A New Array Operation

Philip Wadler
Programming Research Group, Oxford University, UK
and Programming Methodology Group, Chalmers University, Göteborg, Sweden

October 1986
(revised, December 1986)*

Providing efficient operations on arrays in a functional language is problematic. No completely satisfactory solution has yet been found. This paper proposes a new solution, which is a variant on the "monolothic" approach to array operations. The new solution is also not completely satisfactory, but does have advantages complementary to other proposals. It makes a class of programs easy to express, notably those involving the construction of histograms. It also allows for parallel implementations without the need to introduce non-determinism.

The work reported here was motivated by discussions at the Workshop on Graph Reduction, of which this is the proceedings. In particular, it was motivated by the contributions of Arvind and Paul Hudak (see this volume), and especially Arvind's observation that the histogram problem is difficult to solve in a satisfactory way.

After the first draft of this paper was written, I discovered that Simon Peyton Jones independently suggested the same idea, also prompted by one of Arvind's talks. After the second draft was written, I discovered that Guy Steele and Daniel Hillis use a very similar notion in Connection Machine Lisp [SH86]. Apparently the simple innovation described in this paper is an idea whose time has come.

This paper is organized as follows. Section 1 briefly surveys previously proposed array operations. Section 2 introduces the new operation. Section 3 gives a small example of its use. Section 4 discusses two variations on the array operation. Section 5 concludes.

1 Background

Most of the proposed methods for incorporating arrays into functional languages fall into one of two groups. These might be called the *incremental* and *monolithic* approaches (the names were suggested by Paul Hudak).

*Published in Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, October 1986, J. Fasel and R. Keller, editors, Springer-Verlag.

In the incremental approach, the basic operation has the form

$$\mathit{update}\ A\ i\ x$$

and returns a new array that is identical to A except that the position with index i has value x. (Paul Hudak and John Hughes, independently, have suggested the alternate notation $A\,[i] := x$ for the term above.) A naive implementation of *update* would copy the entire array, but this is unacceptably expensive. Usually, we expect that each *update* operation is passed the only existing pointer to the array A, so that the array may be updated destructively. David Schmidt calls programs satisfying this condition *single threaded* [Sch85].

Several ways exist to take advantage of single-threading at run-time. If reference counts are used, A may be destructively updated whenever its reference count is one. Sören Holmström has proposed a simple but ingenious run-time technique that does not require reference counts [Hol83]. John Hughes has suggested a variation of Holmström's method, which may be valuable in some cases where single-threading does not apply [Hug85]. Difficulties with adapting Holmström's and Hughes's methods to a lazy language are discussed by William Stoye [Sto85].

Methods of detecting single-threading at compile-time have also been suggested. Schmidt has given syntactic constraints that guarantee it [Sch85]; Hudak has described analysis methods that discover it [HB85, Hud86]; and Holmström has proposed a type-system that enforces it [Hol86]. The compile-time methods proposed to date (so far as I know) all assume a strict language, while the run-time methods work with either a lazy or a strict language.

In the monolithic approach, the basic operation has the form

$$\mathit{array1}\ n\ f$$

and returns a new array A such that $A\,[i] = f\ i$ for each i in the range from 1 to n. This may be generalized slightly, to give the operation

$$\mathit{array2}\ n\ g$$

that returns a new array A such that $A\,[i] = x$ where $(i, x) = g\ j$ for some j in the range 1 to n. It is assumed that each possible value of i will appear in a pair exactly once; that is, if $j_1 \neq j_2$ and $(i_1, x_1) = g\ j_1$ and $(i_2, x_2) = g\ j_2$ then $i_1 \neq i_2$.

For example, the *permutation inversion* problem is as follows. Given an array A that defines a permutation on 1 to n, we wish to find an array B such that $B\,[A\,[i]] = i$ for each i from 1 to n. Setting

$$\begin{array}{lll} B & = & \mathit{array2}\ n\ g \\ & & \quad \textbf{where}\ g\ j\ =\ (A\,[j], j) \end{array}$$

solves the problem.

