OPTIMIZING CLOSURE ENVIRONMENT REPRESENTATIONS

Andrew W. Appel
Trevor T.Y. Jim

CS-TR-168-88

July 1988

Optimizing Closure Environment Representations

*Andrew W. Appel**
Trevor T. Y. Jim†

CS-TR-168-88

July 1988

ABSTRACT

In lexically scoped languages with higher-order functions, any function may have free variables that are bound in the enclosing scope. The function can be compiled into machine code, but the values of the free variables will not be known until run time. Therefore a *closure* is used: a run-time data structure providing access to bindings of variables free in a given program fragment.

Various schemes have been used to represent closures, from linked lists to flat vectors. Different representations will have different performance characteristics, notably in access time, creation time, storage space, and garbage collection. This paper describes some old representations and some new ones, and gives measurements of their efficiency.

* Supported in part by NSF Grant DCR-8603543 and by a Digital Equipment Corp. Faculty Incentive Grant.

† AT&T Bell Laboratories, Murray Hill, NJ 07974

1. Introduction: nested scope and free variables

In many programming languages, functions can be nested inside each other, and a function may access variables declared in functions that enclose it. For example, consider the function `add(x)` that returns as its result another function that adds x to its argument. Such a function can be written in lambda calculus as

$$(\lambda x. (\lambda y. x+y)).$$

Here, the inner function $(\lambda y. x+y)$ uses the variable x from the outer nesting level $(\lambda x. \cdots)$. We say that x is free in $(\lambda y. x+y)$, whereas y is bound in this fragment by the λy.

When a compiler generates a run-time representation of a function, it must arrange that the function can find the appropriate values bound to free variables. For example, the function-value $(\lambda y. x+y)$ must be given access to the correct value of x. Since the function has not been applied yet, y has not been bound to a value and need not appear in the representation of the function.

The usual way of representing functions with free variables is by a *closure:* a data structure containing both the executable code for the function and the values of free variables[1]. A closure is so called because it takes an "open" expression (one with free variables) and "closes" it by providing bindings for the free variables. For example, if the function $(\lambda x. (\lambda y. x+y))$ were applied to the value 3, the result would be the run-time data structure shown in figure 1.

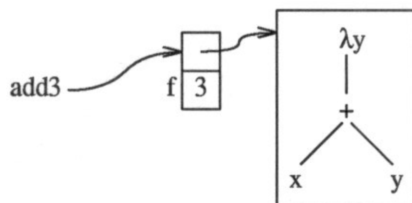

Figure 1

Here, the boxed lambda-expression tree is used to depict a sequence of machine instructions that implements the function $(\lambda y. x+y)$. The *closure* for this function is the pair of cells pointed to by the "add3" arrow. The first component is the machine code, the second component is the value to be used for the free variable x.

When the `add3` function is applied, it must be able to reach both its argument y and the free variable x, so as to add them together. The argument may be found in a conventional place (e.g. in a register, or at a specified stack offset). To ensure that the free variable is accessible, we similarly put a pointer to the closure *add3* in a conventional place when calling the function, and the binding of x may be found relative to that pointer.

In this example, a pair containing the code for $\lambda y. x+y$ and the value for x is an obvious representation for `add3`. In general, however, functions can have many free variables bound at different nesting levels, allowing a number of different representations which satisfy the criterion of providing access to all the free variables.

2. Flat closure data structures

The representation of a function must include a pointer to its executable code, and the values of all free variables of that function. In a functional programming language where variables cannot be altered once they are bound,* it is permissible to have several copies of the same variable or value. In such an environment, one simple closure representation is just a flat record containing all the free variables of a function.

This structure was used by Cardelli[2] in his ML compiler. It is easy to determine at compile time the set

*This includes mostly-functional languages like ML and Scheme (whose compiler can detect which variables cannot be altered).

of free variables of a function; the code to create a function at run time must allocate a record and store the appropriate bindings into it. For example, the closure created when the function $\lambda f.\ (\lambda y.\ (\lambda z. f\ (y+z)))$ is applied to $\lambda x.x$ and the result is applied to 6 is shown in figure 2.

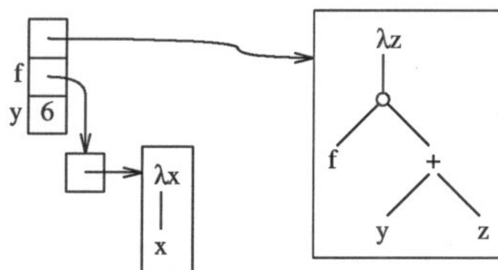

Figure 2

Note that the binding of the variable f is itself a closure, though for a function with no free variables. Therefore, it has a cell for the code pointer, but no cells for free variables.

