ACM Sigplan Notices 26,8 (Aug. 1991),145-147.

Shallow Binding Makes Functional Arrays Fast

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 FAX

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

Access and update for random elements of arrays in imperative programming languages are O(1) operations. Implementing functional programming languages to achieve equivalent efficiency has proved difficult. We show how the straight-forward application of *shallow binding* to functional arrays automatically achieves O(1) update for single-threaded usage.

INTRODUCTION

An old saw among mathematicians states that "a method is a device which you use twice" [Polya57,p.208]. Since computer science is such a young field, we need all the *methods* we can find. In this note, we show how the device of "trailers" [Eriksson84], used in the implementation of functional arrays, is really the device of "shallow binding" [Baker78b], used in the implementation of programming languages, in disguise. Another use of the same "shallow binding" device is for implementing "state spaces" [Hanson84] [Haynes87], which provide "unwind-protect" sorts of resource control [Steele90]. The "shallow binding" device can also be seen in database recovery schemes such as *logging* and *shadowing* [Lampson76], although database folks have yet to discover non-linear (tree) environments. Finally, the device has been upgraded to a "theory" of optimistic concurrency control and the management of state in a parallel system [Baker90b]. As a result of these uses, "shallow binding" resides firmly in the computer science "method" repertoire.

Functional programming languages are so-called because their lack of "side effects" allows their operations to be modelled by true mathematical functions, rather than functions depending upon a hidden "state". True mathematical functions have the property of "referential transparency", in which the result of a function call depends only upon the arguments of the function, and nothing else. Imperative assignment is incompatible with functionality because it can change the "state" of a computation so that *referential transparency* is violated. Imperative assignment is therefore banned from functional languages.

Most assignments to simple variables can be efficiently simulated in a functional language using such mechanisms as "tail recursion",[1] which allows imperative loops to be efficiently simulated by recursion. However, incremental modifications to arrays using array element assignment are not easy to efficiently simulate in a functional language. For example, the simplistic implementation of incremental array modifications involves copying the entire array; such an implementation can result in quadratic behavior for a trivial loop which initializes an array to zero [Bloss89].

Many mechanisms have been proposed to ameliorate this "aggregate update" problem so that functional languages can achieve the efficiency of imperative languages. Run-time mechanisms involving sophisticated data structures [Arvind87] [O'Donnell85] can lower the effective cost of updating, while compile-time mechanisms [Baker90a] [Barth77] [Bloss89] [Gopinath89] [Hederman88] [Hudak85,86] [Schwartz75a,b] can discover situations where the array to be updated will be immediately thrown away, so it can instead be recycled (after modification) as the updated array.

Many functional programming advocates would be happy to achieve efficiency in the most common case, where a "single-threaded" imperative computation is being simulated in a functional programming style. Most run-time solutions to the aggregate update problem are too complex

[1]The implementation of tail recursion [Hanson90] is itself an "aggregate update problem", but for stack frames, not individual program variables.

because they are not optimized for this "single thread" case, while compile-time solutions find it quite difficult to detect "single-threadedness".

We show that a straight-forward application of *shallow binding* elegantly provides O(1) complexity for aggregate reads and updates in the case of single-threadedness, and continues to operate correctly in the case of multiple threads, although O(1) update complexity is lost in this case.

SHALLOW BINDING

Shallow binding [Baker78b] was originally invented for the efficient implementation of the dynamic binding of Lisp variables. The values associated with variables are stored in an *environment* structure. Since Lisp allows for first-class functional values, the environment structures must in general be *tree-shaped*. Environment structures have three operations: *access* a variable, *extend* the environment with a new variable, and *change context* to a different environment structure. Three different implementations exist which provide O(1) complexity for any two of the three operations, while the third is O(n), where n is the size of the environment structure.

A *shallow* binding environment structure allows for O(1) access to program variables and O(1) extensions with new variables in exchange for additional work during context changes—e.g., function calls and returns. A *deep* binding environment structure allows for O(1) extensions and context changes, but access and update of program variables is O(n). An *acquaintance vector* environment structure allows for O(1) access to program variables and context changes, but extension becomes O(n).

Deep binding is conceptually the simplest environment mechanism, because the values of the variables are kept in an association list, which is searched (in O(n)) to provide access to the variable. The extension of an environment requires only the creation of a new extension node which specifies the new variable and the old environment, and hence is O(1). Changing environments can be done in O(1) by simply changing a context pointer.

Shallow binding keeps the value of each variable for the current context in its *value cell* which is O(1) accessible. Extension takes an environment and a variable name and creates a new environment by constructing (in O(1)) an object which indicates that the extension environment differs from the given environment by exactly one variable—the variable given in the extension process. Changing the context to a new environment involves tracing a chain of incremental extensions from the new environment back to the current environment; during this tracing process, the contents of the value cells are swapped with the contents of the incremental extensions, so that the value cells become current for the new context and the extensions now detail the differences between the old context and the new one [Baker78b].

Acquaintance vectors encode a complete environment context in each vector. Therefore, the context can be changed in O(1) by simply changing a context pointer. By assigning each variable a unique (constant) integer index, the value of the variable in the context can be accessed in O(1). However, the extension of an environment requires the copying of an entire acquaintance vector, and hence is O(n).

SHALLOW FUNCTIONAL ARRAYS

The semantics of an array are simply the association of the pair <array-object,indices> with a value. We identify the notion of "array" with "environment" and "indices" with "variable". Functional arrays are not updated in place, but a new version is created; hence "update" is environment "extension". Having made this identification, we can examine the three types of environment structures and their suitability for the implementation of functional arrays.

