: if the argument *a* is an array that has been deleted, or if the memory is full, then the update cannot be performed and an error code is returned. The operation also takes an extra argument $nfy :: Bool$ which indicates whether the value *v* is a pointer to a heap object; the purpose of this is explained below.

The operation *lookupS a i* is essentially the same as the pure *lookup*, except that there are two possible reasons the result may be undefined: (1) if the array *a* does not exist because it has been deleted, and (2) if the array exists but does not have a value defined at index *i*. In these cases an error code is returned. The distinction is important in implementing sparse lookup, where a nonexistent array is really an error but an unbound index should return the default value.

The operations *minDefS*, *maxDefS*, *nextDefS* and *prevDefS* are similar to their pure counterparts. but it is now possible that the array argument *a* does not exist because it has been deleted, and in this case an error is returned.

The *deleteS* and *killZombieS* operations have no pure counterpart; they are used for memory management and perform a side effect on the system state. Executing *deleteS a* removes the array a from the memory and reclaims array elements that have become inaccessible (although shadowed elements may not be reclaimed). After *deleteS a* is performed, *lookupS a i* will return an error code indicating that the array is undefined, for any i. The amount of space that is recovered depends on the sharing of array elements.

The *killZombieS* operation supports integration with a heap memory, which is separate from the ESF machine memory. An array element $i \mapsto v$ becomes inaccessible when every array that contains it has been deleted. If the value v is unboxed, such as a floating point number, the space for the element can be reclaimed immediately. Suppose, however, that v is actually a pointer into the heap memory. In this case, it may be necessary to run a finalizer on v. The ESF system supports this with notification requests and *killZombieS*. When an *updateS* creates an array with an element that requires finalization when it is removed, the *nfy* ("notification request") argument should be *True*. Later, if a *deleteS* causes the element to become inaccessible, it is not reclaimed but instead is marked as a "zombie": inaccessible yet still present. The storage manager can later use *killZombieS* to locate a zombie; if one is found the operation reclaims its space and returns its value v, allowing the storage manager to run the finalizer.

3. Overview of an implementation with circuit parallelism

A good introduction to the problem of implementing functional arrays is to consider two naive algorithms for the *update* and *lookup* functions on dense arrays.

- We implement *update a* $(i \mapsto v)$ *nfy* by copying the array a into a fresh region of contiguous words of memory at address b, and storing the value of v at location $b + i$. All future lookups will take $O(1)$ time, as normally expected for imperative arrays. Unfortunately, the *update* requires both time and space of $O(n)$ where n is the size of a. The equivalent of `for i := 0 to n-1; x[i] := x[i]+1;` would require $O(n^2)$ time and allocate $O(n^2)$ words of memory.

- Since the previous approach is disastrous, we could focus on making *update* efficient. Define an algebraic data type **data** *Arr a* = *Empty* |

Update Arr Idx a. Now the *update* operation is simply an application of the constructor *Update*, and it builds a list of nodes. A *lookup* requires traversing the list, from an *Update* node back toward the root, until the index is found. This approach makes *update* take $O(1)$ time and space, and *lookup* requires time whose worst case is $O(n)$, where n is the size of the array.

Clearly the first approach—wholesale copying—is hopeless. The second approach invites attempts to restructure the tree to reduce the height and thereby make *lookup* faster, but that will never produce $O(1)$ time. Many sophisticated techniques have been developed for making functional data structures more efficient [15]. Generally, a tree structure stored in the heap will give access time proportional to the height of the tree.

The result in this paper is an implementation of functional arrays with the same time and circuit complexity as ordinary imperative arrays on a conventional RAM memory. Each operation in Table 2 always takes O(1) time, measured in clock cycles. There is no variation in execution time depending on the operations or arguments, and there are no restrictions on the usage of the operations.

The naive algorithms above suggest that sharing and searching are the fundamental difficulties, and we now consider these issues more deeply.

Functional arrays allow an arbitrary amount of sharing. If both *a1* and *a2* are defined as updates to a, then each element of a is shared among all three arrays. To achieve $O(1)$ space for an update, an element must be represented in the machine only once, no matter how many arrays it belongs to. To achieve constant time access, however, we cannot follow chains of pointers that link the elements together.

An imperative array is represented as an address that can be used to calculate the address of any element; we can think of an array as "knowing" where its elements are. That approach is inefficient for ESF arrays, because of the sharing.

Therefore consider a major change of perspective. Place each element at an arbitrary location in the memory, and store with each element the set of all arrays to which it belongs. This is called the *inclusion set* (or *iset*) of the element. An array is no longer represented by an address, but just by an arbitrary natural number called a *handle*. An array does not know what its elements are; instead, an element knows which arrays it belongs to.

For this approach to be workable, we need to be able to represent every inclusion set in a fixed amount of space, and also to determine whether any given array belongs to an inclusion set. In general, arbitrary sets cannot be represented in a small fixed amount of space. However, the inclusion sets that appear in the ESF memory are not arbitrary: they satisfy some structural properties that are enforced by the fact that the memory state must be the result of a sequence of *updateS* operations. These invariants allow us to represent the inclusion set for an element e by a pair $(low, high)$ such that an array identified by c is an element of *iset* e if and only if $low \leqslant c \leqslant high$.

The identifying number c for an array is called its *code*. Array codes are not arbitrary, as they must lie between the *low* and *high* fields of every element belonging to the array. Furthermore, as updates take place, the code for an array may need to be changed.

Since the array codes will be changing frequently, how can we represent a pointer to an array? Each array has a stable *handle*, which will never change, and the system contains a mapping from array handles to codes. This consists of a maplet $a \mapsto c$ giving for each array a the current value c of its code. When an array code needs to be changed, the mapping is also adjusted so that every array handle still refers to the same set of elements, even if the code has changed. An array handle is a stable pointer [16] providing a permanent reference to the array. It is an opaque reference: two handles can be compared for equality, but the main program cannot perform any other useful operations on array handles, such as address arithmetic.

Suppose we are updating an array with code c. It turns out that the code of the resulting array has to be $c + 1$, and that code is likely already to be in use. That means that the codes that are larger than c must be incremented, and also the $(low, high)$ fields in many of the cells need to be adjusted. The implementation of update ensures that the newly created array has the right elements, and the existing arrays continue to have the same values although their representations may change.

Every update or delete operation can cause wholesale changes to the representation, affecting a number of memory cells (possibly even all of them). The cost would be prohibitive on a sequential computer, but by adding some logic gates to the flip flops comprising a cell, this can be done in parallel in every cell in unit time. Therefore ESF arrays can be implemented using "hardware/software co-design", with three layers:

1. The lowest layer (Section 4) is a "smart memory" called the ESF Ma-

chine (ESFM) consisting of a tree-structured digital circuit (or abstract machine) that implements a basic machine operation called *sweep*. The circuit is synchronous, with a fixed speed clock. A sweep requires one clock cycle. The clock speed is determined primarily by the gate delay in combinational logic within the circuit. This gate delay is $O(k)$ where k is the depth of the tree. If the circuit is built with a balanced tree, then $k = \log N$ where there are N memory cells, allowing for N array elements to be stored. A conventional memory also has a logarithmic time gate delay, because it needs to decode the memory address. (A conventional tree algorithm running on a conventional machine has time $O(\log N \times \log n)$, where N is the memory size and n is the data structure size.)

2. The middle layer (Section 5) uses the circuit's sweep operation in order to implement a family of map, fold, and scan combinators. Each of these takes one clock cycle and performs a fine-grain parallel computation.
3. The upper layer (Section 6) defines the operations in Table 2 as a sequence of maps, folds, and scans. Each of these algorithms is "straight line code" with no iteration of any kind; thus every Array operation takes a small fixed number of clock cycles.

The main program runs in a host processor with a conventional heap memory, and uses the ESFM to accelerate the performance of array operations. This is analogous to using a floating point unit or a graphics processing unit to speed up specialized operations.

4. Low level: tree circuit

The lowest level of the system is a digital circuit that represents each array element in a small unit of memory enhanced with logic components that allow all the cells to perform calculations in parallel. This is feasible because those calculations are simple (basic arithmetic) and regular (all the cells do the same thing).

The hardware is an extremely fine-grain parallel architecture; for best performance, there should be one processing element for each location in the array memory. The processing elements are not full scale processors; they need only the ability to perform comparisons, increments, and decrements on natural numbers, and a few bit level operations. A processing element

would contain on the order of 100 bits of memory and a few hundred logic gates.

The circuit is a smart memory with a tree structure, similar to the organization of a standard random access memory; see Section 7 for a comparison. Each leaf in the tree contains a state of type s, and the nodes provide combinational logic functions but have no state. The state of the entire machine is modeled by the type *CircuitState*.

```
data CircuitState s
  = Leaf s
  | Node (CircuitState s) (CircuitState s)
```

Each leaf cell is a circuit with one input and one output port, which are connected to the parent node. A port is a set of signals (wires) organized as a tuple of fields. A cell is a state machine with type $s \to d \to (s, u)$ where s is the type of the state, u is the type of the output which goes up the tree, and d is the type of the input which comes down from the tree.

Each node is a pure function implemented by combinational logic gates. It receives three inputs—one coming down from the parent and two coming up from the subtrees—and produces three corresponding outputs. The type of the node function is $d \to u \to u \to (u, d, d)$. There is no state or memory in the tree apart from the state in the leaf cells. The critical path of the circuit (the maximum gate delay) is proportional to the height of the tree, and the clock period is proportional to the critical path.

At a clock tick, each flip flop updates its state by storing the value of its input signal. As the logic gates settle down, information flows from the cells up the tree to the root, which also receives an input from the main processor, and information then flows back down the trees to the cells, preparing for the next clock tick. This general operation is called a *sweep*.

```
sweep
  :: (s → d → (s, u))              -- cf = cell function
  → (d → u → u → (u, d, d))        -- nf = node function
  → d                              -- a = root input from host processor
  → CircuitState s                 -- state of tree circuit
  → (CircuitState s, u)            -- (new state, root output)
```

A sweep causes each leaf cell to apply a logic function to its state to produce an output to send to its parent node; later this function also calculates

the new state using the current state and the incoming down message. These two logic functions are combined in the single function cf.

$$\begin{aligned}&sweep\ cf\ nf\ a\ (Leaf\ c) = (Leaf\ c', y)\\ &\quad \textbf{where}\ (c', y) = cf\ c\ a\end{aligned}$$

Each node uses its nf function to calculate the value a' to send up, and the the values p' and q' to send down to the left and right subtrees. Again, all these calculations are combined in one function nf. An alternative way to define a general tree circuit is to separate the cell and node functions into separate up and down functions, and then to define separate *upsweep* and *dnsweep* functions.

$$\begin{aligned}&sweep\ cf\ nf\ a\ (Node\ x\ y) = (Node\ x'\ y', a')\\ &\quad \textbf{where}\ (a', p', q') = nf\ a\ p\ q\\ &\qquad\qquad (x', p) = sweep\ cf\ nf\ p'\ x\\ &\qquad\qquad (y', q) = sweep\ cf\ nf\ q'\ y\end{aligned}$$

The last argument to *sweep* is the circuit state, and the result is the new state and output: $sweep :: \ldots \to CircuitState\ s \to (CircuitState\ s, u)$. The monadic *sweepm* function does not take the state as an argument; instead, it returns a monadic effect: $sweepm :: \ldots \to SystemState\ s\ e\ u$.

The *sweep* function is a circuit generator: it describes directly what the hardware is doing, and a specific digital circuit can be defined by specifying the functions cf and nf as combinational circuits. The combinators used in the middle level (Section 5) are defined using *sweep*.

The tree machine is a *synchronous* circuit. This means that each flip flop updates its state simultaneously at every clock tick. During the time between clock ticks, the logic gates settle down to produce stable outputs that do not change again for the rest of the clock cycle (the period between ticks). The combinational logic gates (stateless devices, such as the logical *and2* gate, that calculate pure functions) can be partitioned into equivalence classes based on the longest gate delay on any of their inputs. After d gate delays, all the logic gates in the classes below d are stable, the gates in classes above d do not have stable inputs so their outputs are immaterial, and all the gates in class d perform a useful calculation in parallel. The clock period must be slow enough for all the logic gates to settle down to a stable value; this is the maximal gate delay through the circuit, with an additional safety factor.

5. Middle level: parallel combinators

In this section, the general tree circuit *sweep* is used to implement three key combinators: *tmap*, *tfold*, and *tscanl*. The circuit has a state of type *CircuitState s*, where *s* is the state type of a leaf cell. The leaves of the tree form a sequence of cell states, and the combinators (*tmap*, etc.) can be related to the standard functions (*map*, etc.) via conversion functions.

