One Pass Real-Time Generational Mark-Sweep Garbage Collection

Joe Armstrong and Robert Virding

Computer Science Laboratory
Ellemtel Telecommunications Systems Laboratories
Box 1505
S-125 25 ÄLVSJÖ
SWEDEN
Email: joe@erix.ericsson.se, rv@erix.ericsson.se

Abstract. Traditional mark-sweep garbage collection algorithms do not allow reclamation of data until the mark phase of the algorithm has terminated.

For the class of languages in which destructive operations are not allowed we can arrange that all pointers in the heap always point backwards towards "older" data. In this paper we present a simple scheme for reclaiming data for such language classes with a single pass mark-sweep collector.

We also show how the simple scheme can be modified so that the collection can be done in an incremental manner (making it suitable for real-time collection). Following this we show how the collector can be modified for generational garbage collection, and finally how the scheme can be used for a language with concurrent processes.

1 Introduction

The garbage collector described in this paper is one of the collectors used in the implementation of the programming language Erlang [1]. Erlang is a single-assignment, eager functional language designed for programming real-time concurrent fault-tolerant distributed systems. It has no destructive operations which can create forward pointers.

Erlang is currently being used in commercial products where the applications require relatively large online data-bases. For this type of application it is critical that the garbage collection method used is sufficiently real-time and is able to handle large amounts of data efficiently. Real-time copying garbage collectors such as [3] are inappropriate in our problem domain since they entail the copying of large amounts of static data, and make inefficient use of the available memory.

Note that in this paper we will not describe various garbage collections techniques and relative virtues except where they are directly relevant to our discussion. We refer interested readers to surveys like [12].

2 The Basic Algorithm

This section describes the basic algorithm where all the objects on the heap are of the same type.

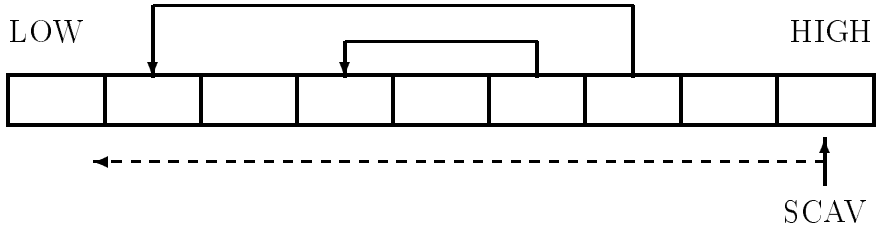

Fig. 1. Heap organisation.

Assume a heap organisation as in Figure 1. New objects are allocated at the "high" end of heap and all pointers in the heap point from high to low addresses.

Assume that each cell in the heap has a mark bit which can be set to MARKED or CLEAR. New cells on the heap have their mark bit set to CLEAR. Assume also that cells in the heap which are pointed to by pointers in the root set have their mark bit set to MARKED.

To garbage collect such a structure we need single pointer SCAV. This scans the heap in the direction of the dotted line in Figure 1. If it encounters a MARKED cell, then the cell is kept. If the marked cell contains a pointer the cell which is pointed to is marked. If it encounters a CLEAR cell then this cell contains garbage.

To illustrate this we start by showing how to garbage collect a conventional list memory consisting of cons cells with `car` and `cdr` fields.

We assume the following functions are available:

`car(i)`	returns the `car` field of cell `i`
`cdr(i)`	returns the `cdr` field of cell `i`
`marked(i)`	returns `true` if cell `i` is marked, otherwise `false`
`type(i)`	returns `atomic` if cell `i` contains an atomic value, otherwise `cons` if cell `i` contains a pointer to a cons cell
`address(i)`	returns the address of a cell, this is only defined when `type(i) != atomic`
`mark(i)`	sets the mark bit in cell `i`
`unmark(i)`	clears the mark bit in cell `i`

In Algorithm 1 we show the simple one pass mark algorithm which marks all accessible cells on the heap. Before executing Algorithm 1 the routine `mark_root` is called, it marks all the cells on the heap which are pointed to directly from

the root set – note that only the 'top-level' `cons` cells pointed to by the root set are marked, `cons` cells which are pointed to by these cells are not marked, these will be dealt with later.

```
SCAV = free - 1;
while (SCAV > HEAP_MIN) {
    if (marked(SCAV)) {
        %% cell SCAV is marked-- so we keep it
        possibly_mark(car(SCAV));
        possibly_mark(cdr(SCAV));
        unmark(SCAV)
    }
    SCAV = SCAV - 2;
}
```

Algorithm 1. Single pass marking.