The incremental and monolithic approaches have complementary advantages. It is easy to see how to define the *array1* and *array2* operations in terms of *update*. Thus, the monolithic approach is less general than the incremental approach. On the other hand, the incremental approach requires that the programmer be careful to maintain single threading and that the implementation be careful to exploit it. The monolithic approach is therefore easier to implement and use, where it is applicable. More importantly, the monolithic approach is also well suited for use in parallel computation, since all n applications of f or g can be computed in parallel. Using parallelism with the incremental approach poses more problems, as analysis is required to discover which updates can be performed in parallel.

In a call of *array2* n g, the only purpose of g is to yield the n pairs $(i_1, x_1), \ldots, (i_n, x_n)$, where $(i_j, x_j) = g\ j$. It would be just as easy to pass an association list of (i, x) pairs. Thus, in place of *array2* we can use the operation

$$\mathit{array3}\ n\ \mathit{ixs}$$

where *ixs* is a list of length n containing pairs of the form (i, x). For example, the permutation inverse problem can be solved by setting

$$B \quad = \quad \mathit{array3}\ n\ [(A\,[j], j) \mid j \leftarrow [1\,..\,n]]$$

(this uses list comprehension notation, as found in KRC [Tur81] and Miranda [Tur85]). Recall that g was restricted so that each index appeared in exactly one pair; we can express an equivalent restriction on *ixs* by insisting that if (i_1, x_1) and (i_2, x_2) are in *ixs* then $i_1 \neq i_2$.

Does this new formulation hamper parallelism? Not necessarily. The function form obviously has sufficient parallelism, since each of $g\ 1, \ldots, g\ n$ can be evaluated in parallel. In the list form, we need to form the list $[g\ 1, \ldots, g\ n]$. If the computation of g is relatively expensive compared to the cost of counting from 1 to n, then this should provide sufficient parallelism. Usually, this will be the case. If it is not, then to achieve sufficient parallelism we would have to use trees in place of lists; this is described in Section 4.

The operations described here have been independently proposed by several people. Bob Keller is one of the people who suggested *array1* [Kel80, Kel82], Paul Hudak is one of the people whose suggested *array2* [Hud84], and David Turner is one of the people who suggested *array3* [Tur86].

Another important approach to array operations uses "I-structures", developed by Arvind and others [ANP86]. The I-structure approach resembles the monolothic approach in that it is more restricted than the incremental approach, but better suited for parallelism.

A completely different approach to the array problem is to forego arrays altogether, on the grounds that tree structures may be more suitable for parallel architectures. This theme has been developed by David Wise [Wis85, Wis86].

The above survey is brief and incomplete. Many proposals for operations on arrays have circulated for a long time without attribution, simply as part of the folklore. I apologize to those whose names and work I have unintentionally omitted.

2 A new array operation

This section presents a new array operation, belonging to the monolithic class. The operation will be derived in several steps by modifying the *array3* operation.

In a lazy functional language, one may choose whether array operations should be strict or lazy. For this paper, we will choose to make the operations strict. The main reason for this choice is that it makes it easier to reason about the time and space complexity of programs that use arrays. Some difficulties in the interactions between arrays and lazy evaluation are discussed in William Stoye's dissertation [Sto85].

We begin by relaxing the assumption in *array3* that each possible index is specified exactly once. That is, we will allow several elements with the same index to appear in the association list that pairs indexes with values. For example, the pairs

$$(i, x_1), \ldots, (i, x_m)$$

which all have the same index i, might appear in the association list *ixs* (although not necessarily adjacent to each other). The question now arises, what value should we give $A\ [i]$? An obvious choice is to use the last value with index i, in this case x_m. A more general choice would be to combine all of the values with index i to get the value of $A\ [i]$. Say the combining function is called h, then we should set $A\ [i] = h\ [x_1, \ldots, x_m]$.

More precisely, we define the operation

$$\mathit{array4}\ h\ n\ \mathit{ixs}$$

to return an array A such that

$$A\ [i] \quad = \quad h\ [x \mid (i, x) \leftarrow \mathit{ixs}]$$

for each i from 1 to n. (Pairs in *ixs* with an index outside the range 1 to n are ignored. Alternatively, once could require that if any such pairs are present then an error value is returned instead of an array.)