3. Linked closure data structures

The free variables of a function are either bound in the immediately enclosing function, or they must be accessible from the closure of the enclosing function. Therefore, a closure containing just the bound variables of the enclosing function, together with a pointer to the enclosing function's closure, will suffice to access any free variable. This method is a variant of the ''static link'' or ''access link'' structure, and was used for the first implementation of closures by Landin[1].

The method is illustrated by figure 3, which is the linked-closure representation of the same function shown in figure 2: $\lambda f.\ (\lambda y.\ (\lambda z. f\ (y+z)))(\lambda x.x)6$

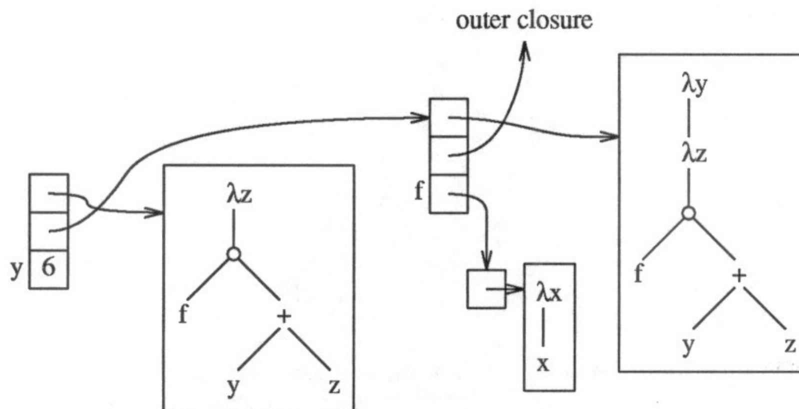

Figure 3

Each closure has a code-pointer, a static link (pointer to outer closure), and a set of free variables.*

4. New strategies for closure representation

Different closure representation strategies (flat vs. linked) may have different costs at run time. Linked closures may require less space than flat closures, since there is less duplication of bound values; but access to variables in flat closures should be faster.

There is no reason to limit ourselves to these two strategies; we could invent new ones. In this section we

*Since the set of free variables is a subset of the local variables of the enclosing procedure, all of this information can be found in the call-frame of that procedure; so this method can be implemented simply by using call-frames as closures, if call-frames are kept on the heap.

describe several strategies, and components of strategies. In the next section we will measure the performance of various combinations of these ideas.

A. Flat

As described in section 2, the closure is a single record containing a copy of every variable free in the associated function.

B. Linked

As described in section 3, the closure of a function ϕ is a record containing a pointer to the closure of the enclosing function ψ, along with the free variables of ϕ bound in ψ.

C. Linked as necessary

Like **B**, except that the link (the pointer to the enclosing closure) is omitted if it is not necessary. This will be the case if ϕ has no free variables, or if all of ϕ's free variables are bound by ψ.

D. Grouping functions

Several functions can share a closure, as long as all of their free variables are available at the time of the closure allocation[3, 4]. This record will contain several machine-code pointers (one for each function), and the union of the free variables of all the functions. Since the implementation requires that the ''first'' element of a function's closure be its code pointer, some functions will be represented by pointers into the middle of the closure record (as illustrated in figure 4).

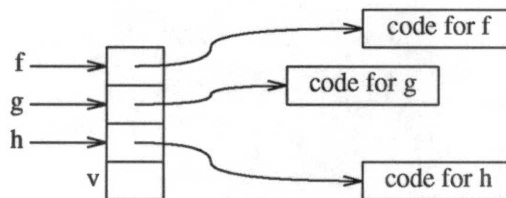

Figure 4. Three grouped functions with one free variable v

E. Only one pointer per closure

Suppose functions f and h share the same closure (as in **D**). If f and h are free in some other closure ϕ, then only a pointer to f need be kept in ϕ, since h can be derived from f by adding an offset. This will incur almost no access penalty, since no extra fetches will be required.

F. Locally minimal size

When a closure is created, there may be several other closures which together contain many of the desired variable bindings. The compiler can take advantage of this, and represent the new closure as a record of pointers to the minimal set of other closures that will cover the set of free variables. A variant of this, in which we minimize the product of closure size times maximum link depth, is provably NP-complete.

G. Heuristic minimization

Suppose a variable f is needed in a closure ϕ, and f happens to be a function (i.e. closure) whose contents are statically known to provide access to a variable v. Then v need not be included separately in ϕ.

H. Path compression

In linked closures, each closure could include a link not only to the immediately enclosing function, but to a closure several levels up. By clever arrangement of these extra links, we could guarantee that no more than $\log n$ fetches are required to access a variable n nesting levels up. This would increase the size of each closure by approximately 1 pointer.

I. Conservative path compression

In any strategy using links, there may be a link to a closure ϕ from which we need only one element x. Instead of including ϕ in the new closure, we could include x directly. This costs no extra space, but now each reference to x will be faster, at the expense of an extra fetch when we construct the closure.