The *tmap* combinator takes a function argument $f :: s \to s$, where s is the type of the state of a cell in the circuit, and maps it over the cells. It is defined as a sweep circuit operating over the tree structure.

$$
\begin{aligned}
&\mathit{tmap} :: (s \to s) \to \mathit{SystemState}\ s\ e\ () \\
&\mathit{tmap}\ f = \mathit{sweepm}\ \mathit{cf}\ \mathit{nf}\ () \\
&\quad \textbf{where}\ \mathit{cf}\ s\ \_ = (f\ s, ()) \\
&\qquad\qquad \mathit{nf}\ a\ p\ q = ((), (), ())
\end{aligned}
$$

The arguments to *sweep*, the leaf cell and node functions, are combinational circuits. The nodes have no role other than to fan out the signals. If map were the only operation to be performed, there would be no need for a tree, but in a synchronous circuit the clock speed is determined by the longest gate delay path. This occurs in the tree when it is instantiated to perform *scanl*, so the system would not run faster if *tmap* were implemented without the tree sweep. Furthermore, it is common for the function f being mapped to be specified as a partial application; in the digital circuit this requires information to be broadcast to the cells, and the tree does so efficiently.

The *tfold* combinator calculates a singleton value by a pairwise combination of values from the leaves. Each leaf cell applies f to its state to obtain a message to its parent node, and the nodes combine messages from the subtrees using g. If the g function is associative, then the *tfold* is equivalent to a *foldl1*, although *tfold* is defined unambiguously even if g is not associative.

$$
\begin{aligned}
&\mathit{tfold} :: (s \to a) \to (a \to a \to a) \to \mathit{SystemState}\ s\ e\ a \\
&\mathit{tfold}\ f\ g = \mathit{sweepm}\ \mathit{cf}\ \mathit{nf}\ () \\
&\quad \textbf{where}\ \mathit{cf}\ s\ \_ = (s, f\ s) \\
&\qquad\qquad \mathit{nf}\ a\ p\ q = (g\ p\ q, (), ())
\end{aligned}
$$

A *tscanl* is similar to an ordinary *scanl*: given an associative function h, it produces all the intermediate fold results that would be calculated by a linear-time *foldl* h, although the results are calculated by the tree in logarithmic

time (see [17] for a correctness proof). The function f obtains a value from a leaf cell, g updates the leaf with an incoming value, and h (which is normally associative) combines two values.

```
tscanl
    :: (s → a)          -- f: get singleton from cell
    → (s → a → s)       -- g: update cell using singleton
    → (a → a → a)       -- h: function to be folded
    → a                 -- initial accumulator
    → SystemState s e a
```

The implementation of *tscanl* is bidirectional: it defines a communication pattern that begins with the leaf cells, transmits information up the tree, and then transmits further information back down the tree.

```
tscanl f g h a = sweepm cf nf a
  where cf s a = (g s a, f s)
        nf a p q = (h p q, a, h a p)
```

6. High level: array algorithms

The previous sections show how a digital circuit generator *sweep* implements parallel *tmap*, *tfold* and *tscanl* combinators. This section uses those combinators to implement the stateful versions of the array operations.

6.1. Machine and cell state

Two general kinds of information are stored in the machine's state: the mapping from array handles to codes, and the set of array elements. In principle, these could be stored independently. However, it is convenient to store one array mapping ($handle \Mapsto code$) and one array element ($index \mapsto value$) in the same cell. The reason for this is that *updateS* creates a new array ($handle \Mapsto code$) and a new element ($index \mapsto value$) at the same time.

Each cell contains a set of fields, defined below, which fall into three groups: arrays, elements, and control flags.

- An array is represented by a ($handle \Mapsto code$) map. If *arrDef* is True, then $handle \Mapsto code$ defines an array with the given handle and code; otherwise there does not exist an array with this handle, and the *code*

field is meaningless. When the system is initialized, each cell is given a unique handle, and the handle field is never changed. However, the *code* field changes frequently. If the array is deleted, then *arrDef* will be set to *False*, but if the element is shared by other arrays it will not be deleted.

- An array element is represented by *eltDef*, *rank*, *low*, *high*, *ind*, and *val*. If *eltDef* is True, then $ind \mapsto val$ is an element with the inclusion set specified by *low* and *high*. The *rank* of the element is the distance (in number of updates) of the element from *empty*.

- The control flags are *notify*, *zombie*, *select*, and *mark*. The first two help to integrate the ESFA memory management with a separate heap memory. The last two are temporary flags used within some of the operations to keep track of sets of cells.

The complete set of fields is defined below, and Figure 1 shows a compact notation used in subsequent example diagrams. Each cell may hold one array element, so this representation requires a constant factor (approximately 5) more space than imperative arrays.

```
data Cell = Cell
  { arrDef   :: Bool,   -- an array is defined in the cell
    handle   :: Nat,    -- stable array handle
    code     :: Nat,    -- changeable code of array (if arrDef)
    eltDef   :: Bool,   -- an element is defined in the cell
    rank     :: Nat,    -- distance from empty (if eltDef)
    low,high :: Nat,    -- inclusion set (if eltDef)
    ind      :: Idx,    -- array element index (if eltDef)
    val      :: Val,    -- array element value (if eltDef)
    notify   :: Bool,   -- report when inaccessible (if eltDef)
    zombie   :: Bool,   -- element must be finalized
    select   :: Bool,   -- temporary control
    mark     :: Bool    -- temporary control
  } deriving Show
```

A cell state is a value of type *Cell* with values for each field. A machine state $st :: \{ Cell \}$ is a set of cell states. The notation *handle* x means the value of field *handle* in the cell state x (and similar for all cell fields). The

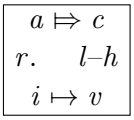


Figure 1: Notation for a cell. The top row of a cell shows the handle a and code c, but if *arrDef* is False these are left blank. The middle row shows the rank r and inclusion interval *low* and *high* (shown as l–h). The bottom row shows the element, which maps index i to value v. If *eltDef* is False the cell does not contain an element, and also cannot contain an array, so the entire cell would be empty. Such cells are omitted from the diagrams.

notation $x\ \{\, code = v \,\}$ denotes a new cell state which is the same as x except that the *code* field has value v.

When $b \leftarrow$ *updateS* $a\ (i \mapsto v)$ *nfy* is performed, an available cell is allocated, and the new array map $(b \Rrightarrow c)$ and new element $(i \mapsto v)$ are stored into that cell. Thus each array element is closely related to an array at the time of creation. Later, the array b may be deleted. If and when that happens, there are two possibilities:

- If the corresponding element $i \mapsto v$ also belongs to another array that has not been deleted, then the element continues to exist. However, the array mapping $b \Rrightarrow c$ is removed by setting *arrDef* to False in the cell. The cell now contains just an element, but no array mapping. The cell itself is still "full" and is not available for an *update*.

- If the element is not shared by any other array, then both the array $b \Rrightarrow c$ and the element $i \mapsto v$ can be deleted. In this case, the cell no longer contains useful information, both *arrDef* and *eltDef* are set to *False*, and the cell is available for allocation by a future *updateS*.

6.2. Array representation: paths, codes, and inclusion sets

As the machine state evolves through a series of updates, the representation of the arrays is scattered among the cells. As a way to visualize the representation, we can think of the cells as nodes in a forest of trees (Figure 2). Whenever an array is defined by updating *empty*, such as $a =$ *update empty* $(i \mapsto v)$ *nfy*, a new tree whose root is a cell containing the new element is added to the forest. When an array b is created by updating a, an available cell is allocated and the new information (i.e. the new array

and the new element) is stored into that cell, which becomes a child of the node for a.

Definition 6.1. *Let $b \leftarrow$ update a $(i \mapsto v)$ nfy. Suppose the array map for a is in cell x, and the array map for b (as well as the element $i \mapsto v$) is in cell y. Then x is the parent of y, and y is a child of x.*

This tree illustrates the relationships among the cells, and it helps to see what is going on in an example. However, the tree is not represented explicitly in the machine: the cell locations are arbitrary, there are no pointers, the algorithm does not perform conventional tree traversal, and this tree does not introduce a logarithmic factor in the time complexity. The tree is purely conceptual; the only real tree in the system is the digital circuit which belongs to a lower level of abstraction.

The *empty* array is a special case. It always exists, and is not the result of an *updateS*. It has a fixed handle, cannot have an element, and it is not stored in any cell. That is why the figure shows a forest of trees rather than one tree rooted in *empty*.

Definition 6.2. *The empty array exists in every machine state, and its handle is -1.*

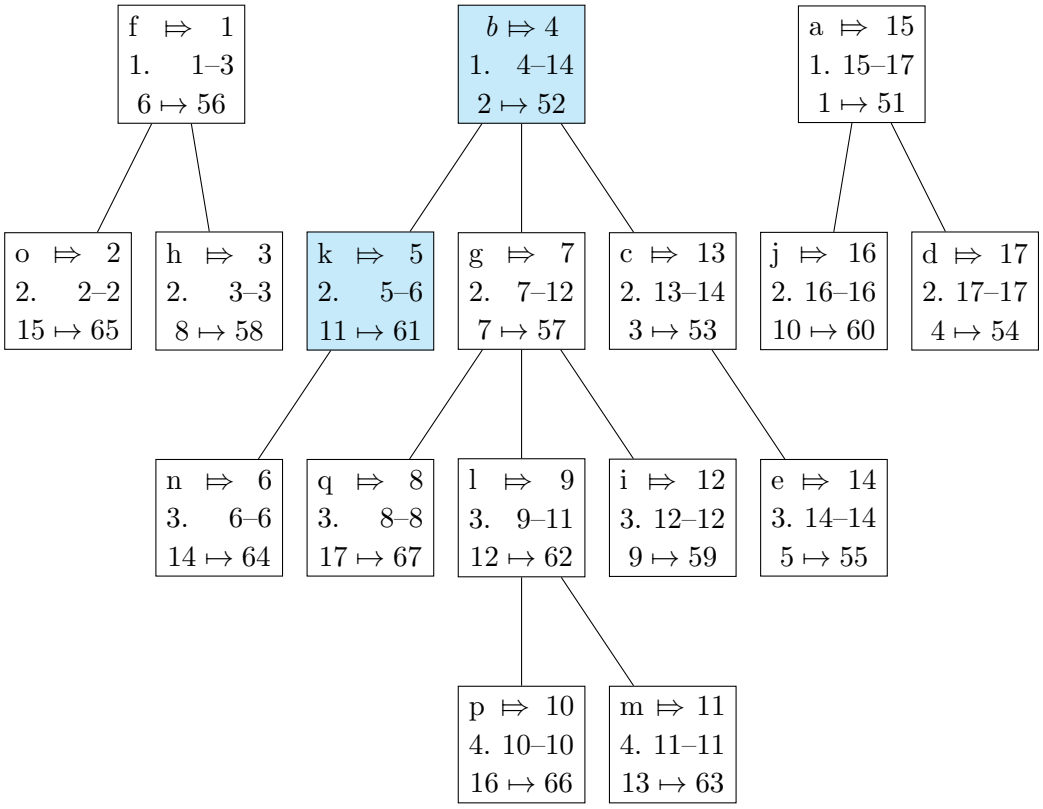
```
empty :: Array   -- always exists, and contains no elements
empty = -1       -- its stable handle is -1
```

Every nonempty array is the result of a sequence of updates, each of which allocates a cell. The sequence of these cells is called the *path* of the array.

Definition 6.3. *The* path *of an array a is the sequence of cells $x_0, \ldots, x_n$ such that handle $x_0 = a$, x_n contains the result of an update to empty, and x_{i+1} is the parent of x_i for $0 \leqslant i < n$.*

The elements of an array are all found on its path. Essentially, *lookupS a i* searches the path of a for the first matching element: i.e. the first element with index i. The path may include several elements with the same index i, and the correct value of the array at that index is first matching element in the path.

Laws 2.1 and 2.2 specify the result of a *lookup* as a recursive traversal of the sequence of updates. This is effectively a search of the path of the



$a = update\ empty\ (1 \mapsto 51)$ $j = update\ a\ (10 \mapsto 60)$
$b = update\ empty\ (2 \mapsto 52)$ $k = update\ b\ (11 \mapsto 61)$
$c = update\ b\ (3 \mapsto 53)$ $l = update\ g\ (12 \mapsto 62)$
$d = update\ a\ (4 \mapsto 54)$ $m = update\ l\ (13 \mapsto 63)$
$e = update\ c\ (5 \mapsto 55)$ $n = update\ k\ (14 \mapsto 64)$
$f = update\ empty\ (6 \mapsto 56)$ $o = update\ f\ (15 \mapsto 65)$
$g = update\ b\ (7 \mapsto 57)$ $p = update\ l\ (16 \mapsto 66)$
$h = update\ f\ (8 \mapsto 58)$ $q = update\ g\ (17 \mapsto 67)$
$i = update\ g\ (9 \mapsto 59)$ $x = lookup\ k\ 2$

Figure 2: State of the machine after a sequence of updates that created arrays $a, \ldots, q$. The cells that are selected during *lookup* k 2 are shaded. The updates could have been performed in any other order, as long as data dependencies are satisfied; the contents of the arrays would be the same as in this figure, and the representation would differ but still satisfy the invariants.

array, but an ordinary search would require iteration and therefore would not achieve $O(1)$ time for lookup.

