Arrays, Non-Determinism, Side-Effects, and Parallelism: A (Pseudo-)Functional Perspective

Paul Hudak
Yale University
Department of Computer Science
August 1987

Abstract

Purely functional incremental updates to arrays, executed in a non-deterministic manner, are shown to achieve the same effect (in both efficiency and functionality) as parallel assignments to imperative arrays. The strategy depends only on the ability of a compiler to recognize that incremental updates to functional arrays can be done destructively (an optimization often called “copy-avoidance” of “single-threaded arrays”). Special “pseudo-functional syntax” is introduced that captures typical array-update patterns. This syntax falls somewhere between the purely functional and (im)purely imperative, and makes the inferencing problem fairly easy. If nothing else, the work represents an interesting intellectual exercise in the relationship between non-determinism, side-effects, and parallelism, and tends to blur the traditionally clear-cut distinction between functional and imperative languages.

This research was supported in part by the US Government under DOE grant FG02-86ER25012.

A preliminary version of this paper is to appear in the Proceedings of the LANL/MCC Graph Reduction Workshop held in Santa Fe, New Mexico, Sept. 29 - Oct. 1, 1986.
1 Introduction

By now most language theorists and experienced programmers know how to express imperative programs in a purely functional, or declarative, style. For example, loops are replaced with tail-recursions (in which actual parameters in the recursive call replace assignment to loop variables); multiple assignments to a single variable are replaced with the creation of new variables in “let expressions” (or eliminated entirely through the construction of larger expressions); and assignments to aggregate data structures such as arrays are replaced with some equivalent expression using “functional” (i.e., “immutable”) aggregates. It is this last example, in particular assignment to arrays, and more particularly parallel assignment to arrays, that we investigate in depth in this paper. Our goal is to achieve with functional arrays the same observable behavior and run-time efficiency that one gets using arrays in an imperative language. Such efforts for sequential implementations have in fact already been fairly successful, and thus in this paper we concentrate on parallel implementations.

The reason that assignment to cells in an array (as opposed to individual variables) is being singled-out for investigation is that such assignments are a common source of parallelism in conventional imperative languages; i.e., it is possible to perform concurrent operations to different parts of the same array, often with a deterministic result (usually by ensuring that no parallel assignments are made to the same location). Unfortunately, the “standard” translation of a sequential imperative program with mutable arrays into a functional program with immutable arrays results in a sequentialization, via data dependencies, of the array updates – after all, that is what the semantics of the sequential imperative language dictates. We demonstrate this by a simple example.

Consider the following imperative program:

```
begin
  a[i] := e_1;
  a[j] := e_2
end
```

A sequence of assignments such as this to a traditional mutable array can be effectively modelled by a nested set of incremental updates to an immutable array in a functional language. Thus we can model the above program with:

```
update(update(a, i, e_1), j, e_2)
```

where `update(a, i, x)` returns an array just like `a` except that `a[i] = x`.

(The knowledgeable reader will notice, of course, that to make incremental updates to immutable arrays practical, in either this sequential case or the parallel case to be described

---
1Functional programming advocates argue that the resulting programs are improvements over their imperative counterparts [2], but these issues will not be pursued here.
Now suppose we augment our imperative language with parallel constructs, thus allowing parallel assignments to arrays. For example, a simple modification to the previous imperative program yields:

\[
\text{parbegin} \\
\quad a[i] := e_1; \\
\quad a[j] := e_2 \\
\text{parend}
\]

Yet the functional solution presented earlier does not properly model this program, in that the nested updates in the functional solution are *implicitly ordered* (namely, innermost first), thus failing to model the parallel (and possibly non-deterministic) behavior of the array assignments. So the question remains: how do we properly model parallel assignment to arrays?

To answer this question, we point out that parallel assignment to arrays in an imperative language introduces a form of *non-determinism*, which is manifested as different intermediate states of the array, whether or not the final result is deterministic. For example, in the above program, consider the value (i.e. state) of the array \(a\) at the point when the second assignment is about to take place – the array may have one of two values, depending on whether or not the first assignment has already occurred. If \(i = j\) a non-deterministic result is possible, but even if \(i \neq j\), making the final result deterministic, there is an implicit occurrence of non-determinism in the program. It is probably not surprising then that the solutions presented in this paper involve combining *explicit non-determinism* with *incremental updates to arrays*. Indeed, we show later that a suitable functional (albeit non-deterministic) model of the above parallel imperative program is:

\[
\text{with a parapply} \\
\text{parfuns} \\
\quad a[i] \leftarrow e_1; \\
\quad a[j] \leftarrow e_2 \\
\text{parend}
\]

which looks surprisingly (and perhaps uncomfortably) similar to the imperative program. Its correctness depends on the semantics of the special constructs “with-parapply,” “parfuns- parend,” and “\(\leftarrow\),” all of which will be described in detail later. Studying these kinds of solutions raises some rather interesting questions about the relationship between parallelism and non-determinism – indeed, we may generate more questions than answers – and tends to blur the traditionally clear-cut distinction between functional and imperative languages.

We proceed by first summarizing in the next section what it means for functional programs to have “implicit parallelism,” which we argue is a special case of a property called *declarative abstraction*. Incremental and monolithic approaches to arrays are then described.
in Section 3, including a summary of research on copy-avoidance strategies to allow incremental updates to be done efficiently, and making an argument for why the incremental approach is most general. Next, "(pseudo-)functional non-determinism" is introduced (Section 4) followed by ways to combine it with incremental arrays to achieve the same behavior as parallel assignment to arrays (Section 5). A possible syntax for common parallel array-assignment patterns is presented in Section 6, which is purposely made to resemble imperative language syntax.

2 Implicit Parallelism and Declarative Abstraction

The literature is replete with discussions of the advantages of declarative programming languages, especially in the context of parallel computing. Most of these issues are beyond the scope of this paper, but we do wish to highlight one property that is traditionally described only as a manifestation of the declarative programming style. That property, simply put, is that declarative programming languages are generally characterized by the implicit presence of operational behavior, and the explicit presence of functional (i.e. observable) behavior. Stated another way, declarative languages try to say what is computed but not how. We refer to this property as declarative abstraction, and understanding it is helpful in understanding the approach to parallel array manipulations described later. The way in which imperative languages exhibit declarative abstraction contrasts sharply with its manifestation in declarative languages, and is perhaps best explained by a few examples. (These examples are given in the context of functional languages, but they could easily be couched in, for example, a (pure) logic programming paradigm as well.)

