Reference Counting of Cyclic Graphs for Functional Programs

T. H. AXFORD
Computer Science, University of Birmingham, Birmingham B15 2TT

A simple method of reference counting applicable to graphs of functional language programs is described. The graph contains strong and weak pointers, but only the strong pointers are counted in the reference counts and by the graph deletion algorithms.

It is shown that graphs of functional programs can be constructed in such a way that the sub-graph got by removing all weak pointers is connected and acyclic. The weak pointers are used only for those recursive references which create cycles in an otherwise acyclic graph. Explicit recursive definitions of functions and data structures may be represented in the graph.

The usual graph reduction rules can be implemented so that they do not destroy the required properties of the graph: the sub-graph of strong pointers always remains connected and acyclic. Thus, simple reference counting can be used safely with cyclic graphs of functional programs.

This method of storage management has the advantage that it incurs little overhead in either storage space or execution time of beta reduction. Nor is there any excessive increase in the complexity of the algorithm needed for graph reduction. It is less suitable, however, for combinator and supercombinator reduction.

Received January 1988, revised May 1988

1. INTRODUCTION

1.1 Reference Counting

When dynamically changing graphical structures occur in a program, one of the most complex aspects of devising a suitable representation for them is in organising the dynamic allocation of memory space. The problem is to identify those parts of the memory which are no longer in use and can be reallocated.

There are two main categories of algorithms for doing this: mark-scan garbage collection and reference counting. In mark-scan algorithms, all accessible structures are fully traversed in the first stage and each memory cell in use is marked; then, in the second stage, the whole available memory is traversed and all unmarked cells are collected and become available for reallocation. Reference counting algorithms, on the other hand, maintain in each memory cell a count of the number of pointers to that cell. These reference counts are updated whenever the structure is changed. If the count in a cell becomes zero, that cell is immediately made available for reallocation.

A large number of algorithms are known in both categories, both for traditional sequential computation and for parallel processing.[3,7,9] While mark-scan techniques are probably the more common, the balance of advantage is moving in favour of reference counting as machines with very large virtual memories become the norm and as parallel processing becomes more widespread.

Firstly, in mark-scan algorithms, the time taken by the marking phase increases with the size of the structures, and the time taken by the scanning phase increases with the size of the available memory. In contrast, reference counting algorithms work only on the part of the structure currently undergoing change and are generally independent of the size of the whole structure and of the size of available memory.

Secondly, mark-scan algorithms do not identify inaccessible parts of the structure for reallocation until the next mark-scan cycle is complete. This cycle can take an appreciable time, and, on sequential machines, it is usual to invoke the garbage collector only when the available space begins to run out. On the other hand, reference counting systems immediately make inaccessible space available for reallocation. On a virtual memory system, this permits newly created parts of a structure to be allocated the same memory locations as recently deleted parts of the structure, thus avoiding excessive paging demands.

Thirdly, mark-scan algorithms require at least two distinct phases, each operating on the whole structure, which must be carried out in sequence. They are, therefore, not so easily adapted for highly parallel operation as reference counting algorithms, which operate on localised parts of the structure only.

1.2 Graphical Representation of Functional Programs

Modern functional programming languages are often implemented by graph reduction, a technique in which the complete program is represented as a graph.[6,8,10,12] In simple terms, a functional program can be thought of as a single mathematical expression. There is no assignment as pure functional languages have no *variables* in the conventional programming language sense, and no concept of *state*. The effect of a loop in a conventional language is achieved through the use of recursive functions.

As a simple example, consider the following functional program for computing y^x, where x is a non-negative integer:

```
power(x,y) =
        if x=0
        then 1
        else
                if even x
                then square(power(x/2,y))
                else y*power(x-1,y)
                fi
        fi
```

The language used for this program is a pseudo-code which should be clear enough to most programmers. We have assumed two predefined functions are available: *even* tests whether its argument is an even number, and *square* simply squares its argument. The symbol **fi** is used simply as a terminator in the

if *a* **then** *b* **else** *c* **fi**

construct, which is considered to be semantically equivalent to a function of three arguments, say *if*(*a*,*b*,*c*), which takes the value *b* if *a* is true, and *c* if *a* is false.

Various slightly different forms of graphical representation have been used in practice, but the differences are not of concern here. Consider, for instance, the graph in Fig. 1, which is a possible representation of the first part of the program for the function *power*.