A particularly interesting case arises if h is defined using the familiar *fold* function. This function, also sometimes called *accumulate* or *reduce*, is defined as follows:

$$\begin{array}{lcl} \mathit{fold}\ f\ a\ [\,] & = & a \\ \mathit{fold}\ f\ a\ (x : xs) & = & \mathit{fold}\ f\ (f\ a\ x)\ xs \end{array}$$

For example,

$$\mathit{sum} \quad = \quad \mathit{fold}\ (+)\ 0$$

defines the function *sum* that computes the sum of a list.

Because we have chosen to use strict operations on arrays, it is convenient to use a version of *fold* that is strict in its second argument:

$$\begin{array}{lcl} \mathit{strictfold}\ f\ a & = & \mathit{strict}\ g\ a \\ & & \textbf{where} \\ & & \quad \begin{array}{lcl} g\ b\ [\,] & = & b \\ g\ b\ (x : xs) & = & \mathit{strict}\ g\ (f\ b\ x)\ xs \end{array} \end{array}$$

Here *strict* is defined so that *strict* f x first evaluates x and then applies f. It is easy to prove that if f is strict in its first argument (that is, $f \perp x = \perp$), then $f = \mathit{strict}\ f$ and $\mathit{fold}\ f\ a = \mathit{strictfold}\ f\ a$. Note that *strictfold* can be computed quite efficiently, since $f\ b\ x$ can be computed directly without forming a closure.

So we define

$$\mathit{array5}\ f\ a\ n\ \mathit{ixs}$$

to be equivalent to

$$\mathit{array4}\ (\mathit{strictfold}\ f\ a)\ n\ \mathit{ixs}$$

This function can be computed in an efficient manner, as follows. First, an array A of size n is allocated, and each element of A is initialized to the value a. Then, each element of the list *ixs* is processed in turn. If the element is (i, x), then if $A\ [i]$ currently contains the value b it is replaced by $f\ b\ x$. (In imperative terms: $A\ [i] := f\ (A\ [i])\ x$.)

Note that the operation of setting $A\ [i]$ to the last value in *ixs* with index i can be derived as a special case. We simply compute

$$\begin{array}{l} \mathit{array5}\ \mathit{right}\ a\ n\ \mathit{ixs} \\ \textbf{where}\ \ \mathit{right}\ x\ y\ \ =\ \ y \end{array}$$

Elements of the array that are unspecified by *ixs* will be set to the initial value a.

However, it is the general case that is of more interest. For example,

$$\mathit{array5}\ (+)\ 0\ n\ \mathit{ixs}$$

is useful for accumulating the list of pairs *ixs* into a histogram. Finding histograms of this sort is quite an important operation. However, neither the monolithic operations suggested by Hudak nor the I-structures suggested by Arvind can compute such histograms efficiently. Indeed, as mentioned previously, this paper was motivated by Arvind's observation that computing histograms is an important problem not addressed by I-structures.

Further, there is even more scope for parallelism when f is a an associative and commutative operator, as in the case of $(+)$. It is easy to see that for such operators, the order in which the operations occur is irrelevant. Hence, in such situations all elements of the list *ixs* may be evaluated in parallel. We therefore define our final operation

$$\mathit{array6}\ f\ a\ n\ \mathit{ixs}$$

which is identical to *array5* except that f is assumed to be associative and commutative. This new operation yields wide scope for parallelism. Indeed, this method yields the maximum amount of parallelism possible for this sort of computation. (However, computation of the list *ixs* may introduce unnecessary sequencing; ways to avoid this are discussed in Section 4.)

3 The plotting problem

As a more detailed example, somewhat similar to histograms, consider the *plotting problem*. We are given a point (x_{max}, y_{max}) and a list of points *xys*, such that every point

(x, y) in xys has $0 \leq x < x_{max}$ and $0 \leq y < y_{max}$. We wish to return a two-dimensional array A, such that $A\ [x, y]$ contains *True* if (x, y) is in xys and *False* otherwise.