The following are examples of the implicit presence of operational behavior:

1. **Implicit Iteration.** In a functional language iteration (or “looping”) is an operational behavior that is implicitly present in tail recursion, as opposed to explicit looping constructs in an imperative language. Perhaps more so than the functional programming community, the Lisp (in particular Scheme) community highlights this as an indication of the generality of recursion. The Scheme standardization effort, in fact, requires that an implementation be “properly tail-recursive” in order to be faithful to the language semantics [20].

2. **Implicit Memory Management.** Data structures having indefinite extent such as lists are managed implicitly by a dynamic memory manager in functional languages, whereas in many imperative languages the memory must be managed explicitly by the programmer. An extreme example of this is the implicit reuse of “single-threaded arrays,” which we study in detail in Section 3.3.

3. **Implicit Parallelism.** Parallelism is said to be implicit in a functional language, being “dictated” solely by the partial-orders induced by data dependencies. In contrast, parallelism in an imperative language is usually introduced with explicit constructs, such as par-begins/parends, process declarations, message-passing primitives, etc. The implicit approach
is heralded as a virtue by functional programming proponents, since the implicit parallelism is easily detected by a compiler, thus freeing the programmer from operational detail [10, 11].

The implicit parallelism in a functional language is well-understood, having its roots in dataflow research which began almost 20 years ago. To give the unfamiliar reader some idea of this behavior, and thus a better understanding of the ideas to be presented later, consider first the expression $e_1 + e_2$. Most functional languages do not specify an evaluation order for strict binary operators such as $+$, and thus its arguments, in this case $e_1$ and $e_2$, may be evaluated in parallel. In fact, the lack of side-effects endows functional languages with the well-known Church-Rosser property, which guarantees a deterministic result regardless of the execution order chosen (assuming termination).

Coupled with recursion, strict operators allow unlimited degrees of parallelism in a functional language. For example, the program:

```
let f(i) = if i = n then 0 else g(i) + f(i + 1)
in f(0)
```

can be safely unwound into $n$ parallel calls to $g$, as shown in Figure 1, even though the individual arithmetic additions are done sequentially. We will see later that updates to strict arrays induce this same kind of “implicit parallelism.”

The following are examples of the explicit presence of functional behavior:

(4) **Side-effects.** Implicitly present in imperative languages are non-local (and thus referentially opaque) side-effects – for example assignment to a global variable or side-effects via aliasing – as opposed to in a functional language, where such behavior can only be “mimicked” by the explicit passing of the object to be side-effected and explicit return of the “modified” object. The attendant increase in referential transparency is claimed to increase program clarity.

(5) **Non-determinism.** A related example is the explicit use of non-deterministic constructs in a functional language, versus the implicit presence of non-determinism in a (parallel)
imperative language. Non-determinism is an example of functional (i.e. observable) behavior, not merely operational detail, since its presence can directly affect the answer returned by a program. Declarative languages try to make the non-deterministic behavior explicit rather then leaving it as an implicit operational detail. We will return to issues of non-determinism in Section 4.

In all five of the examples listed above, functional languages exhibit declarative abstraction, whereas most imperative languages do not. The obvious advantage of declarative abstraction is that it permits a declarative reading of a program, devoid of operational detail. But this does not come without cost – reasoning about operational behavior (in particular, reasoning about efficiency) may be more difficult, since it requires an understanding of the implicit behavior of a language and how it is manifested in a particular implementation. If such reasoning is desired, then there may be a need to formalize the implicit behavior; an example of formalizing the implicit parallelism in functional languages may be found in [7].

3 Functional Arrays

The list is the most popular data structure in Lisp and functional programming languages, due in a large part to the natural way in which it fits into the recursive programming style, but also, we believe, due to historical biases against other, less general data structures. One of these other data structures, central to many conventional languages, is the array. Although less general than lists, arrays have certain properties that make them superior in many applications:

1. A read or an update to any element can be done in constant time. In other words, one can index into the array directly without any kind of searching.

2. An update costs zero space. In other words, an update does not result in the allocation of any extra space to store the new value.

3. The elements can be stored in contiguous memory locations, thus saving the space required by links (i.e. pointers) in more general data structures.

These properties are what we use to define an acceptable realization of an array data structure. On the other hand, in this paper we concentrate first and foremost on achieving in a functional way the same observable behavior (i.e. the same value) as a particular imperative program. Only then do we consider efficiency.

In the remainder of this section we discuss two kinds of functional arrays, incremental arrays and monolithic arrays. This same terminology was adopted in [24], where the reader may find an excellent summary of a variety of monolithic arrays. For reasons that will become apparent, our interest is primarily in incremental arrays.
3.1 Incremental Arrays

The simplest, and perhaps most naive, version of a functional array is the *incremental array*, having three operators:

1. newarray\((n)\) creates a new array of length \(n\) whose elements are initialized to some default value, say nil.

2. select\((a, i)\) returns the \(i\)th element of the array \(a\).

3. update\((a, i, x)\) returns a new array \(a'\) such that select\((a', i) = x\), and select\((a', j) = select(a, j)\) if \(i \neq j\).

Henceforth we use the notation “\(a[i]\)” as syntactic sugar for “select\((a, i)\),” and “\(a[i] := x\)” as syntax for “update\((a, i, x)\).” Thus the notation “\(a[i] := x\)” will be “overloaded,” in that if it appears in an imperative setting it denotes *assignment*, whereas in a functional setting it denotes an *expression* whose value is an array.\(^2\)

3.2 Monolithic Arrays

Monolithic arrays receive their name from the fact that their elements are defined “all at once” (and presumably in parallel) at the time the array is first created. The canonical behavior of the monolithic approach can be captured by the function mka (for “make-array”), defined as follows: mka\((n, f)\) returns an \(n\)-element array, say \(a\), such that \(a[i] = f(i)\).\(^3\) Versions of mka can be found in several functional languages, including FEL [14], Affl [5], and VAL [16].