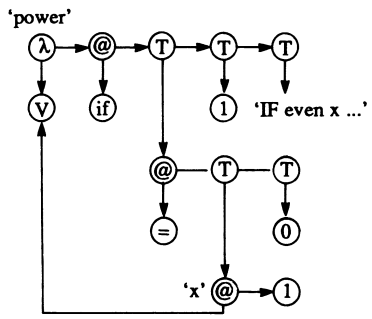

Figure 1. Graph of *power*.

Each circle represents a node of the graph, the symbol inside the circle indicating the type of node. Leaf nodes may be of three types: (i) integers, e.g. 0 and 1; (ii) operators (functions), e.g. = and *if*; (iii) formal arguments, denoted by *V*. Other nodes are of the following types: (i) function definition nodes, denoted by λ, for which the left subgraph (below the node) is the formal argument and the right subgraph (to the right of the node) is the expression which forms the body of the function definition; (ii) function application nodes, denoted @, for which the left subgraph is the function and the right subgraph is the actual argument to which the function is applied; (iii) tuple nodes, denoted *T*, for which the left subgraph is the first element of the tuple and the right subgraph is the rest of the tuple.

In functional languages, following the lambda calculus, it is common to require every function to have exactly one argument. The effect of several arguments can be achieved in a variety of ways, such as currying[6], or, as in the above example, by treating the single argument as a structured object: in the case of the functions *power* and * it is a 2-tuple, while for *if* it is a 3-tuple. The *i*-th element of a tuple can be obtained by 'applying' it to the integer *i*, as if the tuple were a function. So, if $z=(x,y)$, then $x=z(1)$ and $y=z(2)$.

Program and data are treated together in functional languages. To compute 21^5, we would simply evaluate the program

$$answer = power(5,21)$$

where the function *power* has its previous definition. The graph of this program is shown in Fig. 2, in which the subgraph for the function *power* is omitted to save space, but should be filled in exactly as it is in Fig. 1.

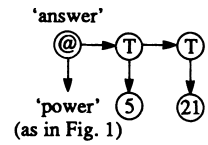

Figure 2. Graph of *power* (5,21).

1.3 Partial evaluation

Interest has been increasing recently in the technique of partial evaluation, although the idea itself is not new.[5,11] As a simple illustration of this technique, consider the following program for the function z^6:

$$sixthpower(z) = power(6,z)$$

using the function *power* defined earlier.

Substituting the definition of *power* gives:

```
sixthpower(z) =
        if 6=0
        then 1
        else
                if even 6
                then square(power(6/2,z))
                else z*power(6–1,z)
                fi
        fi
```

Partial evaluation consists of evaluating those parts of the program which can be evaluated, e.g. 6=0 is false, 6/2 is 3 and 6–1 is 5. By pursuing this process and making further substitutions of the definition of *power*, the final result of this partial evaluation is

$$sixthpower(z) = square(z*square(z))$$

which is a much more concise and efficient program for z^6 than the one we started with!

1.4 A new reference counting technique

Hughes[8] and Brownbridge[2] have described methods of reference counting for cyclic graphs of functional programs. Friedman and Wise[4] have described a method applicable to pure Lisp and similar languages. The methods of Hughes and of Brownbridge are both quite general, but involve a considerable increase in overheads over reference counting for acyclic graphs. The method of Friedman and Wise, on the other hand, involves little added overhead, but is much more limited in its applicability.

The method proposed here is similar to that of Friedman and Wise, but adapted and modified to suit somewhat different and more general circumstances than Friedman and Wise's method. It has been used in the construction of a partial evaluator for a functional programming language.[1]

Partial evaluation, on its own, generally does not succeed in producing the simplest or most efficient program. It is often possible and desirable to carry out further simplification or optimisation by hand. For this reason, it is desirable to retain the original form of the source program as much as possible, so that the partially evaluated program is still recognisable to the programmer.

Hence, the usual technique of combinator reduction[12]

is inappropriate because, once the program has been translated into combinator code, much of the original program structure has been lost. Instead, graph reduction is carried out directly on the graphical representation of the source program, without any other transformations being applied in advance. The reduction method must include beta reduction because the graph contains lambdas, and it must cope with cycles in the graph (which represent recursive function definition). The proposed method of reference counting is applicable and efficient in these circumstances. It is much less efficient for combinator reduction, which does not require beta reduction (because lambdas do not occur in combinator code).