The simplistic (copying) implementation of functional arrays is equivalent to the "acquaintance vector" implementation of environment structures; access is O(1), changing arrays is O(1), but extension (update) is O(n). The use of "shadowing associations" is equivalent to the "deep binding" implementation of environment structures. Finally, the use of "trailers" [Bloss89] is equivalent to "shallow binding".

In a shallow-bound functional array implementation, the array itself can be seen as a *cache* for the values in the current context. If the program is single-threaded, then the cache is always current.

A single-threaded functional program operating on a shallow-bound array will execute within a constant factor of the speed of the equivalent imperative program. This is because every array update will construct an environment difference node ("trailer node" [Bloss89]) of constant size, which will then be immediately let go of, due to single-threadedness. Since array accesses will always be in the current "context" due to single-threadedness, array accesses will be O(1). Finally, "context changes" are O(1), as they are in the imperative program, so there is no difference in speed for this operation.

The major difference in speed between the shallow functional program and the imperative program will be the garbage collection overhead for the trailer nodes that are created for every array update. While reference counting can eliminate the necessity for this garbage collection by allowing the discovery of single-threadedness [Hudak85,86], the overhead for reference counting can exceed that for garbage collecting the trailer nodes [Baker78a]. Furthermore, reference counting does nothing to reduce the copying effort for non-single-threaded programs.

CONCLUSIONS

The three implementation mechanisms for functional arrays which are equivalent to deep binding, shallow binding, and acquaintance vectors, have all been described in the functional programming literature along with their complexity. However, this paper is the first to make the connection with the literature on variable-binding environments.

REFERENCES

Arvind, Nikhil, R.S., and Keshav, K.P. "I-structures: data structures for parallel computing". *Proc. Workshop on Graph Reduction*, Los Alamos, NM, Feb. 1987.

Baker, Henry. "List processing in real time on a serial computer". *CACM 21*,4 (April 1978),280-294.

Baker, Henry. "Shallow Binding in Lisp 1.5". *CACM 21*,7 (July 1978),565-569.

Baker, Henry. "Unify and Conquer (Garbage, Updating, Aliasing ...) in Functional Languages". *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, Nice, France, June, 1990,218-226.

Baker, Henry. "Worlds in Collision: A Mostly Functional Model of Concurrency Control and Recovery". Tech. Memo, Nimble Computer Corp., 1990,14p.

Barth, J.M. "Shifting garbage collection overhead to compile time". *CACM 20*,7 (July 1977),513-518.

Bloss, A. "Update Analysis and the Efficient Implementation of Functional Aggregates". *Proc. 4'th ACM/IFIP Conf. Funct. Progr. & Comp. Arch.*, London, Sept. 1989,26-38.

Eriksson, Lars-Henrik and Rayner, Manny. "Incorporating Mutable Arrays into Logic Programming". *Proc. 1984 Logic Programming Conference*,101-114.

Gharachorloo, K, Sarkar, V., and Hennessy J.L. "A Simple and Efficient Implementation Approach for Single Assignment Languages". *Proc 1988 ACM Conf. on Lisp and Funct. Programming*, Snowbird, UT, July 1988,259-268

Gopinath, K., and Hennessy, John L. "Copy Elimination in Functional Languages". *Proc. 16'th ACM POPL*, Jan. 1989,303-314.

Hanson, Christopher, and Lamping, John. "Dynamic Binding in Scheme". Unpublished manuscript, 1984.

Hanson, Chris. "Efficient Stack Allocation for Tail-Recursive Languages". *Proc. 1990 ACM Conf. on Lisp and Funct. Progr.*, June 1990,106-118.

Haynes, Christopher T., and Friedman, Daniel P. "Embedding Continuations in Procedural Objects". *ACM TOPLAS 9*,4 (Oct. 1987),582-598.

Hederman, Lucy. *Compile Time Garbage Collection*. MS Thesis, Rice Univ. Comp. Sci. Dept., Sept. 1988.

Hudak, P., and Bloss, A. "The aggregate update problem in functional programming systems". *Proc. 12'th ACM POPL*, Jan. 1985.

Hudak, P. "A Semantic Model of Reference Counting and its Abstraction". *Proc. 1986 ACM Lisp and Funct. Progr. Conf.*, Camb. MA,351-363.

Lampson, B.W., and Sturgis, H.E. "Crash Recovery in a Distributed Storage System". Unpublished paper, Xerox PARC, Palo Alto, 1976.

O'Donnell, J.T. "An architecture that efficiently updates associative aggregates in applicative programming language". *Proc. Funct. Progr. Langs. and Computer Arch.*, Springer-Verlag LNCS 201, Sept. 1985,164-189.

Polya, G. *How to Solve It: A New Aspect of Mathematical Method, 2nd Ed.* Princeton U. Press, Princeton, NJ, 1957.

Schwartz, J.T. "Optimization of very high level languages—I. Value transmission and its corollaries". *J. Computer Lang. 1* (1975),161-194.

ACM Sigplan Notices 26,8 (Aug. 1991),145-147.

Schwartz, J.T. "Optimization of very high level languages—II. Deducing relationships of inclusion and membership". *J. Computer Lang. 1*,3 (1975),197-218.

Steele, Guy L. *Common Lisp, the Language; 2nd Ed.* Digital Press, Bedford, MA, 1990,1029p.