In addition, many other variations of mka are possible, such as the function mkga (for “make-general-array”), defined such that mkga\((n, f)\) returns an \(n\)-element array, say \(a\), such that \(a[i] = x\) if there exists \(j\) in the range \(1, ..., n\) where \(f(j) = [i, x]\); otherwise \(a[i] = \text{nil}\).\(^4\) This function can be conveniently used, for example, to solve the “matrix inversion problem” described by Wadler in [24]. Wadler also describes four other kinds of monolithic arrays, the most interesting being variations of mkga which are defined to do useful things when there exists more than one \(j\) such that \(f(j) = [i, x]\). Steele uses a very similar kind of array called a *vector* in Connection Machine Lisp [23].

Another interesting kind of monolithic array is Arvind, Pingali, and Nikhil’s *I-structures* in Id-Nouveau [18]. I-structures are in some sense a blend of incremental and monolithic arrays, in that “assignments” are made to elements of an I-structure in a way not unlike assignment to an imperative array, yet it is an error to assign to the same location more

\(^2\)Although the use of “:=” may seem strange in a functional setting, it is technically no different from the use of any other binary infix operator, such as in “\(a + b\)” (denoting a numeric value) or “\(a = b\)” (denoting a boolean value).

\(^3\)mka is the same as array1 used in [24].

\(^4\)mkga is the same as array2 used in [24].
than once. This preserves the “single-assignment” semantics of a functional language, and is consistent with the nature of monolithic arrays. The primary deficiency of I-structures is their lack of declarative abstraction (as defined in Section 2). In particular, I-structures can be updated in non-local (referentially opaque) ways, including via aliasing. As an aside, the parallelism of I-structures (which Arvind calls “parallel-call-by-value” [1]) can be achieved using a lazy version of mka with an operational semantics that specifies the eager, parallel evaluation of the elements (such an approach is used, for example, in the parallel functional language ParAlfl [6, 12]).

3.3 Implementing Incremental Arrays

At first glance the incremental approach appears to have a very serious problem: it seems to suggest that a copy of the entire array must be made upon each update – this will clearly not satisfy our requirements for an acceptable realization. Fortunately, there exist several copy-avoidance strategies that minimize this problem.

Before proceeding, we first introduce a simple property of arrays that can be exploited in designing a copy avoidance strategy: borrowing terminology introduced in [21], we say that an array is single-threaded if and only if, after every update, the original array is never used again. Note that this definition implies that all mutable arrays are single-threaded; after all, the original array does not even exist after an update, since the update is done destructively. Thus most translations of imperative programs into functional programs result in single-threaded functional arrays, assuming that the translation is done “faithfully.” Indeed, it is easy to argue that most uses of functional arrays, whether or not arrived at by translation, are single-threaded. The significance of this property will become apparent shortly.

Trees

Perhaps the most obvious way to reduce copying is to use some sort of a balanced tree to represent a linear array. Many variations of this idea have appeared in the literature (the most recent being [25]), but they all share a common theme: the array elements are stored in the leaves (thus requiring logarithmic time for access), and updates are done by copying only the part of the tree from the root to the element being modified (thus requiring logarithmic time and space for an update). Although logarithmic complexity is generally considered “good,” it is not as good as constant or zero complexity, and thus clearly does not meet our requirements for an acceptable array realization.

Trailers

There turns out to be a very simple strategy for implementing incremental arrays which goes surprisingly far toward meeting our requirements for an acceptable realization. Consider first
the expression \( \text{update}(a, i, x) \). In computing this value, what if we were to leave the array \( a \) alone, but append onto the front of it a “header” that indicates that the new array has value \( x \) in position \( i \)? This technique is shown graphically in Figure 2a, where the original array has an empty header – note how references to the original array \( a \) are still correct, since \( a \) was left unchanged. In terms of efficiency, it should be clear that updates cost constant time and space, but accesses can take arbitrarily long in time, since the “headers” can become arbitrarily long in length.

Suppose now that instead of constructing a header for the new array, we were to leave a trailer on the old array, as depicted in Figure 2b. Is there any advantage to this approach? The answer is yes, if we assume that the array is single-threaded. For if this is the case, then clearly updates cost constant time and space, and accesses (which will always be to the new array) now cost constant time and zero space. Thus the only requirement for an acceptable realization that we have failed to meet is the one specifying zero space consumption on update. However, even this disadvantage is minimized by noting that if the array is indeed single-threaded, then the new cell that is allocated is immediately “garbage”,\(^5\) since by definition there will be no references to it. Indeed, if a reference-counting implementation were employed as the dynamic memory management scheme, then the memory cell allocation could be avoided altogether – we return to this idea later.

The use of trailers is not a new idea, and in fact was first proposed in a paper on implementing arrays in Prolog [13]. Their use in a functional language was first discussed in

\(^{5}\)This is sometimes called \textit{ephemeral memory consumption}. 

---

Figure 2: Dynamic Copy-Avoidance Strategies
[9], and we have implemented the idea in the functional language Alfl [5].

Compile-time Copy-avoidance

The use of trailers to avoid copying of single-threaded arrays is simple and relatively inexpensive – the only overhead is the (constant) indirection through the header and the (constant) allocation of memory on each update. Yet even this overhead can be avoided, based on the simple observation that if it is known that the array being updated is indeed single-threaded, then the update might as well be done destructively, effectively reusing the old array. Doing so avoids the need for any kind of header or trailer, thus achieving zero-space consumption on update, and achieving the same efficiency as an imperative assignment to the array. If in fact we cannot determine that an array is single-threaded, we can still resort to the trailer implementation technique as back-up.