The central idea behind the parallel implementation of ESF arrays is to introduce a fast way to determine whether an arbitrary element belongs to an array, using a fixed amount of information. The *inclusion set* (*iset*) of an arbitrary element is the set of arrays that contain the element, and is represented using two inequalities on the *low* and *high* fields in the cell containing the element.

The inequalities for an inclusion set require an array a to be identified by a code c with special properties, not by a stable handle which is arbitrary. Furthermore, the code of an array may need to be changed during some updates. That is the reason for maintaining the $a \Mapsto c$ map.

Most operations on ESF arrays take an argument $a :: \mathit{Array}$, which is a stable handle that must be converted to the current value of the array code. The *encode* function, given in Definition 6.4, performs this conversion provided that the array a is defined. If array a exists in state st, then $\mathit{encode}\ a\ st$ is the value of its code. This form of *encode* is convenient for correctness proofs, as it makes the machine state explicit; an implementation *encodeS* is given later. The *empty* array is a special case: its *handle* is -1 and its *code* is always 0. The array operations never change the *code* of *empty*. The function $\mathit{sgl}\ s$ returns the element of a singleton set, and indicates an error if s is not a singleton.

Definition 6.4. *The code of an array a in machine state st is defined as*

$$
\begin{aligned}
&\mathit{encode} :: \mathit{Array} \to \{\,\mathit{Array}\,\} \to \mathit{Nat}\\
&\mathit{encode}\ a\ st\\
&\quad \mid a \equiv \mathit{empty} = 0\\
&\quad \mid a \not\equiv \mathit{empty} = \mathit{sgl}\,\{\,\mathit{code}\ x \mid x \in st \wedge \mathit{arrDef}\ x \wedge \mathit{handle}\ x \equiv a\,\}
\end{aligned}
$$

Inclusion sets are determined using array codes and the *low* and *high* fields. The inclusion set (*iset*) of a cell x in state $st :: \{\,\mathit{Cell}\,\}$ is the set of arrays with a handle that maps to a code within the cell's *low* to *high* interval.

Definition 6.5. *The inclusion set (iset) of a cell x in state st is*

$$\mathit{iset}\ x\ st = \{\,p :: \mathit{Nat} \mid \mathit{low}\ x \leqslant \mathit{encode}\ p\ st \leqslant \mathit{high}\ x\,\}$$

The crucial property of the system is that the inclusion set (which allows $O(1)$ time for membership test) is equivalent to the path (which corresponds to the semantics of *updateS* and *lookupS* but does not achieve constant time or space). This equivalence means that two comparisons suffice to determine whether an arbitrary array belongs to the path of an element.

Invariant 6.6. *Suppose an array a has code c. Then the path of a is the set of cells with an inclusion interval $(low, high)$ such that $low \leqslant c \leqslant high$.*

Thus the inclusion set (*iset*) of a cell that contains an array element is the set of arrays whose code lies between *low* and *high* fields of the cell. This is defined as a function of both the cell x and the machine state *st* that contains x; the full state is required in order to provide the full mapping from array handles to codes.

The following invariant describes the relationship of array codes within subtrees. Its means that isets respect the fact that a cell lies on the path of each of its descendants.

Invariant 6.7. *Suppose an array a is defined in cell x which holds an element with inclusion interval $(low, high)$. Then every descendant of x has a code c such that $low \leqslant c \leqslant high$.*

Figure 2 provides many examples of the invariants. For Invariant 6.6, consider array k, whose code is 5. The path of k consists of the elements whose inclusion interval contains 5: the element $2 \mapsto 52$ because $4 \leqslant 5 \leqslant 14$, and the element $11 \mapsto 61$ because $5 \leqslant 5 \leqslant 6$. No other cell in the machine has an inclusion interval containing 5.

Invariant 6.7 is illustrated by array g, whose element $7 \mapsto 57$ has inclusion interval $(7, 12)$. Every descendant of that cell has a code between 7 and 12, and no other cell in the machine has that property.

If a path contains several elements with the same index, the first such element gives the array value at that index, and any further matching elements are shadowed. The correct element is the one with the highest *rank*. A crucial point is that the tree structure is not represented explicitly in the machine state: there are no pointers from one node to another. Therefore the *rank* field is essential, although it simply gives the depth of a node in the tree. A conventional recursive tree traversal algorithm could keep track of the depth, but the circuit-parallel algorithm cannot do this because it does not traverse the tree (indeed it could not do so, as the pointers shown in the

figure do not exist in the representation). The tree structure is implicit in the values of the cells.

6.3. Basic operations

This section defines a few small operations that are used in implementing the array operations.

In principle, the value of the *handle* field is arbitrary. A convenient way to generate the handles is to enumerate the cells from left to right, using a *tscanl* to set the *handle* fields in the sequence of leaf cells to $0, 1, \ldots$. This is performed by a parallel scan that stores $k = \sum_{i=0}^{k} 1$ in the *handle* field of leaf cell k: f causes each cell to output 1, the sequence is summed by $(+)$, and the result is stored by g. After the *handle* field has been initialized, it is never modified again.

$$
\begin{aligned}
&\mathit{initialize} :: \mathit{EsfaState}\ \mathit{Int} \\
&\mathit{initialize} = \mathit{tscanl}\ f\ g\ (+)\ 0 \\
&\quad \textbf{where}\ f\ x = 1 \\
&\qquad\qquad g\ x\ a = x\ \{\mathit{handle} = a\}
\end{aligned}
$$

The *encode* function (Definition 6.4) is convenient for correctness proofs, as it makes the machine state explicit. However, there are several practical reasons for defining a more concrete version, *encodeS*. First, the array operations need to work in the *EsfaState* monad, with the state hidden. Second, calculations should be carried out with the parallel combinators. Third, the result should be returned in a Maybe type to indicate success or failure. Fourth, it is convenient for *encodeS* to return the *rank* field of the cell as well as the *code*. (This is not necessary but it saves a separate fold later and simplifies the algorithm. The *code* and *rank* are not directly related to each other; the *code* is part of the array ($\mathit{handle}\text{`}\mathit{ArrayCode}\text{`}\ \mathit{code}$) mapping, while the *rank* belongs to the element stored in the same cell.)

The implementation of *encodeS* begins by checking for the special case: if $a \equiv \mathit{empty}$ then the fixed code of *empty* is returned, along with a dummy *rank* of 0. Otherwise, a *tfold* performs a parallel associative search for the cell containing the result.

$$
\begin{aligned}
&\mathit{encodeS} :: \mathit{Array} \to \mathit{EsfaState}\ (\mathit{Maybe}\ (\mathit{Int}, \mathit{Int})) \\
&\mathit{encodeS}\ a = \\
&\quad \textbf{if}\ a \equiv \mathit{empty}
\end{aligned}
$$

```
    then return (Just (0, 0))
    else tfold (getCode a) choose
```

The *tfold* causes each cell to perform (*getCode a*) in parallel; if a cell contains an array definition with the handle being searched for, it returns the corresponding code and rank.

```
getCode :: Array → Cell → Maybe (Int, Int)
getCode a x =
  if arrDef x ∧ handle x ≡ a
    then Just (code x, rank x)
    else Nothing
```

The last argument to *tfold* is an associative combining function *choose* which transmits the result up to the root. The *choose* function makes a Nothing-avoiding choice (a common operation in associative algorithms).

```
choose :: Maybe a → Maybe a → Maybe a
choose x Nothing = x
choose Nothing y = y
choose x y = x
```

The *allocate* function searches associatively for an empty cell that can be used to store a new array element, using a parallel fold to choose the handle from the first available cell.

```
allocate :: EsfaState (Maybe Int)
allocate = tfold availableHandle choose
```

The cell function *availableHandle* determines whether a cell is available

```
availableHandle :: Cell → Maybe Array
availableHandle x =
if ¬ (eltDef x ∨ zombie x)
  then Just (handle x)
  else Nothing
```

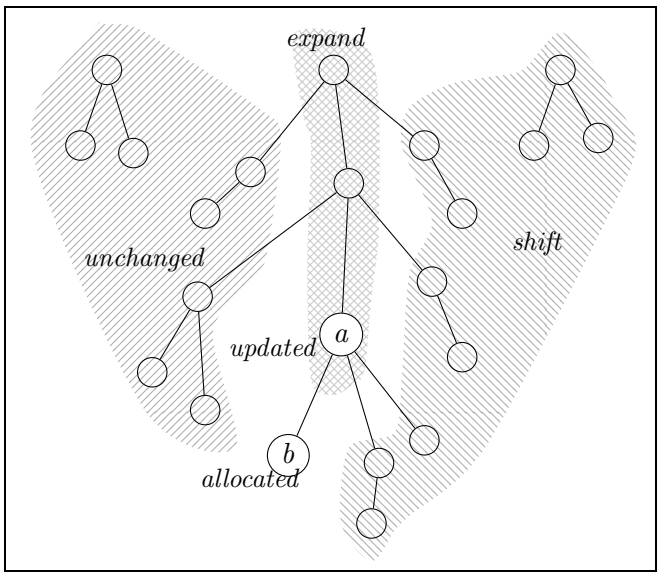


Figure 3: Cases for alterations during $b = update\ a\ (i \mapsto v)\ nfy$. The cell labeled a contains the (*handle* ‘*ArrayCode*‘ *code*) map for a, and the path for a is the shaded group of three cells going up from a; these cells increment *high* while leaving *low* the same (case *expand*). The result b is stored in a newly allocated cell. All cells in the left shaded area (case *no change*) have their inclusion interval unchanged. The cells in the right shaded area “get out of the way” by incrementing both *low* and *high* (case “shift”), and the codes for the corresponding arrays are incremented.

6.4. The updateS operation

An update creates a new array and stores the new array element into the machine state. This is a small fixed amount of information, corresponding to the small amount stored during an imperative array update. A major difference between an ESF update and an imperative array update is that a (possibly large) subset of the cells must modify some of their fields.

The update operation is presented in stages: Section 6.4.1 describes the effect that update has on the state, Section 6.4.2 proves the correctness of the algorithm, and Section 6.4.3 shows how the algorithm is implemented using the parallel combinators.

An update can fail due to two possible conditions: (1) if the array being updated does not exist (i.e. its handle does not appear in the array handle/code map; this occurs with the stateful version of update if the array was deleted but is impossible with the pure update); (2) if the memory is

full, so a new cell cannot be allocated. These conditions are handled by returning an error. To accommodate this, *updateS* returns a result in the *Either* type: *Right* r indicates a success with result r, while *Left* m indicates a failure with message code m.

Consider execution of $b \leftarrow$ *update* a $(i \mapsto v)$ *nfy*. Let st be the current state of the system: the set of cell states before the update. Assume the array a exists and $c =$ *encode* a st; thus the array a has code c in the old state st. (If a does not exist, the *updateS* will return an error, and *encode* a $st = \lambda bot$ is not used.)

6.4.1. Effect of updateS

If the memory is full, or the updated array does not exist, then the update fails and the state is not modified. Otherwise, an update causes each cell x to perform a local computation independently of all the other cells. The local computation consists of (possibly) changing the element, and (possibly) changing the array *handle* ‘*ArrayCode*‘ *code* map. Figure 4 continues with the state in Figure 2, and shows the effect that an *updateS* has on the representation.

Array element.. The action performed by an arbitrary cell x falls into one of five cases, depending on its field values in the old state st (Figure 3).

- **Empty:** if $\neg$ (*eltDef* $x \mid$ *zombie* x) then x contains no relevant information, and if it is not chosen by *allocate* to hold the new array and element, then x remains empty. In this case the cell does not change its state, and it does not affect the arrays that are represented in the machine.

- **Allocated:** if $\neg$ (*eltDef* $x \mid$ *zombie* x) and the empty cell x is chosen by *allocate*, the new array and element are stored into x. Exactly one cell has this property.