2. THEORETICAL BASIS OF THE METHOD

First, some definitions of terminology used.

Definition: A graph is *strongly connected* if, for any two nodes, a and b, there is a path from a to b and a path from b to a.

Definition: A *strongly connected component* of a graph is a maximal strongly connected subgraph.

Definition: A node, a, of a strongly connected component, C, is said to be an *entry node* of C if there exists a node, b, not in C, such that there is an arc from b to a.

The proposed method of reference counting involves forcing all program graphs to have the following properties.

1. There is exactly one entry node to each strongly connected component. If the root of the graph is in a strongly connected component, then the root is deemed to be the entry node and there must be no other entry nodes in that component.
2. Two types of pointers are distinguished: strong and weak.
3. If all weak pointers are removed, the resultant graph contains no cycles.
4. All nodes are reachable from the root via paths of strong pointers only.

Provided these properties always hold, storage management can be safely based on reference counting strong pointers only. This follows because, by properties (2), (3) and (4), if and only if there are no strong pointers to a node, that node is not part of the graph and can be safely returned to the free node pool.

Reference counts see only strong pointers, and graph deletion algorithms see only strong pointers. For this method to be completely safe, all permitted operations on the graph must also preserve the four properties listed above. For graphs representing functional language programs, these constraints can be satisfied without much inconvenience, as will be argued in detail in the following sections.

Property (1) has not been used so far. It is not required directly by the reference counting method, but is needed later to ensure that the transformations used in graph reduction do not destroy properties (3) and (4).

3. GRAPHS OF FUNCTIONAL PROGRAMS

The precise details of the graphical representation do not concern us, but the following general structural properties of the graph must hold. We take the lambda calculus as the functional language for the purposes of illustration, but allow expressions to be named and, hence, referred to more than once.

So, the program

$$f = E$$

simply means the expression E with the name f. Its graph is a directed graph of the expression E, with the root node named f. If and only if the expression E contains the name x, there will be a path from node f to node x.

The program

$$f = E;$$
$$g = H$$

in which the expression E contains free occurrences of the name g, has a graph in which the subgraph for E contains pointers to the root node of the subgraph for H. For simplicity we will say that the expression E contains free occurrences of a name x if H contains free occurrences of x and E contains free occurrences of g. So, in the graph of any program, there is a path from node x to node y if and only if the expression for x contains free occurrences of y.

Non-recursive programs are represented by acyclic graphs and so no problems arise. All pointers are strong pointers and reference counting is done in the usual way.

3.1 Simple recursive definitions

The program for a recursive definition can be written:

$$f = A$$

where A is any expression which may contain free occurrences of f.

For the graph of this expression (ignoring mutual recursion for the moment) to satisfy the required properties, all pointers from inside the definition to the root (node f) must be weak pointers. If f is not mutually recursive, there are no other names upon which f depends and which, in turn, depend on f (except possibly names local to the expression A which are not accessible elsewhere). This means that f is the only entry node to the strongly connected component through f.

We denote weak pointers textually by putting a prime after the name being referenced; so, the textual representation of the graph for f is:

$$f = [f'/f]A$$

where $[A/b]C$ is the usual lambda calculus notation, meaning rewrite expression C by substituting A (an expression) for b (a name) everywhere that b occurs free in C.

Thus, all occurrences of the name f are replaced by the name f'. Of course, if f is part of a larger program, f may be referenced elsewhere and these occurrences of f will remain. They are not part of any recursion, however (assuming f is not mutually recursive with another function) and so do not lead to cycles in the graph of the program.

3.2 Mutually recursive definitions

Consider the following program for two mutually recursive functions:

$$f = A;$$
$$g = B$$

where A and B are expressions which may contain free occurrences of f and g. We transform this into the following equivalent program which has a graph satisfying the required properties:

$$f = [f'/f][h/g]A;$$
$$h = [f'/f][h'/g]B;$$
$$g = [g'/g]B$$

The subgraph for f no longer contains g (it has been renamed as h) so the strongly connected component through f has only one entry point, f itself. The function g is now represented by a different node and its subgraph may itself contain a strongly connected component (if the expression B contains occurrences of g), but that component will be distinct from the strongly connected component through f (because the expression for f contains no occurrences of g or g', i.e. there is no path from f to g).