Determining whether or not an array is single-threaded can be done either at run-time via conventional reference counting, or at compile-time via static analysis. Examples of the former include Goldberg’s implementation of a graph reducer on several commercial multiprocessors [3], and Wise’s proposal for automatic reference counting via hardware support [26]; related to Wise’s work is O’Donnell’s proposal for hardware support of “associative aggregates” [19]. Examples of compile-time analyses include the efforts of Mycroft [17], Schwarz [22], Schmidt [21], Hudak [8], and Hudak and Bloss [9].

As a simple example, consider the canonical problem of initializing every element in some array to 0. In an imperative language we might write:

\[
\text{for } i = 1, \ldots, n \\
\text{do } a[i] := 0
\]

The obvious solution using an incremental array is:

\[
\text{let } \text{init}(lo, hi, acc) = \\
\text{if } lo > hi \text{ then } acc \\
\text{else } \text{init}(lo + 1, hi, a[lo] := 0)
\]

\[
\text{in } \text{init}(1, n, a)
\]

which, using syntactic sugar for tail-recursion, can be rewritten as:

\[
\text{for } i = 1, \ldots, n \\
\text{next } a \text{ is } a[i] := 0 \\
\text{return } a
\]

Although this looks quite similar to the imperative program, it is purely functional, and essentially captures in a functional way a “disciplined” use of imperative assignment in a loop.\(^6\) More importantly, it is quite easy, at least for the analysis described in [8], to

\(^6\)We will return to more of these syntactic games in Section 6.
determine that each of the incremental updates could be done destructively, and thus the “compiled code” for these two programs would essentially be the same.

It is beyond the scope of this paper to discuss these optimization techniques any further, other than to make the following intuitive claim: If a sequential imperative program with mutable arrays is translated “faithfully” into a functional program with incremental updates to immutable arrays, then it is always possible to do the incremental updates destructively. This is because if the translation is indeed faithful, then it is impossible for there to be some other reference to the old array when the update is about to be performed.\(^7\) For the remainder of this paper we assume that the copy-avoidance problem has been completely solved. That is, whenever it is safe to do an update destructively then an implementation will in fact do so. We view the eventual acceptance of this fact by programmers to be philosophically no different from the acceptance of efficient procedure calls, and in particular the acceptance of a compiler’s ability to turn tail-recursions into loops (indeed, note that tail-recursion optimization can be viewed as the reuse of activation records, much like reusing an array during an incremental update!).

### 3.4 The Generality of Incremental Arrays

Monolithic arrays are elegant in their modularity, and exhibit a high degree of parallelism. We have, in fact, experimented with several versions of them in Alfl [5] and ParAlfl [6, 12]. It is perhaps not surprising, however, that incremental arrays are in some sense more general. After all, incremental updates seem to induce the smallest possible change to an array. They can be used to define any of the monolithic arrays described earlier, as well as the ones described in [24]. To highlight this generality, imagine for a moment a situation in which we wish to update once every element in an array of length \(n\). Using the monolithic approach this will consume space proportional to \(n\) (since a new array is created), but the incremental approach (with copy-avoidance optimization) will consume no space. Now suppose we do this operation \(k\) times (fairly common in scientific computing, for example). The monolithic approach will consume \(O(kn)\) space, but the incremental approach + optimization still consumes no space. So the monolithic approach appears to be less efficient in this case than the incremental approach + optimization.

On the other hand, the monolithic approach is attractive since it seems to be better suited to parallelism. In particular, in \(mka(n, f)\) one could evaluate every \(f(i)\) in parallel, and store the results into the new array in parallel as well (meaning in any order). But note that \(mka\) can be defined easily in terms of incremental updates:

\[
mka(n, f) = \begin{array}{l}
\text{let } a = \text{newarray}(n) \\
\text{in for } i = 1, ..., n \\
\text{next } a \text{ is } a[i] := f(i) \\
\text{return } a
\end{array}
\]

\(^7\)This intuitive claim can in fact be formalized in terms of translations between flowchart schema and recursive schema – see [9].
Furthermore, most parallel implementations of functional languages will “unwind” this definition (either statically or dynamically) in exactly the same way that the function \( f \) was unwound in Section 2, thus allowing all of the \( f(i) \) to be evaluated in parallel, just as for the primitive version of \( mka \). On the other hand, the actual array updates cannot be done in parallel, since the recursion specifies a particular order for them, again in analogy to the example in Section 2 where the additions were sequentialized. Although this constraint may not seem too severe, it might be for very large arrays. Fortunately, there is a way around even this problem, to be discussed later.

To conclude this section, we describe a simple solution to the “histogram problem,” which Arvind has posed \cite{1} as a problem not well-suited to I-structures. Our solution demonstrates both of the points made in the previous two paragraphs: incremental updates can solve the problem with minimal space consumption, and with parallelism limited only by the sequential array updates imposed by recursion.

The histogram problem can be described as any situation in which a large amount of data is being accumulated into \( n \) “bins,” where the final heights of the bins represent the histogram (i.e. distribution) of some observed phenomenon. We can represent the bins collectively as an array \( a \) of length \( n \), and we can describe the (imperative) updates that we wish to induce as \( a[f(i)] := g(a[f(i)]) \) for \( i \) from 1 to \( k \), \( k \) being much larger than \( n \), and where the range of \( f \) is \( 1, \ldots, n \). Thus, for example, if we’re just tallying events of some sort, \( g \) might simply add one to its argument. A sequential solution using incremental arrays is very simple:

\[
\begin{align*}
\text{let } a &= \text{newarray}(n) \\
\text{in for } i &= 1, \ldots, k \\
\quad \text{next } a \text{ is } a[f(i)] &= g(a[f(i)]) \\
\text{return } a
\end{align*}
\]

It is easy to determine in this example that \( a \) is single-threaded (assuming \( g \) is single-threaded), and thus this solution consumes \( O(n) \) space. Furthermore, there is still potential for a large amount of parallelism, in that each of the \( f(i) \) could be evaluated concurrently (in fact, let’s assume that the execution of \( f(i) \) corresponds to the “event” being counted). On the other hand, note as in the previous example that the updates are done in the particular order specified by the recursion, thus limiting to some extent the degree of parallelism, or at least the degree of asynchrony. As mentioned earlier, this latter deficiency will be removed in a later section.