- **Shift:** if *eltDef* x and $c <$ *low* x then both *low* and *high* are incremented. The representation of the inclusion set shifts up by a unit, but the size of the set remains the same. Incrementing *low* and *high* does not change the path because the corresponding array codes are also incremented; see below.

- **Expand:** if *eltDef* x and *low* $x \leqslant c$ and $c \leqslant$ *high* x then *low* is unchanged but *high* is incremented. This expands the size of the inclusion

set of x to accommodate the new array element being stored by the update.

- **Unchanged:** if *eltDef* x and $c > \mathit{high}\ x$ then both *low* and *high* are left unchanged.

Each cell performs a computation based on which case it belongs to. The effect of the computation is defined below,

Array code.. There are two cases for an arbitrary cell x.

- If *arrDef* x and $c < \mathit{code}\ x$ then the *code* field is incremented.
- Otherwise the *code* is unchanged.

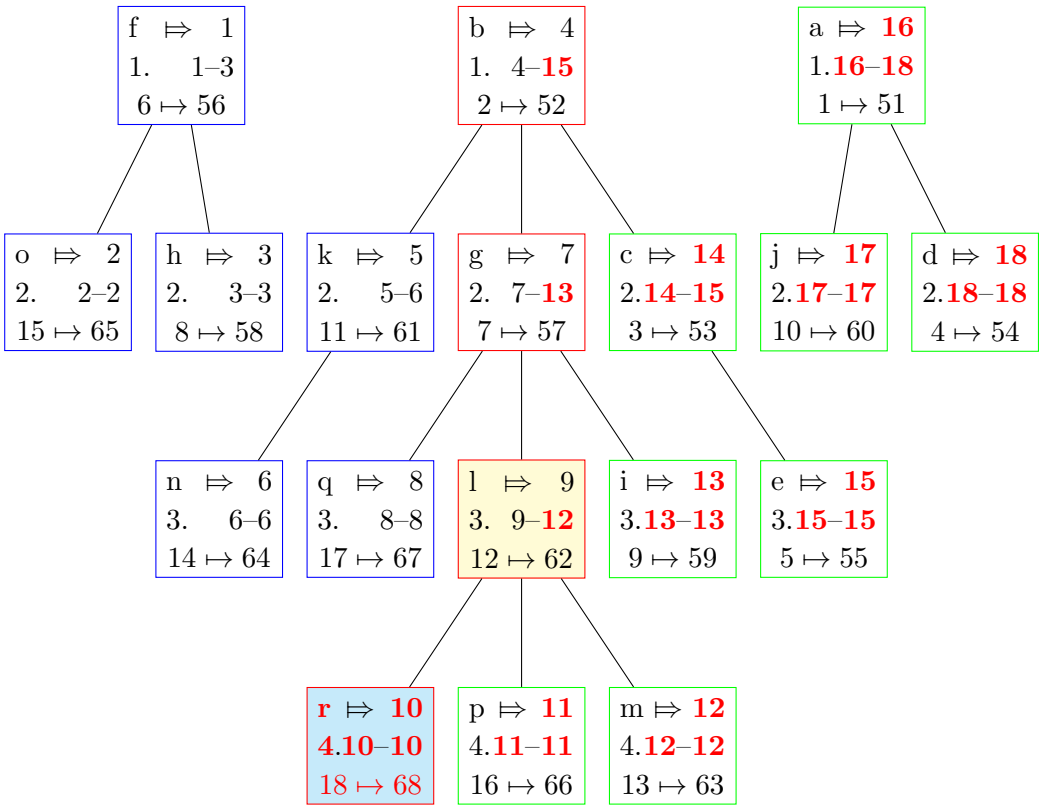
Summary of local computation in a cell.. The computations are performed by *tmap*.

- The allocated cell x has $\mathit{arrDef}\ x = \mathit{False}$. The new array and the new element are stored into this cell: the new value of x is $x\ \{\mathit{arrDef} = \mathit{True}, \mathit{code} = c + 1, \mathit{eltDef} = \mathit{True}, \mathit{low} = c + 1, \mathit{high} = c + 1\}$.
- Every cell x where $\mathit{arrDef}\ x = \mathit{True}$ contains an element of the array-code mapping $(\mathit{handle}\ x \Mapsto \mathit{code}\ x)$. The handle remains constant, but the *code* field is adjusted:

$$\mathit{code}\ x' = \textbf{if}\ c < \mathit{code}\ x\ \textbf{then}\ \mathit{code}\ x + 1\ \textbf{else}\ \mathit{code}\ x$$

- Every cell x where $\mathit{eltDef}\ x = \mathit{True}$ contains an array element. The element itself remains the same, but the representation of the inclusion set is recalculated by adjusting the *low* and *high* fields:

$$\begin{aligned} \mathit{low}\ x' &= \textbf{if}\ c < \mathit{low}\ x\ \ \textbf{then}\ \mathit{low}\ x + 1\ \ \textbf{else}\ \mathit{low}\ x \\ \mathit{high}\ x' &= \textbf{if}\ c \leqslant \mathit{high}\ x\ \textbf{then}\ \mathit{high}\ x + 1\ \textbf{else}\ \mathit{high}\ x \end{aligned}$$



$$r = update\ l\ (18 \mapsto 68)$$

Figure 4: *Example of update.* State of the machine, starting from Figure 2, after defining $r = update\ l\ (18 \mapsto 68)$. The node corresponding to the array being updated contains element $12 \mapsto 62$, and the new element $18 \mapsto 68$ is stored in a freshly allocated cell. Cells where low and high are unchanged (case *no change*) are outlined in blue; cells where only high is incremented (case *expand*) are outlined in red, and cells where low and high are both incremented (case *shift*) are outlined in green.

6.4.2. Correctness of updateS

An update is correct if the new array it returns has the right elements, and all other arrays have the same set of elements they did before. This is not trivial, as the update will change key fields (*code*, *low*, *high*) in many cells.

There are two ways to prove that all the arrays are represented correctly after an update: we could consider each array and show that it has exactly the right elements, or we could consider each element and show that it is contained in exactly the right arrays. This proof takes the second approach, focusing on the elements. The proof consists of a calculation showing that each array element in the new state has the correct inclusion set; that is, each maplet $(i \mapsto v)$ is an element of the correct set of arrays.

Let the state of the machine before an update be st, and define $b \leftarrow$ *updateS* a $(i \mapsto v)$ *nfy* resulting in a new state st' and new array b. If *encode* a $st \equiv c$, then the *updateS* algorithm defines *encode* b $st' \equiv c + 1$. For an arbitrary cell x in state st, the algorithm defines *code* $x' =$ **if** $c <$ *code* x **then** *code* $x + 1$ **else** *code* x. Therefore, if an arbitrary array p is defined in state st,

encode p $st' =$
  if $c <$ *encode* p st **then** *encode* p $st + 1$ **else** *encode* p st

The calculation of the effect of an *updateS* uses the following properties of *encode*.

Lemma 6.8. *Suppose array a with code c is being updated; x is a cell state containing an element in machine state st, and updateS transforms it to x' in st'. Let p be an arbitrary array that exists in st.*

1. *If encode p $st' \leqslant c$ then encode p st' = encode p st.*

   Proof. Assume *encode* p $st' \leqslant c$, and consider the following exhaustive cases:

   - If $c <$ *encode* p st then *encode* p $st' =$ *encode* p $st + 1$, so $c <$ *encode* p $st' - 1$ which violates the assumption. This case cannot hold.
   - If $c \geqslant$ *encode* p st then *encode* p $st' =$ *encode* p st.

   □

2. *If* $c + 1 < encode\ p\ st'$ *then* $encode\ p\ st' = encode\ p\ st + 1$.

   Proof. Consider the following exhaustive cases.

   - If $c < encode\ p\ st$ then $encode\ p\ st' = encode\ p\ st + 1$.
   - If $c \geqslant encode\ p\ st$ then $encode\ p\ st' = encode\ p\ st$, so $c \geqslant encode\ p\ st'$ and $c + 1 \geqslant encode\ p\ st' + 1$ which violates the assumption. This case cannot hold.

   □

Lemma 6.8 considers $encode\ p\ st' \leqslant c$ and $c + 1 < encode\ p\ st'$, but it omits the case where $encode\ p\ st' \equiv c + 1$. The reason is that in that final case p is actually a, the newly created array, and a does not exist in the old state st.

The following lemmas establish that the allocated cell contains the right result, and the other empty cells are unaffected by update.

Lemma 6.9 (Case Empty)**.** *If cell x in state st is empty (i.e. available for allocation) and is not chosen by allocate, then it remains empty: i.e. the new cell state x' in machine state st' is empty.*

Proof. If $\neg\ (eltDef\ x \mid zombie\ x)$ in st, and x is not allocated, then none of the fields of x are changed. □

Lemma 6.10 (Case Update–Allocated)**.** *If cell x is empty in st (i.e. $\neg\ (eltDef\ x \mid zombie\ x)$) and it is allocated, then its state is altered to x' in st such that $iset\ x'\ st' = \{\, b \,\}$, where b is the result of updating a.*

Proof. The new element $i \mapsto v$ is stored in x', and $eltDef\ x'$ is set to True. Also, $arrDef\ x'$ is set to True and $code\ x' = c + 1$, so the array handle-code mapping contains $handle\ x' \Rrightarrow c + 1$. The value of $handle\ x'$ is returned and bound to b. The algorithm defines $low\ x' \equiv c + 1 \equiv encode\ x'\ st' \equiv high\ x'$, therefore $path\ x' = \{\, b \,\}$. □

The previous lemma shows that the allocated cell is correct. It is more complicated to establish that all the existing arrays remain undisturbed. Lemmas 6.11, 6.12 and 6.13 establish that an arbitrary cell containing an array element is handled correctly.

The lemmas use several notational conventions. Assume that array a exists in state st, and is being updated to produce a new array b. Consider an arbitrary cell x in st such that $eltDef\ x$ is True, and let x' be the new value of the cell in st' after the update. Let $c = encode\ a\ st$; thus array a has code c. There are three cases: Shift (if $c < low\ x$), Expand (if $low\ x \leqslant c \leqslant high\ x$), and Unchanged (if $high\ x < c$). These cases are exhaustive.

Lemma 6.11 (Case Update–Shift)**.** *If $c < low\ x$, then $iset\ x'\ st' = iset\ x\ st$.*

Proof. Since $c < low\ x$ and $low\ x \leqslant high\ x$, it follows that $c < high\ x$ and by the definition of $updateS$, the new field values are $low\ x' = low\ x + 1$ and $high\ x' = high\ x + 1$. The effect is that x "shifts out of the way" of the region where the update is taking place. The inclusion set of the modified cell x' is calculated as follows:

$$
\begin{array}{ll}
 & iset\ x'\ st' \\
\equiv & \quad \langle \text{ Definition 6.5 of } iset \rangle \\
 & \{ p :: Nat \mid low\ x' \leqslant encode\ p\ st' \leqslant high\ x' \} \\
\equiv & \quad \langle \text{ Substitute } x \text{ for } x' \rangle \\
 & \{ p :: Nat \mid low\ x + 1 \leqslant encode\ p\ st' \leqslant high\ x + 1 \} \\
\equiv & \quad \langle \text{ Substitute } st \text{ for } st'; \rangle \\
 & \quad \langle\ c < low\ x \text{ so } c + 1 < low\ x + 1 \leqslant encode\ p\ st'; \text{ Lemma 6.8(2)} \rangle \\
 & \{ p :: Nat \mid low\ x + 1 \leqslant encode\ p\ st + 1 \leqslant high\ x + 1 \} \\
\equiv & \quad \langle \text{ Subtract 1 from each side of inequality} \rangle \\
 & \{ p :: Nat \mid low\ x \leqslant encode\ p\ st \leqslant high\ x \} \\
\equiv & \quad \langle \text{ Definition 6.5} \rangle \\
 & iset\ x\ st
\end{array}
$$

□

The most complicated case is Expand, where the new array must be added to the cell's inclusion set.