This transformation is necessary, not because of the mutual recursion *per se*, but because the two mutually recursive functions, f and g, may both be referenced elsewhere which means that f and g will both be entry nodes to a single strongly connected component. The transformation solves this problem by introducing an auxiliary function, h, which is mutually recursive with f, but not referenced elsewhere (h is simply a copy of g, and f is changed so that all references to g become references to h). So there is now only a single entry node, f, to the strongly connected component which is the mutually recursive definition of f and h. The function g remains unchanged (including its references to f) but is no longer mutually recursive because f contains no references to g.

This transformation is easily generalised to any number of mutually recursive functions, but it is inefficient as the number of auxiliary functions that must be created is $2^n - n - 1$ in the case of n mutually recursive functions all of which depend upon all the others, and all of which may be referenced elsewhere.

In a functional language under development by the author, only the trivial form of mutual recursion is allowed, in which only one of the mutually recursive definitions can be referenced elsewhere and so there is no problem. This very restricted form of mutual recursion is essentially no different from simple recursion. If general mutual recursion is allowed then the transformation described above must be applied to eliminate multiple entry points from strongly connected components.

4. GRAPH REDUCTION

We assume that the only transformation operations applied to the graph after it has been constructed are those of graph reduction. The fundamental reduction operation is beta reduction. In addition, most systems include reduction operations for a number of built-in predefined functions (e.g. arithmetic operators, combinators, etc.).

4.1 Beta reduction

Consider the program:

$$f = \lambda x.A;$$
$$z = fB$$

which we represent as:

$$f = \lambda x.[f'/f]A;$$
$$z = fB$$

(assuming no mutual recursion). Beta reduction of this program transforms it to:

$$f = \lambda x.[f'/f]A;$$
$$z = [B/x]A$$

Now, the reduced expression for z (i.e. $[B/x]A$) is represented graphically by an exact copy of the graph for A, except that all pointers to the bound variable x are replaced by pointers to the root of the graph for B (which itself is unchanged). In the graph for z before reduction there is a path from z to A and a path from z to B. Any cycle in the reduced graph must be (a) completely within A, or (b) completely within B, or (c) a path from node z (the root of the new copy of A) through A, then through B and back to z. Any cycle in categories (a) or (b) clearly existed in the unreduced graph also. Any cycle in category (c) requires that the expression for B contain a free occurrence of z. A corresponding cycle exists in the graph before reduction because z depends on B.

Hence, no new cycles can be created by beta reduction (although an existing cycle can be enlarged by the inclusion of new nodes which did not previously exist). Nor can any new entry points be created to existing cycles. This is easily seen by considering the three cases above. Cycles in categories (a) and (b) are unchanged and their entry points remain unchanged. For a cycle in category (c) an entry point before reduction is z, and that remains an entry point after reduction. The graph for B is unchanged, and the new pointers to B (from within the copy of A) are all on paths from z, so no new entry points are created here. There can be no external pointers into the copy of A, simply because it is a copy and any external pointers into A will still point to the original and not to the copy.

The final property we need to show is that all nodes in the reduced graph remain reachable from the root via a path of strong pointers only. This follows simply from the observations that all nodes in B are reachable from the root of B by paths of strong pointers and all nodes in A are reachable from the root of A by paths of strong pointers. Hence all nodes in $[B/x]A$ are reachable from the root by paths of strong pointers.

4.2 Eta reduction

The program

$$f = \lambda x.Ax$$

can be reduced by eta conversion to

$$f = A$$

provided A contains no free occurrences of x.

The only change to the graph during eta reduction is the removal of those nodes representing the lambda abstraction, the application of A to x and the bound variable x itself. No new cycles are created, nor are any new entry points to existing cycles. The graph for A is unchanged and hence all nodes in A remain reachable from the root of A by paths of strong pointers only.

4.3 Operators

Normal arithmetic operators and others which require their arguments to be evaluated before reduction cause no problems. The graph before reduction is of fixed structure and contains no cycles. After reduction it is replaced by a single node (its value). Cycles elsewhere in the program graph are unaffected.

4.4 Combinators

All the combinators can be defined from first principles in the lambda calculus, so combinator reduction is completely equivalent to one or more applications of beta or eta reduction. Hence, combinator reduction must always preserve the required properties of the graph, and there is no need to consider each combinator individually.