\section{An Introduction to “Functional Non-Determinism”}

Most programmers (including the very idealistic among us) admit the need for non-determinism, despite the semantic difficulties that it introduces. It seems to be an essential ingredient

\footnote{A different functional solution to the histogram problem can be found in \cite{24}.}
of “real” systems, such as operating systems and real-time controllers. As discussed ear-
lier, non-determinism in imperative languages is typically manifested by running in parallel
several processes that are side-effecting some global state – the non-determinism is thus
implicit in the semantics of the language. In functional languages non-determinism is mani-
fested through the use of primitive operators such as amb or merge – the non-determinism
is thus made explicit. Quite a few papers have been published on the use of such primi-
tives in functional programming, and it appears quite reasonable to program conventional
non-deterministic applications using them. In this section a few non-deterministic operators
are introduced that are sufficient for our purposes. We refer to them as pseudo-functions,
since we wish to use them in the same way that normal functions are used, but their non-
deterministic behavior makes the word “function” a misnomer.

By way of introduction, McCarthy [15] defined a binary pseudo-function called amb
having the following behavior:

\[
\begin{align*}
amb(e_1, \bot) &= e_1 \\
amb(\bot, e_2) &= e_2 \\
amb(e_1, e_2) &= \text{either } e_1 \text{ or } e_2, \text{ chosen non-deterministically}
\end{align*}
\]

The “operational reading” of \(amb(e_1, e_2)\) is that \(e_1\) and \(e_2\) are evaluated in parallel, and the
one that completes first is returned as the value of the expression. Using \(amb\) one can define
things such as merge that non-deterministically merge two lists, or streams. The pseudo-
function merge is useful, for example, in combining streams of characters from different
computer terminals.

Instead of using amb, however, we define a slightly different primitive called choose_one
that takes a non-empty finite list and returns a pair consisting of a non-deterministically
chosen element of the list, and the list with the one element removed:

\[
\begin{align*}
\text{choose\_one}(\text{[]}) &= [\text{\_} \text{\_}] \\
\text{choose\_one}(\text{[e_1, \ldots, e_n]}) &= [e_i, [e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n]] \\
& \text{where } e_i \neq \bot \text{ is chosen non-deterministically}
\end{align*}
\]

As with amb, choose_one only chooses proper (i.e., non-\(\bot\)) elements of the list to return as
the first element in the pair. In fact, amb can be implemented in terms of choose_one very
easily:

\[
amb(e_1, e_2) = \text{first}(\text{choose\_one}([e_1, e_2]))
\]

However, it is not so easy to implement choose_one in terms of amb, for two reasons: First,
doing so requires nesting of amb, thus raising questions of fairness (which we do not wish
to address in this paper). Second, and perhaps more serious, it is not clear how to specify
the removal of the chosen element from the list, since doing so requires testing the elements,
some of which may be \(\bot\).

(The above is not true. Here’s a simple def’n: choose-one1 [a] = [a,[\_]] choose-one1 a:as
= amb [a,as] (let [a’,as’] = choose-one1 as in [a’,a:as’]) Voila!)
Now let us define a pseudo-function called \texttt{apply\_all} whose behavior is captured by:

\[
\text{apply\_all}(a, [f_1, \ldots, f_n]) = f_{k_1}(f_{k_2}(\ldots(f_{k_n}(a))))
\]

where \(k_1, \ldots, k_n\) is an arbitrary permutation of \(1, \ldots, n\).

Although \texttt{apply\_all} can be defined quite simply in terms of \texttt{choose\_one}:

\[
\begin{align*}
\text{apply\_all}(a, []) &= a \\
\text{apply\_all}(a, lst) &= \text{let } [f, rest] = \text{choose\_one}(lst) \\
&\quad \text{in } \text{apply\_all}(f(a), rest)
\end{align*}
\]

we prefer to think of it as primitive, primarily for reasons of efficiency.

5 Parallel Array Operations

In this section a series of typical parallel array operations are considered, and “pseudo-functional” solutions are given for each.

5.1 Simple Parallel Updates

Recall the canonical problem of updating every element in an array just once. Both an imperative and a functional program for doing this were given in Section 3.3, and we argued that the functional program, with a suitable copy-avoidance optimization, could be made to run as efficiently as the imperative program. If these two programs were run on a parallel system, pretty much the same behavior would result, since the inherent structure of the algorithm is strictly sequential.

But now suppose we wish to run this program in such a way that every update is done in parallel. The use of a monolithic array is one solution, although this creates an entirely new array. On the other hand, in an imperative language we might write something like:

\[
\begin{align*}
\text{forall } i = 1, \ldots, n \\
\text{do } a[i] := 0;
\end{align*}
\]

where we assume that “forall” is a parallel construct that creates, in this case, \(n\) copies of its body and executes them in parallel. This program does not allocate new space, and thus is more efficient than the monolithic approach. How can we achieve such behavior in a functional program?

To answer this question it is helpful to first describe what this program is really “saying” in a \textit{declarative} sense. The answer is that it is defining a “new” array whose elements are created \textit{in any order}; i.e., non-deterministically. This theme often repeats itself: parallel behavior involving side-effects in an imperative language can often only be expressed using
non-determinism in a functional language. A variation of this theme, mentioned in the introduction, is that *implicit* non-determinism in an imperative language must be made *explicit* in a functional language.

With this hint, and the previous discussion about non-determinism, the most naive solution would be to non-deterministically choose one execution order from all possible execution orders in the above program. For example, for $n = 3$, we could write:

\[
\text{first( choose\_one( [ ((a[1] := 0)[2] := 0)[3] := 0,}
\]

\[
((a[1] := 0)[3] := 0)[2] := 0,}
\]

\[
\]

\[
\]

\[
\]

\[
\]

It should be clear that this style of translation results in a program of size $n \times n!$, whereas the original program’s size is independent of $n$. This is clearly unsatisfactory! The observant reader will notice, however, that one could define a recursive function to generate, for any $n$, the above list of expressions given to choose\_one, thus making the program size independent of $n$. Yet this is still unsatisfactory, since, for example, the list is still of length $n \times n!$. A better solution is to use apply\_all:

\[
\text{apply\_all( a, [ } \lambda a.a[1] := 0,}
\]

\[
\lambda a.a[2] := 0,}
\]

\[
\lambda a.a[3] := 0 ] )
\]

It should be clear that this program generates the desired behavior, and although it grows linearly with $n$ (since the list is of length $n$), one could again write a recursive function to generate the list, which in all fairness simply corresponds to the tail-recursive interpretation of “forall” (in other words, if we allow ourselves functional syntax corresponding to “forall,” then the programs would be the same size – such syntax in fact will be introduced in Section 6).