J. Heuristic path compression

Flat closures economize on access time at the expense of closure-building time and space; linked closures economize on closure size at the expense of access time. Our *heuristic path compression* is a compromise between these two approaches. A value is copied into each scope where it is directly used, but other free variables* must be accessed by static links.

5. Performance criteria

We have implemented several closure strategies in our Standard ML compiler[3]. These strategies are combinations of the ideas described in the previous section. We instrumented the compiler to provide run-time profiles of the size of closure records, access path lengths through linked closures to find a variable, the average amount of live (non-garbage) data, and other measures.

Our benchmark program is the compiler itself, run with itself as input. The compiler is a 16,000 line ML program, and applying it to itself executes several hundred million instructions. This is a long enough run to get interesting profile data. We have compiled the same compiler several times, once with each closure strategy. The resulting object modules are then measured, all applied to the same input.

5.1. Execution time

The most straightforward measure of performance is total execution time. Figure 5 shows a graph of CPU time, separated into ordinary execution and garbage collection time. There is a twelve percent difference between the best strategy (heuristic path compression with grouping) and the worst.

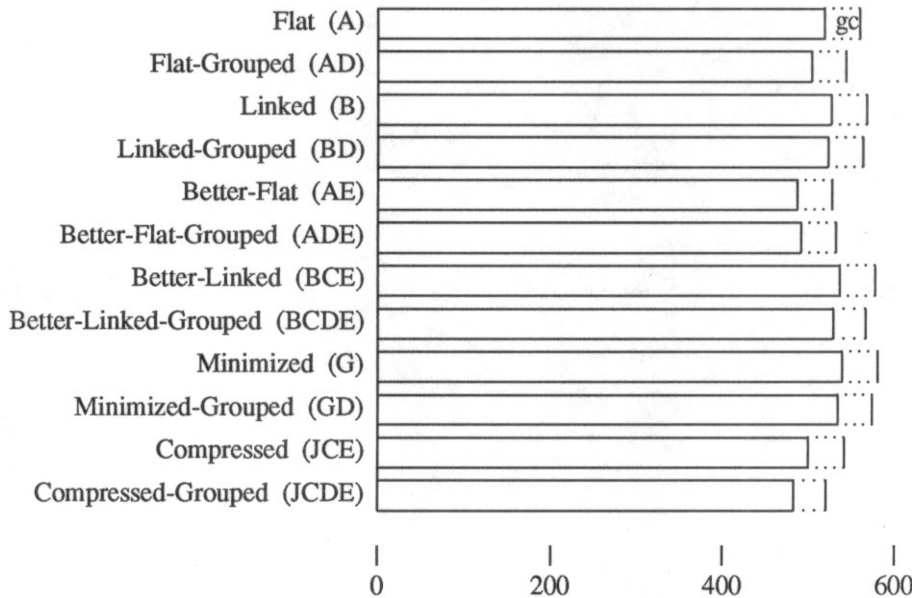

Figure 5. Run time (seconds)

*If a variable is not directly used in a given scope, then it may still be free if internal scopes use the variable.

5.2. Stores and fetches

Total execution time of compiled programs is a measure of efficiency of the entire code generator. In order to isolate the effects of closure strategy, we measured just those memory stores and fetches attributable to closures. In our ML implementation, allocating a record is extremely cheap; creating a record of N words takes exactly $N+1$ *store* instructions, and we can imagine an implementation where only N *stores* are required. We will therefore assume that the number of store instructions related to closure creation is equal to the total size of the closures.

When accessing a variable from flat closures, exactly one fetch instruction is required, since a pointer to the current closure is kept in a register. With linked closures, more fetches are required to traverse the links. The total number of variable-access fetches is a good measure of the cost of accessing variables in a closure strategy. Other fetches, such as those used to find the head and tail (car and cdr) of a list, are not counted in this measure.

If stores and fetches have similar cost, then the execution-time cost of closure strategies should be equal to the sum of closure-related stores and closure-related fetches. This is plotted in figure 6. As mentioned in section 5, we expect flat closures to have more stores (larger closure records) and fewer fetches than linked closures, and this is indeed what happens.