`possibly_mark(x)` checks field `x` to see if it contains a pointer and if so follows the pointer and marks the indicated cell:

```
possibly_mark(x)
{
   if (type(x) != atomic) mark(address(x))
}
```

Note that this algorithm finds all garbage in the heap in a single pass and that garbage is detected as soon as the scavenger pointer reaches an unmarked cell.

In the algorithm presented so far the age of an object is defined by its address, the lower the address the greater the age. This means that it is impossible to reuse the unmarked cells found in Algorithm 1, the invariant that pointers always point backwards in time, that is towards lower addresses, would be broken.

To be able to reclaim the unmarked cells we need a new method of keeping track of the relative ages of objects, to do so we introduce the idea of a *history list*. The history list is a chain of all the cells connected in the chronological order in which cells were allocated. In what follows we assume that cons cells are represented as in Figure 2.

Fig. 2. Cons cell structure.

The `car` and `cdr` fields contain conventional tagged pointers. The `hist` field stores a pointer to the last previously allocated cons cell.

Two additional pointers `first` and `current` point to the first allocated cell and the last allocated cell respectively.

The chain of pointers in the `hist` cells we call the history list. It represents the historic order in which the list cells were allocated.

```
SCAV = current;
while (SCAV != first) {
    SCAV = hist(SCAV);
}
```

Algorithm 2. Traverse all cells.

The pseudo code in shown Algorithm 2 traverses all cells in the system. Where we assume that the function `hist(i)` returns the address of the last cell allocated before cell `i` was allocated.

We can now modify Algorithm 1 to develop a one pass mark and sweep garbage collection algorithm. We assume, as before, that cons cells in the heap are unmarked prior to calling Algorithm 3.

```
last = current;
SCAV = hist(last);
while (SCAV != first) {
    if (marked(SCAV)) {
        possibly_mark(car(SCAV));
        possibly_mark(cdr(SCAV));
        unmark(SCAV);
        last = SCAV
        SCAV = hist(last);
    } else {
        %% Free cell SCAV, and re-link the
        %% adjacent cells in the history list
        tmp = SCAV;
        SCAV = hist(SCAV);
        set_history(last, SCAV);
        free_cons(tmp);
    };
}
```

Algorithm 3. One pass concurrent mark and sweep.

Algorithm 3 is very similar to Algorithm 1, the differences are that the variable SCAV now traverses the history list and that cells which the collector discovers to be unused are freed. The routine `free_cons(i)` frees the cons cell `i`

for reuse and the routine `set_history(i, j)` is assumed to set the value of the `hist` field of cell `i` to the cell address `j`.

When freeing a cons cell we must also be careful to correctly adjust the history list by bridging over the 'gap' left by the cell which was returned to the free list.

This pointer manipulation can be seen in Figure 3 where we assume that `free_cons(i)` adds cell `i` to a free list of cons cells with head `Free`. The free list is assumed to be linked through the `hist` cell of the cons cells.

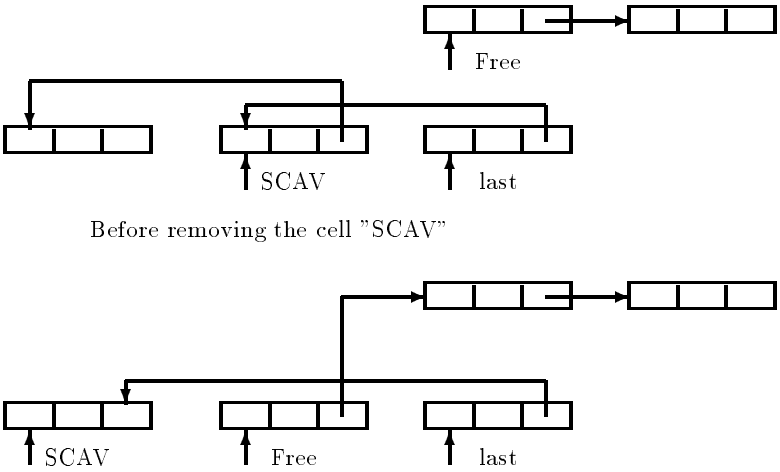

Fig. 3. Freeing a cell.

