This new algorithm deals correctly and automatically with the kind of cyclic (i.e. self-referencing) structures which arise in a combinator graph reduction machine. By extending the standard reference count algorithm, cycles can be handled safely at little extra cost. Cyclic reference counting uses one extra bit per pointer and per object and one extra reference count per object. Extra computation amounts to inspection of the objects in a cyclic structure from time to time before the structure becomes free. In the absence of cycles, no computation overhead is incurred. When executing some typical recursive programs, the number of objects inspected is approximately equal to the number of new objects used during execution.

Introduction

The motivation of this work is the need to store structured objects in a distributed computer system. In such a system, many computers operate concurrently; no one computer can learn the state of the whole system and no one activity can require synchronised action by all the computers[1]. Such systems are seen by many as the best way to exploit advances in computer technology and may in turn provide good support for functional programming languages[2,3,4,5,6,7].

In this context the most commonly used storage management algorithm, (mark-scan garbage collection[8]) is wholly inappropriate. It can require both synchronisation of all the computers in the system and a co-ordinated search of their entire storage. On the other hand, reference counting[8,9] has very attractive properties of localised and incremental processing. The major problem is of course that reference counting is not fully general: it cannot deal with self-referencing cycles of objects.
When the storage manager is to be incorporated as a fundamental feature of the system it must be able to manage all structures correctly. Even in systems where cycles are considered undesirable and only arise in error[4,10], the storage manager has to deal properly with them if storage space is not to be lost forever. Many modified versions of mark-scan have appeared, including 'copying' and 'on-the-fly' garbage collectors [11,12,13,14,15]. These go a long way toward removing the disadvantages of mark-scan whilst retaining efficiency and effectiveness. This paper attempts to remove the disadvantages of reference counting in the belief that the result may be a better compromise than modified mark-scan algorithms[16,17,18,19,20].

Cyclic reference counting was evolved as the 'minimal' addition to standard reference counting which can deal with cycles. Three 'micro-code' pointer operations NEW, COPY and DELETE are used to implement graph manipulating combinators as described by Turner[21]. The nature of combinators ensures that all cycles result from the executions of exactly one instruction (the Y combinator). At the point where Y is executed a single 'cyclic' pointer is created and marked as being 'weak'. Subsequent pointer manipulations proceed as usual taking care to ensure that copies of weak pointers are also marked as being weak. In most cases, pointer deletion can be handled in the standard way. By observing the relative numbers of strong and weak pointers to an object, the deletion of cycles can be handled properly. This is achieved using a new method described below forming the main innovation of this paper.

The appendix describes a form of cyclic reference counting which would be appropriate to support languages such as Pascal or LISP where arbitrary assignment to pointers is permitted. This extension will be the subject of a further paper.

**Implementing Combinators**

A graph reduction machine is an alternative form of computer in which the program is represented by a directed graph. Program execution amounts to replacing (reducing) selected sub-graphs in situ, so that eventually the program is replaced by its result[22,23,24,25].

Combinator graph reduction machines (in short 'combinator machines') are a particular class of reduction machines which use combinators [26,27,21] to perform name-value binding and any associated copying or sharing. Combinators play the same role as 'displays' or the 'static chain' found in many high level language implementations[28]. The advantage is that each combinator is simple to implement, performing some small incremental operation and that functional programs can be compiled into combinator form easily. For our purpose, combinators are also attractive because they provide a complete, well-defined interface between the program and the