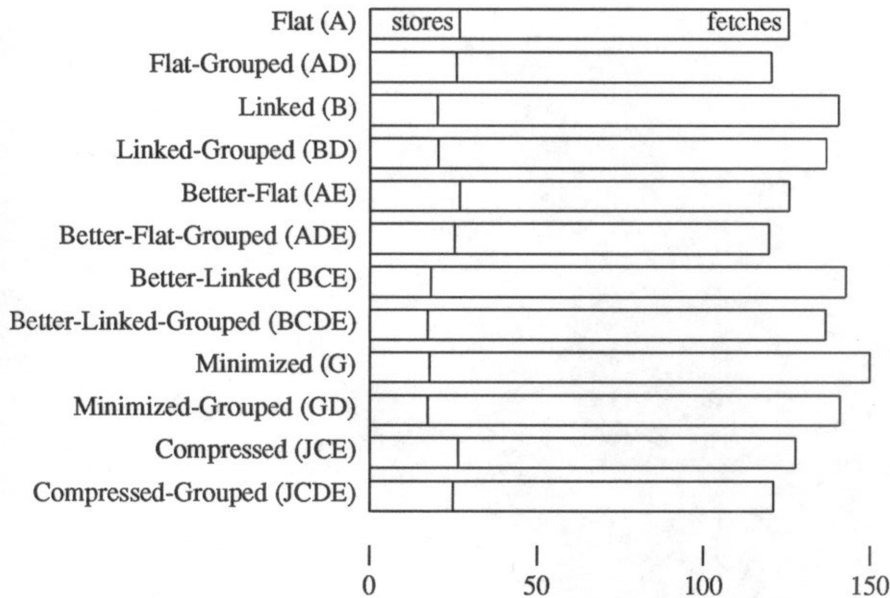

Figure 6. Stores and Fetches (millions)

5.3. Garbage collection

Though it would seem that the stores and fetches described above can be the only effect of closure strategy on performance, this is not the case. With the linked representation, there may be more unreclaimable garbage. That is, an outer closure may point to a variable binding which is no longer needed; but that variable cannot be reclaimed by the garbage collector because various nested closures point to the outer closure. These nested closures may have use for other variables in the outer closure. So the performance relative to garbage collection of linked closures may be worse than for flat closures.

Though we normally use a generational garbage collector (whose performance can be seen in figure 5), we also ran some benchmarks with a simple two-space copying collector to measure the average amount of live data. From this we calculate a measure of "liveness:" the average number of instructions that a record remains live after it is created. Figure 7 shows these results; as predicted, flat closures seem to keep fewer

pointers to unneeded data than linked closures do. This effect is mitigated, however, by the larger size of flat closure records.

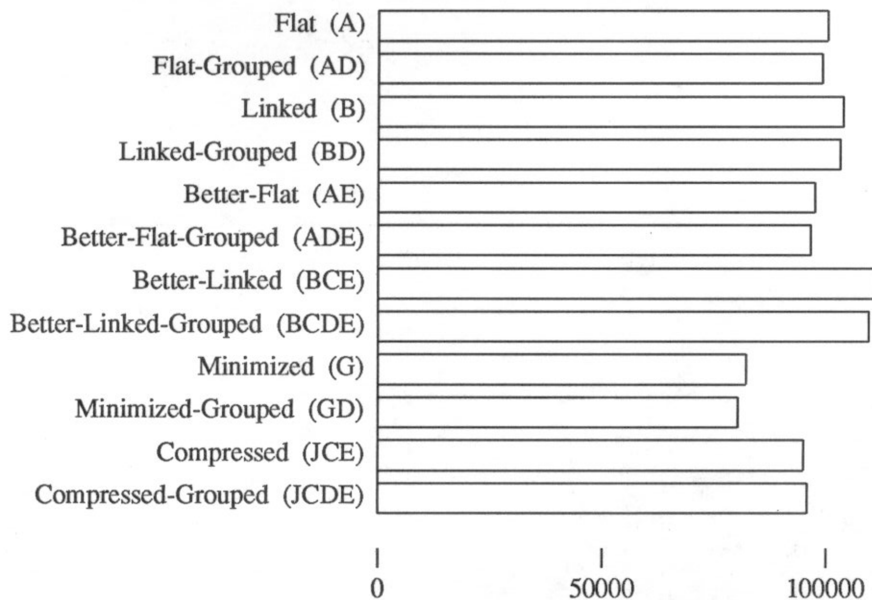

Figure 7. Average liveness (instructions until reclaimable)

6. Conclusion

We have designed and implemented several different strategies for representing closures in functional languages. The choice of closure strategy can make a significant difference in overall performance — on the order of ten percent. Our closure strategies take between 189 and 245 lines of code to implement (including module interface descriptions, etc.); an extra hundred lines of code are a small price to pay for such an improvement.

References

1. P. J. Landin, "The mechanical evaluation of expressions," *Computer J.*, vol. 6, no. 4, pp. 308-320, 1964.

2. Luca Cardelli, "Compiling a functional language," *1984 Symp. on LISP and Functional Programming*, pp. 208-217, ACM, 1984.

3. Andrew W. Appel and David B. MacQueen, "A Standard ML compiler," in *Functional Programming Languages and Computer Architecture (LNCS 274)*, pp. 301-324, Springer-Verlag, 1987.

4. David Kranz, "ORBIT: An Optimizing Compiler for Scheme," PhD Thesis, Yale University, 1987.