While it is possible to use this method of reference counting with combinator reduction, it incurs a heavy performance penalty. Combinator reduction is attractive because it avoids the copying (of a potentially large subgraph) which is required with ordinary beta reduction when the body of the lambda expression is instantiated. With the proposed method of reference counting, however, combinator reduction of expressions containing a recursive reference to themselves would involve making a copy of the expression to enable the recursive references to be changed to direct references. This additional copying would negate the main advantage of using combinators.

5. FACTORS AFFECTING PERFORMANCE

The reference counting method requires a little more memory space for the graphical representation and a little more processing time than would be required for simple reference counting in a graphical representation without cycles at all. The overhead in both space and time is quite small in most circumstances, however.

To distinguish between strong and weak pointers requires one additional bit per pointer. The number of pointers per node is unchanged. An alternative way of distinguishing strong and weak pointers is to represent a weak pointer by a strong pointer to a special indirection node which contains another pointer to the actual destination object. If the proportion of weak to strong pointers is very small this method can be more efficient.

There will be a corresponding increase in the processing time because of the need to distinguish weak and strong pointers. If the extra bit per pointer method is used, all graph processing algorithms will be slowed down slightly, but if the indirection node method is used there need not be any reduction in processing speed for graphs which do not contain cycles (and hence no weak pointers).

6. COMPARISON WITH OTHER METHODS

This reference counting method is essentially similar to that of Friedman and Wise. Their method is described in terms of a rather specific graphical representation of a LISP-like language, while the method described here is presented in a more general form that is applicable to a wider variety of graphical representations and different types of functional languages, although not appropriate to combinator schemes. It depends primarily on the correspondence between a path from x to y in the graph and the expression for x containing a free occurrence of y (in the terms of the lambda calculus). It should be applicable to any graphical representation of a lambda-calculus-based functional language for which this is the case.

Hughes' method is considerably more complex, requiring two reference counts to be kept in some nodes and an extra pointer field in all nodes. Brownbridge's method uses an extra reference count in every node and also distinguishes strong and weak pointers, but makes no restrictions on the form of the graph that can be handled (so it is applicable to situations other than graph reduction of functional programs). Both these methods incur greater overheads (both in memory space and in execution time) than our method, although they are of more general applicability.

Acknowledgements

I would like to thank Mike Joy for discussions on several points in this paper, and a referee for helpful suggestions on the presentation.

REFERENCES

1. T. H. Axford, *Parital Evaluation in a Functional Language*. Internal report CSR-87-9, Computer Science Dept., University of Birmingham (1987).
2. D. R. Brownbridge, Cyclic Reference Counting for Combinator Machines. *Lecture Notes in Computer Science* (ed. J-P. Jouannaud) **201**, 273–288 (1985).
3. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens, On-the-Fly Garbage Collection: an Exercise in Cooperation. *Communications of the ACM* **21** (11), 966–975 (1978).
4. D. P. Friedman and D. S. Wise, Reference Counting Can Manage the Circular Environments of Mutual Recursion. *Information Processing Letters* **8** (1), 41–45 (1979).
5. Y. Futamura, Partial Evaluation of the Computation Process – An approach to a Compiler-Compiler. *Systems, Computers, Control* **2** (5), 45–50 (1971).
6. H. Glaser, C. Hankin and D. Till, *Principles of Functional Programming*. Prentice-Hall, Englewood Cliffs, New Jersey (1984).
7. D. H. Grit and R. L. Page, Deleting Irrelevant Tasks in an Expression-Oriented Multiprocessor System. *ACM Transactions on Programming Languages and Systems* **3** (1), 49–59 (1981).
8. R. J. M. Hughes, *The Design and Implementation of Programming Languages*. D.Phil. Thesis, Programming Research Group, Oxford University Computing Laboratory (1983).
9. D.E.Knuth, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*. Addison-Wesley, London (1968).
10. S. L. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, New Jersey (1987).
11. P. Sestoft and H. Søndergaard, A Bibliography on Partial Evaluation. *ACM SIGPLAN Notices* **23** (2), 19–27 (1988).
12. D. A. Turner, A New Implementation Technique for Applicative Languages. *Software – Practice and Experience* **9** (1), 31–49 (1979).