This requires that we choose a representation for two-dimensional arrays. We will pick the traditional row-major representation. The two-dimensional array A with dimensions x_{max} by y_{max} will be represented by a one-dimensional array B with dimension $x_{max} \times y_{max}$, and the value of $A\ [x, y]$ will be found in $B\ [(x_{max} \times y) + x + 1]$.

We first define an analogue of *array6* for two-dimensional arrays:

$$\begin{aligned} & \mathit{array6in2d}\ f\ a\ (x_{max}, y_{max})\ xyzs \\ & \quad = \quad \mathit{array6}\ f\ a\ (x_{max} \times y_{max})\ [((x_{max} \times y) + x + 1, z) \mid ((x, y), z) \leftarrow xyzs] \end{aligned}$$

The solution is then given by

$$\begin{aligned} & \mathit{plot}\ (x_{max}, y_{max})\ xys \\ & \quad = \quad \mathit{array6in2d}\ \mathit{or}\ \mathit{False}\ (x_{max}, y_{max})\ [((x, y), \mathit{True}) \mid (x, y) \leftarrow xys] \end{aligned}$$

Note that *or* is associative and commutative, and thus satisfies the requirements of an argument to *array6*. Also note that passing *False* as the second argument neatly achieves the desired effect of setting an array element to *False* if it is not specified by the list xys.

Another way to represent a two-dimensional array is by an array of arrays. The bad news is that such a representation does not work well with the array operations suggested here. The good news is that this is not a severe problem, as the row-major representation is quite adequate.

4 Alternative representations

In the above, finite collections of (i, x) pairs are represented by lists. Lists have the advantage that they are conceptually simple, and that most functional languages contain features that make them easy to manipulate. Further, if the list we are forming is, say,

$$[g\ j \mid j \leftarrow [1\,..\,n]]$$

and the function g is relatively expensive to compute, then little potential parallelism is lost by representing collections of pairs in this way. On the other hand, if g is relatively inexpensive, then it would be better to use a different representation for the collection.

One alternative representation would be to represent the list above by g and n, thus returning to the functional representation used by *array1* and *array2*. Since g can be applied in parallel to all the values from 1 to n, this offers maximum parallelism. Although adequate for most purposes, this representation may be cumbersome for some applications.

Another possibility is to use a representation based on trees. Several researchers have suggested that lists be represented using a tree data type [Kel80, SH82, Mee84]. This type could be declared

$$\mathit{tree}\ \alpha \quad ::= \quad \mathit{Empty} \mid \mathit{Unit}\ \alpha \mid \mathit{Join}\ (\mathit{tree}\ \alpha)\ (\mathit{tree}\ \alpha)$$

(this uses the notation of Miranda). Here *Empty* denotes the empty list, *Unit* x denotes the list containing the single element x, and *Join* xs ys denotes the result of appending the lists xs and ys. Clearly, such trees can be generated in a divide-and-conquer manner, and thus provide sufficient parallelism.

The *fold* operation is defined over such trees as follows:

$$\begin{array}{lcl} \mathit{fold}\ f\ a\ \mathit{Empty} & = & a \\ \mathit{fold}\ f\ a\ (\mathit{Unit}\ x) & = & x \\ \mathit{fold}\ f\ a\ (\mathit{Join}\ xs\ ys) & = & f\ (\mathit{fold}\ f\ a\ xs)\ (\mathit{fold}\ f\ a\ ys) \end{array}$$

Meertens points out that this definition is only sensible if f is an associative operation with unit a, and that in this case *fold* corresponds to a homomorphism from lists onto the range of the fold. Meertens also notes that if in addition f is commutative, then this is equivalent to making the *Join* operation commutative, so that the trees represent bags rather than lists. Note that this fits neatly with the use of *fold* in *array5* and *array6*. For more on folding and homomorphisms, see Meertens [Mee84] or Bird [Bir86].