Algorithm 3 only works if we can ensure that all pointers in an object point to objects which are "older" than the current object.

This algorithm avoids all the problems of marking deep data structures, either by recursive algorithms or more complex but flat pointer reversal techniques [7]. As the ages of objects are determined by their position in the history list, however, it can be very difficult to test the relative age of two objects.

3 Extensions to the Basic Algorithm

We now show some extensions to the basic algorithm which make it practical to use, also show how it can be extended to be sufficiently real-time to be useful when implementing Erlang.

3.1 Multiple Object-types on the Heap

Extending the basic algorithm to handle different types of objects is very simple. For each object in the history list we need to be able to determine how many pointers it contains to other objects in the list. This can be done by either knowing its type or by storing in the object information about where pointers to other objects can be found.

Knowing the type is probably easier to handle and is useful for other things. Whether we obtain the type through type information stored in the object or through typed pointers is irrelevant for the algorithm.

Once the basic history list has been extended to allow objects of different types we can keep all objects in the history list and extend the collection to objects of different types.

Note that we make no assumptions as to how the different types of objects are managed. We are free to use free lists, single or multiple, or any other mechanisms such as BIBOP (Big Bag Of Pages) [8] or Ungar's large object area [10]. We also note that as the freeing of objects is explicit then it is easy to add finalisation of objects where necessary.

3.2 Incremental Collection

Algorithm 3 can easily be made incremental by limiting the number of iterations in the inner `while` loop:

```
start_gc()
{
    mark_root();
    last = current;
    SCAV = list(last);
}

resume_gc()
{
    i = 0;
    while( SCAV != first && i++ < MAX){
        %% same body of the while loop as in Algorithm 3
    }
}
```

Algorithm 4. One pass incremental concurrent mark and sweep.

By setting `MAX` to some suitable limit `resume_gc()` will execute in bounded time. Interleaving `resume_gc()` with the mutator provides a suitably interactive garbage collection that would be sufficient for most soft real-time systems.

For hard real-time systems this simple test may not be precise enough in limiting the time spent in sweeping. In such cases it would be easy to count

the number of objects inspected, objects marked, and objects freed to determine more precisely how long time to sweep. It can also be seen that there is no difficulty in "turning off" the collector for really time critical sections.

3.3 Generational Mark-Sweep

Generational garbage collection is based on the supposition that most objects only live a very short time while a small portion live much longer [6]. By trying to reclaim newly allocated objects more often than old objects it is expected that the collector will work more efficiently. For the history list collector this means that we will sweep the beginning of the list (the new objects) more often than the end of the list (the old objects).

Algorithm 4 can be modified so as to only scavenge the most recent data. All we have to do is abort the scavenge when SCAV reaches some pre-defined limit as is shown in Algorithm 5.

```
start_gc(){
    mark_root();
    last = current;
    SCAV = hist(last);
}

resume_gc(){
    i = 0;
    while( now != LIMIT && i++ < MAX){
        %% same body of the while loop as in Algorithm 3
    }
}
```

Algorithm 5. One pass "generational" and incremental concurrent mark and sweep.

Here we have set the limit at a certain object but it is trivial to modify the algorithm to stop at some other point, for example after a certain number of objects have been swept or a percentage of the total number of objects. In fact the sweeper can be aborted at any time.

Note that when we abort the scavenge loop certain cells below the point where we aborted the search may have been marked. On a later pass of the garbage collector these cells may be retained even through they are no longer reachable from the root set. Such cells can however be collected by allowing the garbage collector to make a complete cycle though memory without prematurely aborting the collection.

We can choose different methods for collecting the older regions. The simplest is to occasionally continue and sweep the whole heap. This can be done in stages - more and more seldom we sweep deeper and deeper into the heap. This

corresponds to traditional generational algorithms where older generations are collected more and more seldom, but always after all younger generations have been collected. We, however, do this interactively.

Another, more sophisticated, method is to run multiple "sweepers" concurrently with one another. We can do this as follows: when the first sweeper reaches its limit we do not stop there but let it continue. At the same time we start a new sweeper at the head of the history list. The two sweepers then sweep concurrently, the one sweeping the older data more slowly, this in keeping with the basic principle that older data lives longer and dies more seldom. When the younger sweeper reaches the limit it stops and is restarted at the head of the list to collect new objects. When the older sweeper reaches the end of the list it is removed and the next time the younger sweeper reaches the limit it is split again.