5.2 The Closure of apply\_all

The use of apply\_all results in a more concise translation of a set of parallel imperative statements than choose\_one. Indeed, it would seem that in general we could replace something like:

\[
\text{parbegin}
\]

\[
a[i_1] := e_1;
\]

\[
a[i_2] := e_2;
\]

\[
\ldots
\]

\[
a[i_k] := e_k;
\]

\[
\text{parend}
\]

14
in an imperative language, where \( e_1 \) through \( e_k \) are arbitrary expressions, with:

\[
\text{apply\_all( } a, [ \lambda a.a[i_1] := e_1, \\
\lambda a.a[i_2] := e_2, \\
\vdots \\
\lambda a.a[i_k] := e_k ] )
\]

in a functional language; note that the program size is essentially the same in either case.

Is this in fact a valid translation? The answer depends on \( e_1 \) through \( e_k \): If each of these expressions makes no references to the array \( a \), then yes, it is a valid translation. But what if each of them makes a single reference to some other element in \( a \)? By “parameterizing” this reference we can write such a program imperatively as:

\[
\text{parbegin} \\
\quad a[i_1] := f_1(a[j_1]); \\
\quad a[i_2] := f_2(a[j_2]); \\
\quad \vdots \\
\quad a[i_k] := f_k(a[j_k]); \\
\text{parend}
\]

The conventional imperative interpretation of this program would allow arbitrary interleavings of the array references with the array updates, and thus the non-deterministic possibilities increase considerably. Unfortunately, the functional translation given above would ensure “atomic” behavior of each update, which is not what is desired! (We will return to this atomic kind of behavior shortly.) How can we express this yet-higher degree of non-determinism in a functional language?

Here is a solution. First define \( \text{apply\_all*} \) by:

\[
\begin{align*}
\text{apply\_all*}(a, []) &= a \\
\text{apply\_all*}(a, lst) &= \text{let } [f, rest] = \text{choose\_one}(lst), \\
&\quad \text{val} = f(a) \\
&\quad \text{in if array?}(val) \\
&\quad \quad \text{then apply\_all*}(val, rest) \\
&\quad \quad \text{else apply\_all*}(a, \text{cons}(val, rest))
\end{align*}
\]

\( \text{apply\_all*} \) can be thought of as the “closure” of \( \text{apply\_all} \) (thus the “\*”), since it keeps applying the functions in the list until an array is yielded.

Returning to the example, the correct translation is then:

\[
\text{apply\_all*}( a, [ \lambda a.\lambda a'.a'[i_1] := f_1(a[j_1]), \\
\lambda a.\lambda a'.a'[i_2] := f_2(a[j_2]), \\
\vdots \\
\lambda a.\lambda a'.a'[i_k] := f_k(a[j_k]) ] )
\]
In other words, each of the functions $\lambda a.\lambda a'.a'[i] := f(a[j])$ is “executed” (i.e. applied) twice; once to capture the value $a[j]$, and once to do the update to $a[i]$. After the first execution it is “thrown back into the pool” of things to choose from non-deterministically, and thus by the time the update happens the array may not be the same as it was when $a[j]$ was read. This is just the kind of non-determinism exhibited by the original imperative program.

5.3 Parallel Atomic Updates

Now let us return briefly to the “atomic” behavior that was achieved “accidentally” in the first functional translation. To be fair, we should ask ourselves how one would achieve that kind of behavior in an imperative language (it is easy to conjure up situations where such behavior would be desired – indeed, the histogram problem is one such example). Here is a typical solution using a semaphore:

```plaintext
sem := make_binary_semaphore();
parbegin
    begin P(sem);
        a[i_1] := f_1(a[j_1]);
        V(sem);
    end;
    begin P(sem);
        a[i_2] := f_2(a[j_2]);
        V(sem);
    end;
    ...
    begin P(sem);
        a[i_k] := f_k(a[j_k]);
        V(sem);
    end;
parend
```

whereas in a functional language one would simply write:

```plaintext
apply_all( a, [ \lambda a.a[i_1] := f_1(a[j_1]), 
                 \lambda a.a[i_2] := f_2(a[j_2]), 
                 ... 
                 \lambda a.a[i_k] := f_k(a[j_k]) ] )
```

What can we conclude from all this? Simply put: in imperative languages one must work hard to ensure mutual exclusion, i.e., to reduce the degree of non-determinism; and in a functional language one must work hard to defeat mutual exclusion, i.e., to increase the degree of non-determinism. This is an interesting contrast in programming styles.
5.4 Recursive Nesting of Parallel Array Operations

Consider this definition of a “divide-and-conquer” procedure in an imperative language for updating an entire array (where we assume the array $a$ is “global” to the definition):

\[
\text{init}(lo, hi) = \begin{cases} 
\text{if } lo \geq hi \\
\text{then } a[lo] := 0 \\
\text{else begin } mid := (lo + hi)/2; \\
\text{parbegin } \\
\quad \text{init}(lo, mid); \\
\quad \text{init}(mid + 1, hi) \\
\text{parend } \\
\end{cases}
\]

which would then be used in a context such as: “$a := \ldots; \text{init}(1, n)$.” Here is an equivalent solution in a functional language:

\[
\text{init}(lo, hi) = \begin{cases} 
\text{if } lo \geq hi \\
\text{then } \lambda a.a[lo] := 0 \\
\text{else let } mid = (lo + hi)/2 \\
\text{in } \text{append}( \text{init}(lo, mid), \\
\quad \text{init}(mid + 1, hi) )
\end{cases}
\]

which would then be called via $\text{apply\_all}(a, \text{init}(1, n))$. With such a technique it is possible to define parallel, pseudo-functional, “destructive” programs for problems such as Hoare’s original formulation of Quicksort [4, 8].