Lemma 6.12 (Case Update–Expand)**.** *If $low\ x \leqslant c \leqslant high\ x$ then $iset\ x'\ st' = iset\ x\ st \cup \{ b \}$ where b is the result of the update.*

Proof. In this case, $\neg\ (c < low\ x)$ so low is unchanged, but $c \leqslant high\ x$ so $high$ is incremented. This expands the size of the inclusion set, allowing the

new array b to be added to it. Some of the array codes that arise in the inequalities may be larger than c and will be incremented, while others are not. Therefore the calculation splits the inclusion set into several subsets.

$$
\begin{array}{ll}
& \mathit{iset}\ x'\ st' \\
\equiv & \quad \langle\ \text{Definition 6.5}\ \rangle \\
& \{\, p :: \mathit{Nat} \mid \mathit{low}\ x' \leqslant \mathit{encode}\ p\ st' \leqslant \mathit{high}\ x' \,\} \\
\equiv & \quad \langle\ \text{Substitute } x \text{ for } x'\ \rangle \\
& \{\, p :: \mathit{Nat} \mid \mathit{low}\ x \leqslant \mathit{encode}\ p\ st' \leqslant \mathit{high}\ x + 1 \,\} \\
\equiv & \quad \langle\ \text{split into three sets based on location of } c\ \rangle \\
& \{\, p :: \mathit{Nat} \mid \mathit{low}\ x \leqslant \mathit{encode}\ p\ st' \leqslant c \,\} \\
& \cup \{\, p :: \mathit{Nat} \mid \mathit{encode}\ p\ st' \equiv c + 1 \,\} \\
& \cup \{\, p :: \mathit{Nat} \mid c + 1 < \mathit{encode}\ p\ st' \leqslant \mathit{high}\ x + 1 \,\} \\
\equiv & \quad \langle\ (1)\ \text{Lemma 6.8(1)}\ \rangle \\
& \quad \langle\ (2)\ \text{Algorithm defines } \mathit{encode}\ b\ st' = c + 1\ \rangle \\
& \quad \langle\ (3)\ \text{Lemma 6.8(2)}\ \rangle \\
& \{\, p :: \mathit{Nat} \mid \mathit{low}\ x \leqslant \mathit{encode}\ p\ st \leqslant c \,\} \\
& \cup \{\, b \,\} \\
& \cup \{\, p :: \mathit{Nat} \mid c + 1 < \mathit{encode}\ p\ st + 1 \leqslant \mathit{high}\ x + 1 \,\} \\
\equiv & \quad \langle\ \text{Subtract 1 from both sides of inequalities in (3)}\ \rangle \\
& \quad \langle\ \text{Reorder the sets as } \cup \text{ is commutative}\ \rangle \\
& \{\, p :: \mathit{Nat} \mid \mathit{low}\ x \leqslant \mathit{encode}\ p\ st \leqslant c \,\} \\
& \cup \{\, p :: \mathit{Nat} \mid c < \mathit{encode}\ p\ st \leqslant \mathit{high}\ x \,\} \\
& \cup \{\, b \,\} \\
\equiv & \quad \langle\ \text{Combine the set comprehensions}\ \rangle \\
& \{\, p :: \mathit{Nat} \mid \mathit{low}\ x \leqslant \mathit{encode}\ p\ st \leqslant \mathit{high}\ x \,\} \\
& \cup \{\, b \,\} \\
\equiv & \quad \langle\ \text{Definition 6.5}\ \rangle \\
& \mathit{iset}\ x\ st \cup \{\, b \,\}
\end{array}
$$

□

Lemma 6.13 (Case Update–Unchanged)**.** *If* $c > \mathit{high}\ x$ *then* $\mathit{iset}\ x'\ st' = \mathit{iset}\ x\ st$.

Proof. This is the simplest case, as the code c is above the relevant fields of x. Since $c > \mathit{high}\ x$, it follows that $c > \mathit{low}\ x$ and by the definition of *updateS*,

the new field values are $low\ x' = low\ x$ and $high\ x' = high\ x$. The inclusion set of the modified cell x' is calculated:

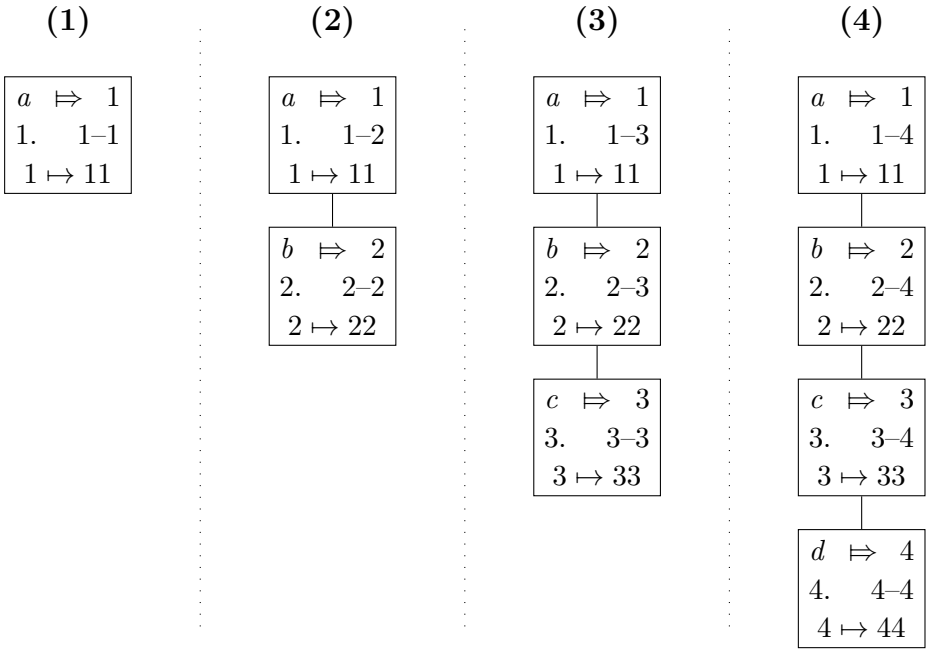
$$
\begin{array}{ll}
 & \mathit{iset}\ x'\ st' \\
\equiv & \quad \langle \text{ Definition 6.5 } \rangle \\
 & \{ p :: \mathit{Nat} \mid \mathit{low}\ x' \leqslant \mathit{encode}\ p\ st' \leqslant \mathit{high}\ x' \} \\
\equiv & \quad \langle \text{ Substitute } x \text{ for } x' \rangle \\
 & \quad \langle\ \mathit{low}\ x' = \mathit{low}\ x \text{ and } \mathit{high}\ x' = \mathit{high}\ x\ \rangle \\
 & \{ p :: \mathit{Nat} \mid \mathit{low}\ x \leqslant \mathit{encode}\ p\ st' \leqslant \mathit{high}\ x \} \\
\equiv & \quad \langle \text{ Substitute } st \text{ for } st' \rangle \\
 & \quad \langle\ \mathit{encode}\ p\ st' \leqslant \mathit{high}\ x < c; \text{ Lemma 6.8(1) } \rangle \\
 & \{ p :: \mathit{Nat} \mid \mathit{low}\ x \leqslant \mathit{encode}\ p\ st \leqslant \mathit{high}\ x \} \\
\equiv & \quad \langle \text{ Definition 6.5 } \rangle \\
 & \mathit{iset}\ x\ st
\end{array}
$$

□

Theorem 6.14 (Correctness of update)**.** *Suppose* $b \leftarrow \mathit{updateS}\ a\ (i \mapsto v)\ \mathit{nfy}$ *is executed in machine state* st*, resulting in a new state* st'*, where each cell state* x *in* st *is altered to* x' *in* st'*, and an empty cell* y *in* st *is allocated for the update, with new cell state* y'*. Then every cell in* st' *has the correct inclusion set:*

- *If* $a \in \mathit{iset}\ x\ st$ *then* $\mathit{iset}\ x'\ st' = \mathit{iset}\ x\ st \cup \{ b \}$ *(Lemma 6.12).*
- *If* $a \notin \mathit{iset}\ x\ st$ *then* $\mathit{iset}\ x'\ st' = \mathit{iset}\ x\ st$ *(Lemmas 6.11 and 6.13).*
- $\mathit{iset}\ y'\ st' = \{ b \}$ *(Lemma 6.10).*
- *All other cells remain empty (Lemma 6.9).*

Proof. The lemmas above show that every element has the correct inclusion set after an update. Together they establish the correctness of the update algorithm. □



$$
\begin{array}{lllll}
\textbf{(1)} & a = \mathit{update}\ \mathit{empty}\ (1 \mapsto 11) & = & \{1 \mapsto 11\} \\
\textbf{(2)} & b = \mathit{update}\ a\ (2 \mapsto 22) & = & \{1 \mapsto 11, 2 \mapsto 22\} \\
\textbf{(3)} & c = \mathit{update}\ b\ (3 \mapsto 33) & = & \{1 \mapsto 11, 2 \mapsto 22, 3 \mapsto 33\} \\
\textbf{(4)} & d = \mathit{update}\ c\ (4 \mapsto 44) & = & \{1 \mapsto 11, 2 \mapsto 22, 3 \mapsto 33, 4 \mapsto 44\}
\end{array}
$$

Figure 5: A chain of updates. The first step updates *empty*, and each subsequent step updates the most recently created array. The intermediate arrays are retained. Each box shows one ESFM cell. **(1)** Since *empty* has $\mathit{code} = 0$, the result (a) has $\mathit{code} = 1$. **(2)** Now a with $\mathit{code} = 1$ is updated, so the result b has $\mathit{code} = 2$. The inclusion set of the new element is 2–2 which contains $\{b\}$; the inclusion set of the element in cell with $\mathit{handle} = a$ is altered so that it now contains codes 1 and 2, representing $\{a, b\}$. **(3, 4)** Each update allocates a new cell for the value of the element, and modifies the inclusion sets of the other elements.

6.4.3. Implementation of updateS

The effect of *updateS* was specified in Section 6.4.1. The implementation uses the parallel combinators, and proceeds in several steps.

1. Find the code and rank of a, using *encodeS* (which requires a *tfold*).
2. Find an available cell to store the new array and element in, using *allocate* (which is a *tfold*).
3. Store the new information (new array, new element) into the allocated cell, and adjust the handle–code map and the inclusion sets in all other cells (apart from empty ones). This step is performed by a *tmap*.

There are two ways *updateS* can fail: the array a may not exist (for example, if a program erroneously deletes a and then tries to update it), and the system memory may be full. In these cases *updateS* returns an error code.

```
updateS :: Array → (Idx ↦ Val) → Bool → EsfaState (Result Array)
updateS a elt nfy =
  do e ← encodeS a
     case e of
       Nothing → return (Left errArrayDoesNotExist)
       Just (acode, arank) →
         do ee ← allocate
            case ee of
              Nothing → return (Left errNoSpace)
              Just h →
                do tmap (f_update h acode arank elt nfy)
                   return (Right h)
```

The *f_update* function alters each cell state in parallel: c is the code of array being updated, r is the rank of c, $(i \mapsto v)$ is the new element, and x is the old value of a cell in the machine state

```
f_update :: Nat → Nat → Nat → (Idx ↦ Val) → Bool → Cell → Cell
f_update h c r (i ↦ v) nfy x =
  let newcode = c + 1
      newrank = r + 1
  in if handle x ≡ h
     then x   -- this is the allocated cell
       {arrDef = True, code = newcode,
```

```
        eltDef = True, rank = newrank,
        low = newcode, high = newcode,
        ind = i, val = v,
        notify = nfy, zombie = False}
    else x   -- alter existing cells
      {low  = if c < low x then low x + 1 else low x,
       high = if c ⩽ high x then high x + 1 else high x,
       code = if c < code x then code x + 1 else code x}
```

Several examples are helpful in following how the algorithm works. Figure 5 shows a simple "chained" sequence of updates. Each element is represented in exactly one cell but it is included in every array which should contain that element. Figure 6 shows several arrays, where some elements are shared and others are not.

