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

Paul Hudak
Yale University
Department of Computer Science

November 1986

Abstract

Incremental, functional updates to arrays, executed in a non-deterministic manner, are shown to achieve the same effect (in both efficiency and functionality) as parallel assignment to imperative arrays. The strategy depends critically on the ability of a compiler to recognize not only that the incremental updates can be done destructively, but also that the updates may be done in any order. Special functional syntax is introduced that "captures" typical array-assignment patterns. The syntax falls somewhere in 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.

1 Introduction

By now most language theorists/designers/programmers know how to express imperative programs in a functional 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 creation of new variables in let expressions (or subsumed completely within 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 I wish to investigate in depth in this paper. My 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 I will concentrate on parallel implementations.

1This research was supported in part by the US Government under DOE grant FG02-86ER25012.
2Functional programming advocates have argued endlessly why the resulting programs are better than imperative ones, but I do not wish to pursue that issue here.
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, and to do so in a deterministic manner (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 immutable arrays — after all, that is what the semantics of the sequential imperative language dictates. So the question is now this: when one extends the semantics of an imperative language to accommodate parallel operations to arrays, how can one achieve the same behavior in a functional language?

One approach to solving this problem is to invent special functional arrays whose elements are defined all at once (and presumably in parallel) at the time the array is first created — I refer to these as monolithic arrays. Another approach, and the primary one to be investigated in this paper, is to use what I call incremental arrays, in which an “empty” array is incrementally “modified” until the desired result is achieved. Although the incremental approach depends critically on clever optimization strategies to make it efficient, I argue that it is the most general of approaches, and is in fact suitable for modeling both sequential and parallel assignments to mutable arrays.

In the context of incremental arrays, the primary message that I wish to convey is this: 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 array is deterministic. For example, in:

```parbegin
a[i] := e1;
a[j] := e2
parend```

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 occurred already. 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 arrays. Doing so raises some rather interesting questions about the relationship between parallelism and non-determinism; indeed, in this context I may raise more questions than answers.

I will proceed by first summarizing briefly the incremental and monolithic approaches to arrays, including a summary of research on copy-avoidance strategies to allow the incremental updates to be done efficiently, and making an argument for why the incremental approach is most general. Next, “functional non-determinism” is introduced, followed by ways to combine it with incremental arrays to achieve the same behavior as parallel assignment to arrays. Syntax for common parallel array-assignment patterns is then described, which is (uncomfortably) similar to imperative language syntax. Finally, I will discuss strategies for implementing the resulting programs as efficiently as their imperative counterparts.

---

3The terms monolithic and incremental were adopted in [16], where the reader may find an excellent summary of a variety of monolithic arrays.