5.5 Efficiency

In this section we argue that all of the pseudo-functional examples given earlier can be done as efficiently as their imperative counterparts. Our argument will only be “partial” in the sense that the above claim is only for pseudo-functional programs that are designed to “mimic” a particular parallel imperative program – a more general claim is difficult to make. This is exactly analogous to arguing that tail-recursion can be implemented as efficiently as an imperative loop, in that such a claim cannot be made about recursion in general.

Naturally, we take as given the ability to implement a suitable copy-avoidance strategy such as discussed in Section 3.3. Also, we make the reasonable assumption that a single assignment to an array location is atomic.

**Proposition 1:** The expression:

\[
\text{apply\_all}(a, [ \lambda a.a[i_1] := e_1, \\
\quad \lambda a.a[i_2] := e_2, \\
\quad \ldots \\
\quad \lambda a.a[i_k] := e_k ] )
\]
with the constraint that each \( e_j \) and \( i_j \), \( j = 1, \ldots, k \), makes no reference to \( a \), may be evaluated by executing all of the updates destructively and in parallel.

**Proof:** (Informal) It should be clear that each lambda expression is single-threaded. Further, since each \( e_j \) and \( i_j \) makes no reference to \( a \), then an atomic assignment to an array location is sufficient to ensure the correct non-deterministic behavior.

**Proposition 2:** The expression:

\[
\text{apply}_{all*}( a, \left[ \lambda a.\lambda a'.a'[i_1] := e_1, \right. \\
\left. \lambda a.\lambda a'.a'[i_2] := e_2, \right. \\
\ldots \\
\left. \lambda a.\lambda a'.a'[i_k] := e_k \right] )
\]

with the constraint that each \( i_j \) makes no references to \( a \) and each \( e_j \) makes at most one access to an element of \( a \), may be evaluated by executing all of the updates destructively and in parallel. Furthermore, each array access may be implemented by reading the array element asynchronously at the time it is needed during the evaluation of \( e_i \).

**Proof:** Essentially the same as for Proposition 1.

Note that the “firing” of either an array reference or array update at the time it is needed is analogous to the operational reading of \( \text{amb}(e_1,e_2) \) that says the expression that “completes first” is the one that is chosen.

The observant reader will note that there is a hidden source of inefficiency in the pseudo-functional programs as given: namely, the space required of the lists that hold the lambda expressions in the argument to \( \text{apply}_{all} \) or \( \text{apply}_{all*} \). In general this is a real cost (especially considering the amount of list appending that might occur in a recursive application). However, if the “disciplined” use of the pseudo-functions as captured by the syntax in Section 6 is used, as well as optimizations based on Propositions 1 and 2, then it should be clear that the lists could be eliminated altogether, since they are being replaced by whatever it is that coordinates the parallel execution of the updates (which must exist in any case).

Implementing any of the “atomic” kinds of updates requires either a locking mechanism or, interestingly, just a way to make the lambda expressions compose themselves (pardon the pun) as they are non-deterministically chosen. Thus conventional graph-reduction techniques can in fact be used to ensure the mutual exclusion. This highlights once again a contrast in styles, this time in regard to the implementation, between imperative and declarative programming, and reflects once again the point made earlier: mutual-exclusion is hard to defeat (since it is sort of “built-in”) in functional languages, but is hard to ensure (extra machinery is needed) in imperative languages.
6 It’s All Syntax

Now for the fun of it let’s play the time-honored game of \textit{syntactic sugaring}.\footnote{We have already done this to some extent with syntax for \textit{update} and tail-recursion.} The philosophy behind this game-playing is that if there is a particular pattern of use of non-determinism and arrays that we like, why not provide special syntax for it? For sake of argument let us continue the trend set in the previous sections, and assume that the traditional imperative style of programming with arrays is one such desirable pattern.\footnote{This may be a rather strong assumption, but it makes for good exercise in any case.}

Before proceeding, note that if \texttt{apply\_all(a, lst)} is a proper use of \texttt{apply\_all} (i.e., the elements of \texttt{lst} are all of form $\lambda a.\text{exp}$, where \texttt{exp} evaluates to an array), then \texttt{apply\_all(a, lst)} is equivalent to \texttt{apply\_all\_all*(a, lst)}. Thus we will concentrate only on syntax for \texttt{apply\_all\_all*}. But before doing that, we need syntax for lambda expressions.

Recall that “$a[i] := \text{exp}$” is syntax for “\texttt{update(a, i, exp)}.” Another popular syntax for assignment is “$a[i] \leftarrow \text{exp},$” and thus we use that as syntax for “$\lambda a.\lambda a'.a'[i] := \text{exp}.$” (Note that the update to $a'$ is changed in the syntax to an update to $a$, but that $\text{exp}$ remains the same, and may or may not have references to $a.$) Furthermore, let “$a[i] \leftarrow \text{exp}$” be syntax for “$\lambda a.a[i] := \text{exp}.$” This will be used to achieve the “atomic” behavior discussed in Section 5.3; in this sense the brackets “$\langle ... \rangle$” can be viewed as syntax for a lock, and can be read “do this atomically.”

Next define the following three syntactic sugarcubes:

1. Let “parfuns $s_1; s_2; ...; s_n$ paretnd” be syntax for:

\begin{align*}
\text{append1}(s_1, & \text{append1}(s_2, \ldots \text{append1}(s_{n-1}, s_n))) \\
\text{where } \text{append1}(s_1, s_2) &= \text{append}(	ext{listify}(s_1), \text{listify}(s_2)) \\
\text{listify}(s) &= \text{if list?}(s) \text{ then } s \text{ else } [s]
\end{align*}

2. Let “with a parapply $s$” be syntax for “\texttt{apply\_all\_all*}(a, s).”

3. Just as in Section 3.3 syntax was defined for a for-loop, we now define syntax for a forall-loop. Let:

\begin{align*}
\text{forall } i = 1, ..., n \\
\text{collect } \text{exp}(i)
\end{align*}

be syntax for:

\begin{align*}
\text{let } \text{loop}(lo, hi, acc) &= \text{if } lo > hi \text{ then } acc \\
&\quad \text{else } \text{loop}(lo + 1, hi, \text{cons}(\text{exp}(i), acc)) \\
\text{in } \text{loop}(1, n, [])
\end{align*}
The reader should be cautioned that this syntax is (intentionally) very misleading. In particular, “parfuns ... paren” is really not “executing” anything – it is just packaging together a bunch of lambda expressions into a list.