6.5. The lookupS operation

The operation *lookupS a i* returns the value of array *a* at index *i*, if it exists. The result is returned as an *Either* type, so that *Right v* indicates a successful lookup giving the value *v*, and *Left m* indicates an unsuccessful lookup with a message *m*. If the array *a* does not exist then an error value *Left errArrayDoesNotExist* is returned. If *a* exists but does not have an element with index *i*, then *Left errNoElement* is returned, which can either be treated as an error or can be used to return the default value for a sparse array. The *lookupS* operation does not change the machine state.

```
lookupS :: Array → Idx → EsfaState (Result Val)
lookupS a i =
  do e ← encodeS a
     case e of
       Nothing → return (Left errArrayDoesNotExist)
       Just (acode, arank) →
         do tmap (f acode i)
            ee ← tfold g h
            case ee of
              Nothing → return (Left errNoElement)
              Just (v, r) → return (Right v)
   where f c i x = x {mark = eltDef x ∧ ind x ≡ i
                                  ∧ low x ⩽ c ∧ c ⩽ high x}
```

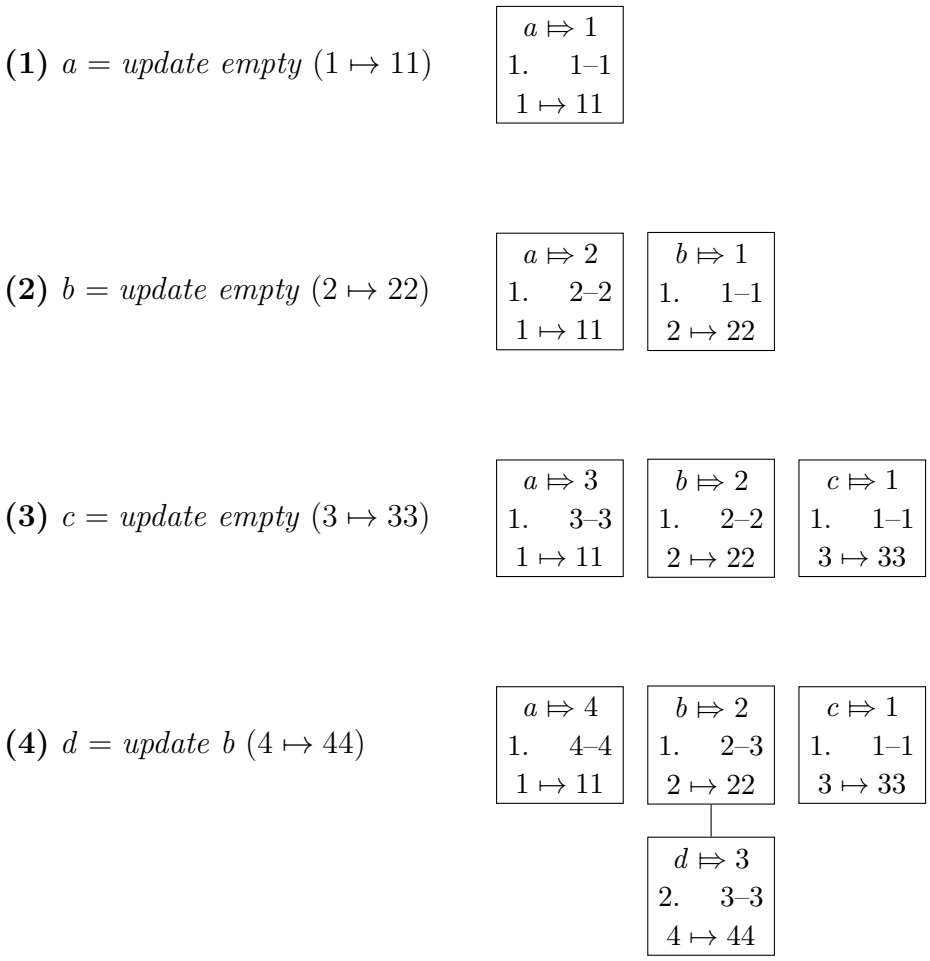


Figure 6: Shared and unshared elements.

$$
\begin{array}{l}
g\ x = \textbf{if}\ \mathit{mark}\ x\ \textbf{then}\ \mathit{Just}\ (\mathit{val}\ x, \mathit{rank}\ x) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{Nothing} \\
h\ \mathit{Nothing}\ y = y \\
h\ x\ \mathit{Nothing} = x \\
h\ (\mathit{Just}\ (v1, r1))\ (\mathit{Just}\ (v2, r2)) = \\
\quad \textbf{if}\ r1 \geqslant r2\ \textbf{then}\ \mathit{Just}\ (v1, r1)\ \textbf{else}\ \mathit{Just}\ (v2, r2)
\end{array}
$$

1. The code for the array with $\mathit{handle} = a$ is found by an associative search using $c \leftarrow \mathit{encodeS}\ a$.
2. The *mark* flag is set in each cell where $\mathit{ind} \equiv i \wedge \mathit{lo} \leqslant c \wedge c \leqslant \mathit{hi}$. This is a parallel map taking 1 clock cycle. The marked cells include the elements of the array as well as shadowed elements; these are called the *candidates*.
3. The candidate with the highest rank is fetched, using a parallel fold, finding the right value and discarding any shadowed elements. The rank of an element is the number of updates, starting from empty, that created the element, so the candidate with the highest rank is the correct result. The rank field cannot overflow, as its value can never exceed the number of cells in the machine.

6.6. The deleteS operation

The *deleteS a* operation is similar to *update*, except that it decrements codes and shrinks intervals, rather than incrementing codes and expanding intervals.

6.6.1. Effect of deleteS

Just as with to *updateS*, there are several cases that determine the action a cell containing an element will perform.

- **Shift:** If $c < \mathit{low}\ x$ then both the *low* and *high* fields are decremented. The representation of the inclusion set shifts down by a unit, but the size of the set remains the same.

- **Shrink:** if $\mathit{low}\ x \leqslant c \leqslant \mathit{high}\ x$ then *low* is unchanged but *high* is decremented. This shrinks the size of the inclusion set of x to accommodate the removal of the array being deleted, and if it results in $\mathit{high}\ x < \mathit{low}\ x$ then the element is inaccessible.

- **Unchanged:** if *high* $x < c$ then both the *low* and *high* fields are left unchanged.

To implement this, each cell performs a local computation:

- Every cell x where *arrDef* x = *True* either adjusts its *code* field (if *handle* $x \not\equiv a$) or sets *arrDef* x' = *False* (if *handle* $x \equiv a$).

$$\begin{aligned} &\mathit{arrDef}\ x' = \mathit{arrDef}\ x \wedge \mathit{handle}\ x \not\equiv a \\ &\mathit{code}\ x' \quad = \textbf{if}\ c < \mathit{code}\ x\ \textbf{then}\ \mathit{code}\ x - 1\ \textbf{else}\ \mathit{code}\ x \end{aligned}$$

- Every cell x where *eltDef* x = *True* adjusts its representation of the inclusion set. If its *handle* matches a, then its *arrDef* field is set to False; this deletes the array (but not necessarily the element). The cell calculates a temporary Boolean *remains* which is True if and only if the element should remain in memory after the deletion, and *deleteLater* which is True if the element will be deleted but needs to be retained until the host processor is notified (this enables a storage manager to cope with boxed values in the ESF machine).

$$\begin{aligned} &\mathit{low}\ x' \quad\quad = \textbf{if}\ c < \mathit{low}\ x\ \ \textbf{then}\ \mathit{low}\ x - 1\ \ \textbf{else}\ \mathit{low}\ x \\ &\mathit{high}\ x' \quad\quad = \textbf{if}\ c \leqslant \mathit{high}\ x\ \textbf{then}\ \mathit{high}\ x - 1\ \textbf{else}\ \mathit{high}\ x \\ &\mathit{remains} \quad\quad = \mathit{eltDef}\ x \wedge \mathit{low}\ x' \leqslant \mathit{high}\ x' \\ &\mathit{deleteLater} = \mathit{eltDef}\ x \wedge \neg\ \mathit{remains} \wedge \mathit{notify}\ x \end{aligned}$$

Inclusion intervals on the path of the array being deleted have their *high* field decremented, while their *low* field remains unchanged. As a result, it is possible that an element will have *low* > *high*. If this happens, that element is inaccessible.

6.6.2. Correctness of deleteS

A deletion removes an array from the handle-code mapping, and it also removes the array from the inclusion set of all its elements.

Consider the deletion of an existing array a in state st. In the resulting state st' the array a is undefined: *arrDef* $y' \equiv$ *False* in the cell y' where *handle* $y' \equiv a$. Thus *lookupS* a i will fail in state st'. For an arbitrary cell x in state st, the *deleteS* algorithm defines *code* x' = **if** $c <$ *code* x **then** *code* $x - 1$ **else** *code* x. Therefore, if an arbitrary array p is defined in state st',

encode p st′ =
  if $c <$ *encode p st* **then** *encode p st* $- 1$ **else** *encode p st*

The calculation of the effect of deletion uses the following properties of *encode*.

Lemma 6.15. *Suppose array a with code c is being deleted; x is a cell state containing an element in machine state st, and deleteS transforms it to x′ in st′. Let p be an arbitrary array that exists in st.*

1. *If encode p st′* $<$ *c then encode p st′* $\equiv$ *encode p st.*

   Proof. Assume *encode p st′* $<$ *c* and consider the following exhaustive cases.

   - If $c <$ *encode p st* then *encode p st′* $\equiv$ *encode p st* $- 1$ so $c <$ *encode p st′* $+ 1$; therefore $c \leqslant$ *encode p st′* which violates the assumption.
   - If $c \equiv$ *encode p st*, then $p \equiv a$ and the algorithm sets *arrDef x* $=$ *False*, so *encode p st′* is undefined.
   - If $c >$ *encode p st* then *encode p st′* $\equiv$ *encode p st.*

   □

2. *If* $c \leqslant$ *encode p st′ then encode p st′* $=$ *encode p st* $- 1$.

   Proof. Assume $c \leqslant$ *encode p st′* and consider the following exhaustive cases.

   - If $c <$ *encode p st*, then *encode p st′* $=$ *encode p st* $- 1$.
   - If $c \equiv$ *encode p st*, then $p \equiv a$ and the algorithm sets *arrDef x* $=$ *False*, so *encode p st′* is undefined.
   - If $c >$ *encode p st*, then *encode p st′* $=$ *encode p st* so $c >$ *encode p st′* which violates the assumption.

   □

The following lemmas show that a cell is handled correctly in each of the three cases. The lemmas for cases Shift and Unchanged are analogous to those for *updateS*. The crucial case is Shrink: if a is in the inclusion set for an element, then after the deletion it is removed and the size of the inclusion set is reduced. If this causes *low x′* $>$ *high x′* then the inclusion set is empty and the cell is inaccessible.

Lemma 6.16 (Case Delete–Shift)**.** *In this case,* $c < low\ x$.

Proof. Since $c < low\ x$ the algorithm defines $low\ x' = low\ x - 1$ and $high\ x' = high\ x - 1$.

$$
\begin{array}{ll}
& iset\ x'\ st' \\
\equiv & \langle \text{ Definition 6.5 } \rangle \\
& \{\, p :: Nat \mid low\ x' \leqslant encode\ p\ st' \leqslant high\ x' \,\} \\
\equiv & \langle \text{ Substitute } x \text{ for } x' \rangle \\
& \{\, p :: Nat \mid low\ x - 1 \leqslant encode\ p\ st' \leqslant high\ x - 1 \,\} \\
\equiv & \langle\ c < low\ x \text{ so } c \leqslant low\ x - 1 \leqslant encode\ p\ st'\text{; Lemma 6.15 (2) } \rangle \\
& \{\, p :: Nat \mid low\ x - 1 \leqslant encode\ p\ st - 1 \leqslant high\ x - 1 \,\} \\
\equiv & \langle \text{ Add 1 to each side of inequality } \rangle \\
& \{\, p :: Nat \mid low\ x \leqslant encode\ p\ st \leqslant high\ x \,\} \\
\equiv & \langle \text{ Definition 6.5 } \rangle \\
& iset\ x\ st
\end{array}
$$

□

Lemma 6.17 (Case Delete–Shrink)**.** *Let* x *be a cell in state* st*, and let* a *be an array with code* c*. Suppose deleteS* a *is executed, resulting in a new state* st' *where the cell is transformed to* x'*. If* $low\ x \leqslant c \leqslant high\ x$ *then* $iset\ x'\ st' = iset\ x\ st - \{\, a \,\}$.