5 Conclusions

A new monolithic array operation has been proposed. The operation is similar to earlier proposals, except that it uses the *fold* operation to combine multiple values with the same index. The new operation conveniently handles certain important kinds of computation, such as histogram, and provides parallelism in a convenient way.

This method solves the important problem of calculating histograms in a fashion suitable for parallel computation. I-structures, devised by Arvind and his colleagues, sacrifice referential transparency in order to increase the opportunities for parallelism, but do not provide a good solution to the histogram problem. I am grateful to Arvind for pointing out this defect of I-structures, which in turn inspired this paper.

Although the method presented here is useful for an important class of problems, it is not the best solution for all problems involving arrays. Further experience is needed to assess its advantages and shortcomings.

Acknowledgements. I am grateful to Joseph Fasel (of Los Alamos National Laboratory). His special attention made it possible for me to attend the Workshop on Graph Reduction, and his hard work was largely responsible for the workshop's success.

I thank Paul Hudak, Bob Keller, Simon Peyton Jones, David Turner, and David Wise for commenting on an earlier draft of this paper and providing background information for Section 1.

This work was performed in part while on a research fellowship sponsored by ICL.

References

[ANP86] Arvind, R. S. Nikhil, and K. Pengali. *Id Nouveau Manual.* Computation Structures Group Memo 265, MIT, 1986.

[Bir86] R. S. Bird. An introduction to the theory of lists. In *Marktoberdorf Workshop on Logics of Programming*, August 1986.

[HB85] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *12th ACM Symposium on Principles of Programming Languages*, pages 300–314, 1985.

[Hol83] Sören Holmström. A simple and efficient way to handle large data structures in applicative languges. In *Proceedings SERC/Chalmers Workshop on Declarative Programming*, University College London, 1983.

[Hol86] Sören Holmström. Types for unshared objects in functional programming. Chalmers University of Technology, Göteborg, Sweden, 1986. To appear.

[Hud84] P. Hudak. *ALFL Reference Manual and Programmer's Guide.* Research Report YALEU/DCS/RR-322, Second Edition, Yale University, October 1984.

[Hud86] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *ACM Symposium on Lisp and Functional Programming*, pages 351–363, August 1986.

[Hug85] R. J. M. Hughes. *An efficient implementation of purely functional arrays.* Technical Report, Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden, 1985.

[Kel80] R. Keller. Divide and CONCer. In *ACM Symposium on Lisp*, pages 196–202, Palo Alto, California, August 1980.

[Kel82] R.M. Keller. *FEL programmer's guide.* Technical Report AMPS TR 7, University of Utah, March 1982.

[Mee84] L. Meertens. Algorithmics: towards programming as a mathematical activity. In J. W. de Bakker, *et. al.*, editor, *Mathematics and Computer Science*, North-Holland, 1984.

[Sch85] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7:299–310, 1985.

[SH82] M. R. Sleep and S. Holmström. A short note concerning lazy reduction rules of append. *Software Practice and Experience*, 12(11):1082–4, November 1982.

[SH86] G. L. Steele Jr. and W. D. Hillis. Connection Machine Lisp: fine-grained parallel symbolic processing. In *ACM Symposium on Lisp and Functional Programming*, pages 279–297, Cambridge, Massachusetts, August 1986.

[Sto85] W. Stoye. *The Implementation of Functional Languages using Custom Hardware*. PhD thesis, Cambridge University, December 1985. Computer Laboratory Technical Report No. 81.

[Tur81] D. A. Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and D. Turner, editors, *Functional Programming and Its Applications*, Cambridge University Press, 1981.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Springer-Verlag, September 1985.

[Tur86] D. A. Turner. *Applicative arrays—an alternative proposal*. Technical Memo, Burroughs Austin Research Center, April 1986.

[Wis85] D. S. Wise. Representing matrices as quadtrees for parallel processors. *Information Processing Letters*, 20:195–199, May 1985.

[Wis86] D. S. Wise. Parallel decomposition of matrix inversion using quadtrees. In *International Conference on Parallel Processing*, pages 92–99, 1986. (IEEE Cat. No. 86CH2355-6).