We are now in a position to try on our new clothes. Consider first the histogram problem from Section 3.4; a sugared, pseudo-functional solution is:

\[
\text{with } a \text{ parapply for all } i = 1, ..., n \\
\text{collect } \{ a[f(i)] \leftarrow g(a[f(i)]) \}
\]

This is an improvement over the functional solution given in Section 3.4, in that the updates can now be done in any order (and therefore presumably in parallel). Note that the updates are still done \emph{atomically}, which is necessary in order to achieve the correct histogram.

Similar solutions exist for all of the examples given in Section 5. To facilitate the comparison of the imperative and sugared pseudo-functional solutions, we present them below in tabular format along with the section numbers for cross-reference purposes. In each case, note carefully the similarities and differences. In particular, note that in the functional solutions the expressions are \emph{lexically closed} over the array name, whereas in the imperative solutions the array name is “global” to the program fragment shown (and thus might conceivably have other parallel operations being performed on it).
<table>
<thead>
<tr>
<th>Section</th>
<th>Imperative Version</th>
<th>Pseudo-Functional Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>forall ( i = 1, \ldots, n ) do ( a[i] := 0; )</td>
<td>with a parapply forall ( i = 1, \ldots, n ) collect ( a[i] \leftarrow 0 )</td>
</tr>
<tr>
<td>5.2</td>
<td>parbegin ( a[i_1] := f_1(a[j_1]); ) ( \ldots ) ( a[i_k] := f_k(a[j_k]) ) parend</td>
<td>with a parapply parfuns ( a[i_1] \leftarrow f_1(a[j_1]); ) ( \ldots ) ( a[i_k] \leftarrow f_k(a[j_k]) ) parend</td>
</tr>
<tr>
<td>5.3</td>
<td>( sem := \text{make_binary_semaphore}(); ) parbegin begin ( P(sem); ) ( a[i_1] := f_1(a[j_1]); ) ( V(sem); ) end; ... begin ( P(sem); ) ( a[i_k] := f_k(a[j_k]); ) ( V(sem); ) end; parend</td>
<td>with a parapply parfuns ( \langle a[i_1] \leftarrow f_1(a[j_1]) \rangle; ) ( \ldots ) ( \langle a[i_k] \leftarrow f_k(a[j_k]) \rangle ) parend</td>
</tr>
<tr>
<td>5.4</td>
<td>( \text{init}(1, n); \ldots )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{init}(lo, hi) = ) if ( lo \geq hi ) then ( a[lo] := 0 ) else begin ( mid := (lo + hi)/2; ) parbegin ( \text{init}(lo, mid); ) ( \text{init}(mid + 1, hi) ) parend end</td>
<td>with a parapply ( \text{init}(1, n) ) where ( \text{init}(lo, hi) = ) if ( lo \geq hi ) then ( a[lo] := 0 ) else let ( mid = (lo + hi)/2 ) in parfuns ( \text{init}(lo, mid); ) ( \text{init}(mid + 1, hi) ) parend</td>
</tr>
</tbody>
</table>
What can be said about all of these syntactic games? At least two things:

- “Imperative folk” might like the result, since it looks more like conventional assignment statements and explicit parallelism in an imperative language.
- “Declarative folk” might like the result, since it imposes constraints on, or encourages a discipline for, the use of assignment-like operations.

How far can one go with these games? Note, for example, that we have given no translation strategy that accounts for an arbitrary number of references to an arbitrary number of different arrays. To do this, and to deal with other insidious behaviors of imperative languages (such as side-effects in the presence of aliasing), one would have to resort to machinery such as used in denotational semantics to capture the notion of store, continuation, etc. We are certain that this can be done, and may be useful in capturing a “pseudo-functional semantics” for a parallel imperative language, but that is not our purpose here – we are just trying to make apparent the range of syntactic possibilities. On the other hand, although we do not claim to know what the right choice of syntax is, we are suggesting that syntax such as presented here may be a better alternative than either the completely unconstrained and undisciplined possibilities in an imperative language, or the somewhat over-constrained and over-disciplined equivalences in a functional language.

7 Conclusions

We have presented an argument for the use of incremental arrays plus non-determinism to achieve the effect of parallel assignment to imperative arrays. Furthermore, we have suggested a possible syntax for expressing such behavior that falls somewhere between the purely functional and (im)purely imperative. The strategy depends critically on the ability of a compiler to recognize that the incremental updates can be done destructively.

Much work remains. In particular, our pseudo-functional syntax is interesting in its similarity to imperative language syntax, but perhaps far too similar to feel comfortable with, while still claiming to be using a functional language. For example, the syntax for lambda expressions completely destroys referential transparency. Of course, non-determinism in general causes this problem, but perhaps there are better alternatives to the syntax given here. Finally, although we have implemented copy avoidance in a real functional language implementation, we have not yet combined it with parallelism and non-determinism in the ways described here. Without doing so, and without gaining experience writing and debugging programs in this way, we can never be sure how good (or bad) the ideas are.
8 Acknowledgements

I wish to thank Los Alamos National Laboratory and MCC for their generous support of the Santa Fe Graph Reduction Workshop, which inspired many of the ideas presented here. A special thanks is extended to Joe Fasel and Bob Keller for their organizational efforts. Also thanks to Adrienne Bloss for commenting on an earlier draft of this manuscript, and to the rest of the Wrestling Team at Yale for the inspiration induced by their critical scrutiny of every new idea.

References