Proof. Since $\neg\ (c < low\ x)$, the algorithm defines $low\ x' = low\ x$. Since $c \leqslant high\ x$, it defines $high\ x' = high\ x - 1$. Some of the code values in the intervals may be greater than c and will be decremented, while others may be less than or equal to c and will remain unchanged. Therefore the calculation splits the set into several subsets, similar to the proof for the Expand case for *updateS*. The array a is removed from the inclusion set of the modified cell x', which is calculated as follows:

$$
\begin{array}{ll}
& iset\ x'\ st' \\
\equiv & \langle \text{ Definition 6.5 } \rangle \\
& \{\, p :: Nat \mid low\ x' \leqslant encode\ p\ st' \leqslant high\ x' \,\} \\
\equiv & \langle \text{ Split into sets based on relationship with } c \rangle \\
& \langle \text{ (1) set of arrays whose codes are unchanged } \rangle
\end{array}
$$

$\langle$ (2) set of arrays with decremented codes $\rangle$
$\langle$ Code c in x is overwritten in x' $\rangle$
$\{p :: Nat \mid low\ x' \leqslant encode\ p\ st' < c\}$
$\cup \{p :: Nat \mid c \leqslant encode\ p\ st' \leqslant high\ x'\}$

$\equiv$ $\langle$ Substitute x for x' $\rangle$
$\{p :: Nat \mid low\ x \leqslant encode\ p\ st' < c\}$
$\cup \{p :: Nat \mid c \leqslant encode\ p\ st' \leqslant high\ x - 1\}$

$\equiv$ $\langle$ Substitute st for st'.(1) Lemma 6.15 (1).(2) Lemma 6.15 (2) $\rangle$
$\{p :: Nat \mid low\ x \leqslant encode\ p\ st < c\}$
$\cup \{p :: Nat \mid c \leqslant encode\ p\ st - 1 \leqslant high\ x - 1\}$

$\equiv$ $\langle$ (2) add 1 to each side of inequality $\rangle$
$\{p :: Nat \mid low\ x \leqslant encode\ p\ st < c\}$
$\cup \{p :: Nat \mid c + 1 \leqslant encode\ p\ st \leqslant high\ x\}$

$\equiv$ $\langle$ Add and subtract singleton set at c $\rangle$
$\{p :: Nat \mid low\ x \leqslant encode\ p\ st < c\}$
$\cup \{p :: Nat \mid encode\ p\ st \equiv c\}$
$- \{p :: Nat \mid encode\ p\ st \equiv c\}$
$\cup \{p :: Nat \mid c + 1 \leqslant encode\ p\ st \leqslant high\ x\}$

$\equiv$ $\langle$ Reorder (3) and (4) as they are disjoint $\rangle$
$\{p :: Nat \mid low\ x \leqslant encode\ p\ st < c\}$
$\cup \{p :: Nat \mid encode\ p\ st \equiv c\}$
$\cup \{p :: Nat \mid c + 1 \leqslant encode\ p\ st \leqslant high\ x\}$
$- \{p :: Nat \mid encode\ p\ st \equiv c\}$

$\equiv$ $\langle$ (4) $c = encode\ a\ st$ $\rangle$
$\{p :: Nat \mid low\ x \leqslant encode\ p\ st < c\}$
$\cup \{p :: Nat \mid encode\ p\ st \equiv c\}$
$\cup \{p :: Nat \mid c + 1 \leqslant encode\ p\ st \leqslant high\ x\}$
$- \{a\}$

$\equiv$ $\langle$ Combine set comprehensions $\rangle$
$\{p :: Nat \mid low\ x \leqslant encode\ p\ st \leqslant high\ x\}$
$- \{a\}$

$\equiv$ $\langle$ Definition 6.5 $\rangle$
$iset\ x\ st - \{a\}$

□

Lemma 6.18 (Case Delete–Unchanged)**.** *In this case, high* $x < c$*.*

Proof. Since *high* $x < c$ the algorithm defines *low* $x' =$ *low* x and *high* $x' =$ *high* x.

$$
\begin{array}{ll}
& \mathit{iset}\ x'\ \mathit{st}' \\
\equiv & \quad \langle\ \text{Definition 6.5}\ \rangle \\
& \{\, p :: \mathit{Nat} \mid \mathit{low}\ x' \leqslant \mathit{encode}\ p\ \mathit{st}' \leqslant \mathit{high}\ x' \,\} \\
\equiv & \quad \langle\ \text{Substitute } x \text{ for } x'\ \rangle \\
& \{\, p :: \mathit{Nat} \mid \mathit{low}\ x \leqslant \mathit{encode}\ p\ \mathit{st}' \leqslant \mathit{high}\ x \,\} \\
\equiv & \quad \langle\ \text{Lemma 6.15 (1}\ \rangle \\
& \{\, p :: \mathit{Nat} \mid \mathit{low}\ x \leqslant \mathit{encode}\ p\ \mathit{st} \leqslant \mathit{high}\ x \,\} \\
\equiv & \quad \langle\ \text{Definition 6.5}\ \rangle \\
& \mathit{iset}\ x\ \mathit{st}
\end{array}
$$

□

6.6.3. Implementation of deleteS

The code for the array with *handle* $= a$ is found by *encodeS*, and then a *tmap* adjusts each cell is updated in parallel. This implementation silently returns () if the array being deleted did not exist; it is straightforward to return an error message instead.

```
deleteS :: Array → EsfaState ()
deleteS a =
  do e ← encodeS a
     case e of
       Nothing → return ()
       Just (acode, arank) → tmap (fDelete a acode)


fDelete :: Array → Nat → Cell → Cell
fDelete a c x =
  let low' = if c < low x then low x − 1 else low x
      high' = if c ⩽ high x then high x − 1 else high x
      remains = eltDef x ∧ low' ⩽ high'
      deleteLater = eltDef x ∧ ¬ remains ∧ notify x
  in x {arrDef = arrDef x ∧ handle x ≢ a,
```

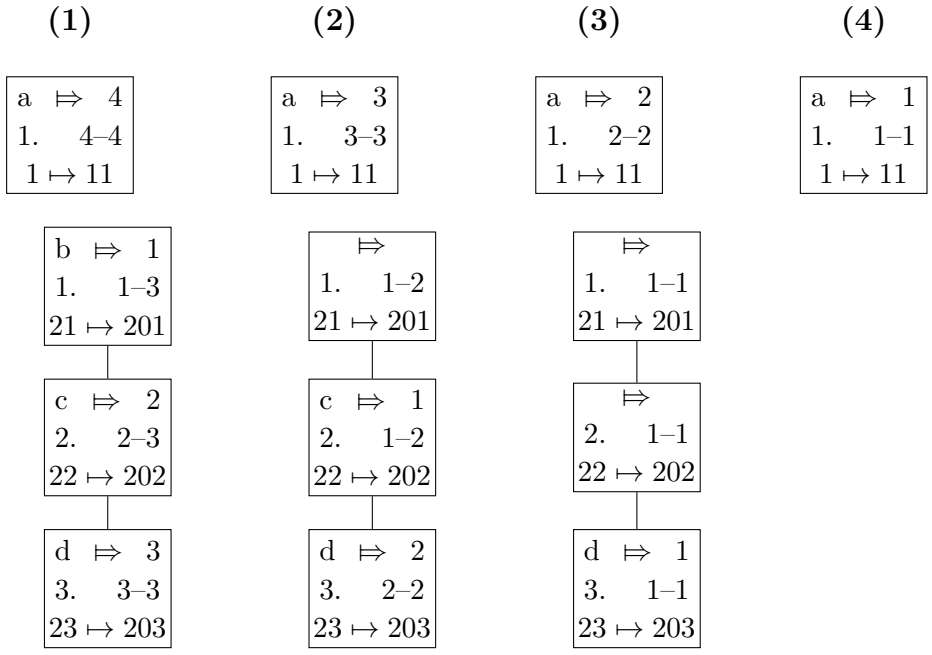


Figure 7: Deleting an array leaves shared elements intact. **(1)** Several arrays are created, some with shared elements: $a \leftarrow$ *updateS empty* $(1 \mapsto 11)$ *False*, then $b \leftarrow$ *updateS empty* $(21 \mapsto 201)$ *False*, then $c \leftarrow$ *updateS* b $(22 \mapsto 202)$ *False*, and $d \leftarrow$ *updateS* c $(23 \mapsto 203)$ *False*. The final representation is shown. **(2)** *deleteS* b removes array b but the element $21 \mapsto 201$ is shared with other arrays, so it remains. **(3)** *deleteS* c removes array c but again all elements are shared and remain. **(4)** *deleteS* d removes array d. At this point three elements no longer contain any array in their inclusion set. The corresponding three cells contain no useful information (no array, no element) so they are immediately reclaimed and are available for use in a future *updateS*.

$$
\begin{aligned}
&\mathit{code} = \textbf{if } c < \mathit{code}\ x \textbf{ then } \mathit{code}\ x - 1 \textbf{ else } \mathit{code}\ x,\\
&\mathit{low} = \mathit{low}', \mathit{high} = \mathit{high}',\\
&\mathit{eltDef} = \mathit{eltDef}\ x \wedge \mathit{remains},\\
&\mathit{notify} = \mathit{notify}\ x \wedge \neg\ \mathit{deleteLater},\\
&\mathit{zombie} = \mathit{zombie}\ x \vee \mathit{deleteLater}\}
\end{aligned}
$$

Figure 7 shows how a sequence of deletions can leave all the array elements undisturbed until a final deletion removes the last bit of sharing; as a result many cells are reclaimed simultaneously.

Deleting an array is efficient in both time and space. The operation takes a small constant amount of time. Furthermore, it identifies every cell that has

become inaccessible, apart from shadowing. After a deletion, two properties hold:

- Every element that is inaccessible (i.e. there does not exist a future *lookupS* that could access it) is reclaimed, provided that the element is not shadowed.
- Every element that is accessible (i.e. it is possible for a future *lookupS* to fetch it) is retained.

7. Complexity

If it can be proved that a functional array is never accessed again after it has been updated, then the compiler can implement that update by overwriting the old array element, achieving the efficiency of imperative arrays. This approach can be supported using compiler analysis, the type system, or monads [18]. It works well for algorithms that were designed in the first place for imperative arrays, but it does not help when the flexibility of pure functional arrays is needed. This approach has been developed extensively, with efficient array accesses for parallel processors [19]. However, our concern is with general functional arrays where there is no restriction on the usage of the operations.

In discussing the complexity of algorithms, it is essential to be clear about what machine model we are using. The Random Access Machine is a theoretical model that corresponds closely to standard von Neumann computer architectures. The Random Access Machine has a Random Access Memory, which allows any memory location to be accessed in $O(1)$ time, regardless of the address. This differs from a Turing Machine, which takes account of locality and makes it more expensive to access distant locations.

Several issues prevent a straightforward implementation of ESF array operations on a Random Access Machine in unit time and space. For update to take $O(1)$ space, array elements must be shared. Since a shared element belongs to several arrays, there cannot be a simple function from an array handle and index to the address of the element. However, a linked data structure such as a tree or list cannot be traversed in $O(1)$ time. Therefore it appears that a RAM cannot achieve $O(1)$ time and space for *update* and *lookup*, and no such implementation has been found, despite many efforts.

cost model	RAM	ESFM
unit time cycle	$O(1)$	$O(1)$
gate delay	$O(\log n)$	$O(\log n \times \log(\log n))$
propagation delay	$O(\sqrt{n})$	$O(\sqrt{n})$

Table 3: Comparison of cycle time for RAM and ESFM of size n. In the gate delay model, the RAM needs a tree of multiplexers with delay $O(\log n)$ to decode addresses; the ESFM uses a more complex tree to perform the folds and scans, and the delay is slightly larger. For large circuits the propagation delay becomes larger than the gate delay.

Conjecture There does not exist an implementation of functional arrays (*update* and *lookup*) such that every operation always takes $O(1)$ time and space on a Random Access Machine.

The conjecture says that the performance of imperative arrays cannot be attained by functional arrays, without placing restrictions on how the arrays are used.

In complexity theory, it is common to consider the costs of operations using unbounded memory size and unbounded word size. In real computer systems, there is a fixed word size k and memory size limited by 2^k. We can still analyze the complexity of an algorithm running in a bounded system; the algorithm will fail if it exceeds the bounds. In an unbounded system, many of the complexity measures have an additional factor of $\log(\log N)$ because the index size grows as the memory grows. In this section, we will consider machines of bounded size, but the essential result (unit time Array operations with the same circuit size and delay as for RAM) remains valid for the unbounded case as well.

The system presented in this paper uses its massively parallel digital circuit to do exactly what the conjecture says is impossible on a RAM architecture.

The time required by a computer system to perform a computation is $k \times p$, where k is the number of clock cycles required and p is the clock period (the reciprocal of the clock speed). It is straightforward to find k. In calculating p it is essential to state clearly what cost model for the circuit is being used. It is a fallacy to say that each instruction in a RAM takes $O(1)$ time but the ESFM has a tree circuit so its clock period is $O(\log n)$, and then to use these figures to compare their speeds.

The ESFM system performs all the operations in Table 2 in a constant

number of clock cycles (and the constant is small, typically 2 or 3 cycles). The number of cycles does not depend on the past history of operations and does not assume any restriction on the usage of arrays. Memory usage is also optimal: each *updateS* allocates exactly one cell and there is never any loss of sharing. Each *deleteS* identifies every cell that is inaccessible, apart from shadowing, in constant time. (Reclaiming shadowed cells that are inaccessible requires additional work.)

Does the ESFM achieve its optimal $O(1)$ cycle count at the expense of an asymptotically longer clock period, or at the expense of an asymptotically larger circuit? Table 3 shows that it does not, and Table 4 shows why.

The ESFM contains $O(n)$ flip flops and $O(n)$ logic gates for a system with n cells. There are many technologies for implementing a random access memory, and some do not use flip flops or logic gates, but equivalent devices are used. A RAM contains $O(n)$ "flip flop equivalents" and $O(n)$ "logic gate equivalents", so its size is asymptotically the same as the ESFM.