Note that we can have more than two generations by having more limits and splitting the sweepers when they reach a limit.

3.4 Forward Pointers

So far we have only considered the case where the heap is strictly ordered by age. While this is true for data created by applications in the types of languages we are considering, their *implementation* may create forward pointers by destructive assignments of internal data structures, for example in the G-machine [2, 5]. We now show how it is possible to modify the basic algorithm to handle forward pointers if they occur in a restricted and controlled way.

A simple way to implement forward pointers is to make all destructive assignments pass through one level of indirection. We can then keep all these references in a separate list and use them as part of the root set when marking objects. This ensures that all referenced objects will be marked, even those reached through forward pointers. When the heap is swept we mark all live references and later sweep the reference list freeing all unmarked references.

There are two main problems with this approach:

1. We cannot recover circular data structures. As we use the reference list as part of the root set and mark all referenced objects then a circular reference will force the reference to be marked, even if there are no external references to this data structure.
2. It is very difficult to dereference references, that is bypass the references and make objects point directly to each other. It is very difficult to determine the relative ages of objects so we cannot dereference a reference even to an older object.

An alternative to using indirect references would be to keep a log of all destructive assignments. We would log both the object which contains the assignment and the object being assigned. This method would be more difficult to manage than using references.

4 Discussion

The garbage collection scheme presented in this paper is a specialisation of the standard non-compacting mark-sweep scheme. As such it shares most of the advantages and disadvantages of the standard scheme. It, however, has some additional advantages:

1. Simplicity. The algorithm is much simpler and avoids some of the problems of the traditional mark-sweep, for example the problem of recursive marking.
2. Ease of Extension. It is much easier to extend the algorithm to handle real-time and generational systems.
3. Objects can be reused faster as they are reclaimed in one pass.

The ideas embodied in Algorithm 1 are similar to those using genetic ordering [4, 9]. Our scheme differs in the way we preserve order when collecting objects, genetic ordering algorithms shift the data whereas we use an additional history field to maintain the ordering.

5 Conclusions and Future Work

This paper has presented a garbage collection scheme which is suitable for a language with non-destructive semantics.

We have shown how the basic algorithm can be extended to handle both real-time requirements and generational collection.

One research field currently being investigated with Erlang is large real-time distributed databases. Real-time copying collectors are very bad at handling large amounts of data [11] so an alternative collector is needed. As Erlang is a non-destructive language then this type of mark-sweep collector has distinct advantages. Whether the lack of compaction leads to an unacceptable amount of fragmentation is something which must be investigated.

Work is at present going on in implementing Erlang with a garbage collector based on the principles presented in this paper. While not yet complete the initial results seem promising. Future work will look at the possibility of combining an allocator of this type with a copying collector in a generational scheme. Hopefully this will combine the best aspects of both.

References

1. Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.
2. Lennart Augustsson. A compiler for lazy ML. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 218–227, Austin, Texas, August 1984. ACM Press.
3. Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978. Originally appeared as MIT Artificial Intelligence Laboratory Working Paper No. 39, February 1977.

4. David A. Fisher. Bounded workspace garbage collection in an address-order preserving list processing environment. *Information Processing Letters*, 3(1):29–32, July 1974.
5. T. Johnsson. Efficient compilation of lazy evaluation. In M. Van Deusen, editor, *Compiler construction: Proceedings of the ACM SIGPLAN '84 symposium (Montreal, Canada, June 17–22, 1984)*, ACM SIGPLAN Notices, vol. 19, no. 6, June, 1984, pages 58–69, New York, NY, USA, 1984. ACM Press.
6. Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.
7. H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, 1967.
8. Guy L. Steele Jr. Data representation in PDP-10 MACLISP. MIT AI Memo 421, Massachusetts Institute of Technology, 1977.
9. Motoaki Terashima and Eiichi Goto. Genetic order and compactifying garbage collectors. *Information processing Letters*, 7(1):27–32, January 1978.
10. David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, pages 1–17, November 1988. Published as Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, number 11.
11. Robert Virding. A garbage collector for the real-time concurrent language Erlang. Submitted to IWMM95.
12. Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

This article was processed using the LaTeX macro package with LLNCS style