The clock period must be expressed according to a *cost model* that defines what aspects of the hardware are being considered, and what aspects are abstracted away. Some reasonable cost models include:

- *Unit time cycle.* Assume that each clock cycle is one unit of time. This model is commonly used in analysis of algorithms: the average case time for Quicksort is $O(n \times \log n)$ *assuming the unit time cycle model.*

- *Gate delay.* Find the critical path in the circuit and measure its gate delay, and multiply by a safety factor slightly more than 1. The RAM needs to decode the memory address, which requires a gate delay of $O(\log n)$ for a memory of size n. The ESFM has a gate delay proportional to the height of the tree, which is also $O(\log n)$. A subtle point is that *lookupS* requires a comparison of two ranks in each tree node; this leads to a gate delay of $O(\log n \times \log n)$ with slow ripple comparators or $O(\log n \times \log(\log n))$ with fast comparators.

- *Propagation delay.* Storage elements take physical space, and accessing them requires signals to travel a distance $O(\sqrt{n})$ for a memory of size n. RAM and ESFM both have a propagation delay of $O(\sqrt{n})$.

Table 3 compares the time complexity of a clock cycle for ESFM and RAM. The choice of cost model depends on what aspects of a system seem most important. For small circuits the gate delay dominates the cycle time,

	logic gate calculations	useful calculations
RAM	$O(n)$	$O(1)$
ESFM	$O(n)$	$O(n)$

Table 4: Usage of calculations in RAM and ESFM for a memory access or sweep.

and the gate delay model, which ignores the lengths of wires, is reasonable. For large circuits, the wire length dominates. The important point is that *comparisons should use the same cost model*.

It is not the case that ESFM performs as well as RAM according to every cost model; in particular, it is likely to require more power. In VLSI circuits, there is a residual power consumption that remains constant, but further power is required for each transition that changes the value of a signal. This means that the parallel calculations in the cells of the ESFM will require more power than the corresponding bits in a RAM.

How does ESFM achieve the same time complexity as imperative arrays, while (it is conjectured) a RAM cannot? The answer lies in the address decoder in a RAM. This is a tree of multiplexers; each level in the tree uses one bit of the address to select which subtree contains the memory word being addressed. The tree has a gate delay of $O(\log n)$ and it contains $O(n)$ logic gates, and all it does is to select one word based on an address *even though every flip flop and every logic gate in the tree performs a calculation.* Thus the RAM performs $O(n)$ logic gate calculations in a clock cycle in order to perform $O(1)$ *useful* work. The ESFM also performs $O(n)$ logic gate calculations, but these enable it to perform the $O(n)$ calculations on codes and inclusion sets that are required to implement the array operations in a fixed number of cycles (Table 4).

8. Implementation and parallel platforms

The complexity analysis in the previous section assumes that *sweep* is implemented as a digital circuit. It is also possible to implement the ESFA system on a standard parallel platform; this will not achieve optimal performance but it is still possible to achieve fast array operations. This section discusses the issues in using a parallel computer, and describes an implementation that has been carried out using a GPGPU.

An alternative to implementing the circuit in dedicated hardware is to

emulate it using a field programmable gate array (FPGA). This is a hardware device consisting of an array of small scale units (general logic functions, small memories, and the like) along with a programmable interconnection network that can be used to connect the small units to form a digital circuit. The advantage of an FPGA is that it is an off-the-shelf component; the disadvantage is that it is slower and less dense than a custom VLSI chip.

In practice, the hardest part of using an FPGA is often interfacing it with a host computer. This is not at all standardized, and it requires a combination of communication software, device driver software, and interfacing hardware that is laid out on the FPGA chip itself. Some FPGAs are tightly coupled with a processor on the same chip, but others require slower I/O connections.

An FPGA platform might benefit by tiling. The idea is that a small number k of tree machine cells are mapped onto one physical cell. Each sweep operation would require k cycles to allow each physical cell to emulate its virtual ones. This increases the size of the memory without requiring the extra tree nodes and logic functions, with a commensurate slowdown.

General purpose graphical processing units (GPGPU) [12] are a form of fine-grain multicore processor targeted at a restricted form of data parallelism. Originally these devices were intended for graphics algorithms, but they have found increasing usage for more general data parallel computation, including circuit simulation. Current GPGPU chips have many small processor cores (on the order of 1000), and offer good performance for regular applications.

The system described in this paper has been implemented in several ways:

- The laws for arrays provide an executable specification in Haskell.
- The *sweep* function, parallel combinators, and implementations of *updateS*, *lookupS* and *deleteS* given in this paper comprise part of a high level circuit simulation written in Haskell. The rest of the code, documentation, examples, and test cases, are available on the web [13].
- A synthesizable circuit specification in the Hydra hardware description language [10] allows simulation at the level of logic gates.
- A portable parallel implementation has been written in C and CUDA and tested on an NVidia GeForce GTX 590 GPGPU with 512 cores and 1.22 GHz clock speed [20]. Performance was measured using a

test sequence of 50,000 operations that were generated randomly, running on an ESFM system with 8K cells. The operation time including memcpy (overheads introduced because of the GPU architecture) for *updateS* is 18.8 μs (microseconds); for *lookupS* is 16.5 μs; for *deleteS* is 11.9 μs. Using the nvprof tool, which measures the calculation time but does not include the memcpy overhead, the time for *updateS* is 8.3 microseconds; for *lookupS* is 8.1 microseconds; and for *deleteS* is 3.5 microseconds.

The *sweep* circuit can perform other parallel computations, and is particularly useful for associative searching. Thus the API can be extended to support general content-addressable parallel processing [3].

Most algorithms have been developed for languages that provide imperative arrays. These algorithms use chained updates to arrays (i.e. once an array is updated it is never used again), and there is no particular advantage in replacing them with ESFA. The most promising applications for ESFA make essential use of sharing with multi-threaded access and/or sparse traversal and searching.

Functional arrays provide a flexible undo/redo facility. Suppose a program records transactions by updating an ESFA, keeping the most recent array as the current state. The program can revert to any previous state simply by accessing the corresponding state; there is no need to rebuild any data structures.

ESFA can represent the environment in a lambda calculus reducer. Each variable is represented by a unique integer, each environment is an ESFA, and the initial environment is *empty*. Obtaining the value of a variable is performed by a lookup, while extending the environment for a beta reduction requires an update:

$$eval\ (Var\ x)\ env = lookup\ env\ x$$
$$eval\ (App\ (\lambda x \rightarrow e1)\ e2)\ env = eval\ e1\ (update\ env\ x\ e2)$$

An SECD machine has been developed using this method, and has been used to test the ESFA system running on a simulated circuit.

Programming language implementations sometimes use complex data structures to represent the evaluation stack as well as variable environments. Sometimes this can be inefficient: a dynamically bound variable may require a list traversal to find its value, while an Array could obtain the value

in one operation. Constraint solving algorithms use backtracking or coroutines to explore several alternative paths, and functional arrays may prove useful for representing the constraint sets.

9. Conclusion

This paper conjectures that a standard computer based on the Random Access Machine model cannot implement functional array operations in constant time, unless the program makes restricted use of the operations.

Despite this conjecture, the paper gives the design of a digital circuit implementing a smart memory and a set of algorithms that use that memory to achieve the goal: functional array lookup and update, as well as several other operations supporting extensible and sparse arrays, can all be implemented in unit time. Each operation requires a small constant number of steps, and there is no restriction on the past history of updates. A lookup or update always takes exactly the same time, regardless of how much sharing there is among all the existing arrays.

The algorithms are implemented in Haskell, and the circuit is specified using Hydra, a hardware description language embedded in Haskell. There is a parallel GPU implementation that gives good performance, and the algorithm also runs on the simulated circuit.

The implementation of ESFA relies on a fine-grain data parallel platform to hold the array memory. Each operation involves a small amount of calculation in every location in the entire memory. The system performs extra work—a small amount of arithmetic in every location—and then mitigates the extra work with massive parallelism—ideally, a processing element in every location.

However, there is a more insightful way to think of the system. Consider a sequential program running on standard hardware, with a RAM memory. Programmers think of the RAM as just doing a little work on the word that is accessed (if they think of the RAM at all). However, a RAM is a digital circuit that actually has to perform an enormous amount of work on every access (not exactly a computation on every location, but that is a fair intuition). We think of the RAM as performing a small amount of work because most of its work is *wasted and not worth thinking about*. The ESFM does more work than the RAM, but only by a constant factor, and it uses this work to enable it to support a useful data structure more efficiently than a RAM can.

ESF arrays are not just a theoretical novelty. Although the fastest host for the ESFM would be an application specific integrated circuit (ASIC), which has not been implemented, the current GPU implementation is fast enough for some applications, and is easily portable, supporting future research in purely functional data structures.

Acknowledgments. Cordelia Hall implemented the GPU program in C+CUDA and the SECD interpreter. The anonymous reviewers provided many valuable suggestions, including improvements to the notation and presentation of the algorithms. Roseanne English provided useful comments on the text.

References

[1] J. O'Donnell, Data parallel implementation of Extensible Sparse Functional Arrays, in: Parallel Architectures and Languages Europe, volume 694 of *LNCS*, Springer-Verlag, 1993, pp. 68–79.

[2] J. T. O'Donnell, Extensible sparse functional arrays with circuit parallelism, in: Proc. 15th International Symposium on Principles and Practice of Declarative Programming, ACM, 2013, pp. 133–144. DOI 10.1145/2505879.2505891.

[3] C. Foster, Content Adressable Parallel Processors, Cengage Learning EMEA, 1976. ISBN-13: 978-0442224332.

[4] G. R. Andrews, D. P. Dobkin, Active data structures, in: ICSE'81: Proc. 5th Int. Conf. on Software Engineering, IEEE Press, 1981, pp. 354–362.

[5] G. Bloom, G. Parmer, B. Narahari, R. Simha, Shared hardware data structures for hard real-time systems, in: Proceedings of the Tenth ACM International Conference on Embedded Software, EMSOFT '12, ACM, New York, NY, USA, 2012, pp. 133–142.

[6] T. St. John, J. B. Dennis, G. R. Gao, Massively parallel breadth first search using a tree-structured memory model, in: Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '12, ACM, New York, NY, USA, 2012, pp. 115–123. doi:10.1145/2141702.2141715.

[7] A. Firoozshahian, A. Solomatnikov, O. Shacham, Z. Asgar, S. Richardson, C. Kozyrakis, M. Horowitz, A memory system design framework: creating smart memories, in: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09, ACM, New York, NY, USA, 2009, pp. 406–417.

[8] T. R. Martinez, Smart memory architecture and methods, Future Generation Computer Systems 6 (1990) 145–162.

[9] A. Triossi, S. Orlando, A. Raffaetà, T. Frühwirth, Compiling CHR to parallel hardware, in: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming, PPDP '12, ACM, New York, NY, USA, 2012, pp. 173–184.

[10] J. O'Donnell, Overview of Hydra: A concurrent language for synchronous digital circuit design, in: Proceedings 16th International Parallel & Distributed Processing Symposium, IEEE Computer Society, 2002, p. 234 (abstract). Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications—PDSECA.

[11] NVIDIA CUDA C Programming Guide, version 4.2 ed., NVIDIA Corp., 2012.

[12] J. D. Owens, et al., A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (2007) 80–113.

[13] J. T. O'Donnell, ESF Arrays, School of Computing Science, University of Glasgow, 2014. `www.dcs.gla.ac.uk/~jtod/research/esfa`.

[14] P. J. Landin, The mechanical evaluation of expressions, Computer Journal 6 (1964) 308–320. DOI 10.1093/comjnl/6.4.308.

[15] C. Okasaki, Purely functional data structures, Cambridge University Press, 1999.

[16] S. Peyton Jones, S. Marlow, C. Elliott, Stretching the storage manager: weak pointers and stable names in Haskell, in: P. Koopman, C. Clack (Eds.), 11th International Workshop on Implementation of Functional Languages, volume 1868 of *LNCS*, Springer, 1999, pp. 37–58.

[17] J. O'Donnell, A correctness proof of parallel scan, Parallel Processing Letters 4 (1994) 329–338.

[18] S. L. Peyton Jones, P. L. Wadler, Imperative functional programming, in: Proc. 20th ACM Symposium on Principles of Programming Languages (POPL), 1993, pp. 71–84.

[19] M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonnell, V. Grover, Accelerating Haskell array codes with multicore GPUs, in: Declarative Aspects of Multicore Programming, ACM, 2011.

[20] J. O'Donnell, C. Hall, S. Monro, Active data structures on GPGPUs, in: The 6th Workshop on UnConventional High Performance Computing, number 8374 in LNCS, Springer, 2013.