Department of Computing Science and Mathematics
University of Stirling

The Implementer's Dilemma: A Mathematical Model of Compile Time Garbage Collection

Simon B Jones
Andrew S Tyas

Department of Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-786-467421, Facsimile +44-786-464551
Email sbj@compsci.stirling.ac.uk

Technical Report CSM-118

September 1994

Abstract

Optimization by compile time garbage collection is one possible weapon in the functional language implementer's armoury for combatting the excessive memory allocation usually exhibited by functional programs. It is an interesting idea, but the practical question of whether it yields benefits in practice has still not been answered convincingly one way or the other.

In this short paper we present a mathematical model of the performance of straightforward versions of mark-scan and copying garbage collectors with programs optimized for *explicit deallocation*. A mark-scan heap manager has a free list, whereas a copying heap manager does not — herein lies the dilemma, since a copying garbage collector is usually considered to be faster than a mark-scan, but it cannot take advantage of this important optimization.

For tractability we consider only heaps with fixed cells.

The results reported show that the garbage collection scheme of choice depends quite strongly on the *heap occupancy ratio*: the proportion of the total heap occupied by accessible data structures averaged over the execution of the program. We do not know what typical heap occupancy ratios are, and so are unable to make specific recommendations, but the results may be of use in tailoring applications and heap management schemes, or in controlling schemes where the heap size varies dynamically.

An important result arising from the work reported here is that when optimizing for explicit deallocation, a very large proportion of cell releases must be optimized before very much performance benefit is obtained.

Acknowledgements

This paper was presented at the 1993 Glasgow Workshop on Functional Programming, held at Ayr in July 1993. The authors would like to thank the two referees, Pieter Hartel and Patrick Sansom, for their very useful comments on the draft version of the paper.

(The paper is to appear in a form very close to this in the proceedings of the above workshop: *Functional Programming, Glasgow 1993*, published by Springer Verlag.)

Compile time garbage collection: A reminder

Compile time garbage collection is an optimization technique in which a compile time analysis is applied to a program to determine whether special purpose storage management operations can be placed in the compiled code. The aim is to place them where the expense of run time decision making about the recycling of storage cells can be avoided: run time garbage collection is replaced by "compile time garbage collection" (for one approach to this see [3]). There are two possible optimizations:

- Deallocation: when, at a particular point in a program where a cell reference is discarded, *every* time that the discard occurs it fully dereferences the cell, then the cell can be explicitly returned to the pool of free cells.
- Direct re-use (or destructive allocation): when a cell can be explicitly deallocated, and a fresh cell is required immediately afterwards, then the dereferenced cell can be re-used directly without the expense of passing it via the free list. (This is easy in the case of fixed size cells, but of more limited scope if cells are variable sized.)

We consider the case of a simple first order functional language with pattern matching of function arguments, and strict, left to right evaluation.

Example: Deallocation Consider the following function definition:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

and the reduction sequence for a "main program" consisting of the expression `sum (1:2:3:[])`:

```
   sum (1:2:3:[])
=> 1 + sum (2:3:[])
=> 1 + (2 + sum (3:[]))
=> 1 + (2 + (3 + sum []))
=> etc...
```

In each of the three reduction steps shown, the second equation for `sum` has been used, and the head cons cell that appears in the pattern matched argument is not referred to in the result of the reduction. Thus the second equation of `sum` is optimizable: code could be inserted which returns the head cons cell to the free list as soon as `x` and `xs` have been extracted from it.
[Note that `sum` may not be optimizable in all contexts: if the argument list is not an unshared storage reference, then `sum` cannot be optimized. Here is an example of this problem: suppose that `sum` is called in the context:

```
f xs = sum xs + sum xs
```

with a call of `f` as the main program expression. In the left hand `sum xs`, the `xs` does not have a unique reference, so it would be disastrous if `sum` were to deallocate the cons cells of `xs`.]

Example: Direct re-use Consider the following function definition:

```
append [] ys = ys
append (x:xs) ys = x : append xs ys
```

and the reduction sequence for a "main program" consisting of the expression `append (1:2:[]) (3:[])`:

```
   append (1:2:[]) (3:[])
=> 1 : append (2:[]) (3:[])
=> 1 : 2 : append [] (3:[])
=> etc...
```

In each of the two reduction steps shown, the second equation for `append` has been used, and the head cons cell that appears in the pattern matched argument is not referred to in the result of the reduction, but a fresh cons cell has been allocated for the result. Thus the second equation of `append` is optimizable: code could be inserted which re-uses the head cons cell of the argument to construct the result.

Again, this optimization depends on the context of the call of `append`.

These simple examples show the possibilities, and illustrate that the optimization requires a *global analysis* of the program. Schemes have been proposed for carrying out this analysis, for example [3]. In this paper it is not our concern to discuss the analysis techniques themselves, but to consider the interaction between optimization options and heap management schemes.

Which heap management scheme? A dilemma for optimization

The implementer who wishes to choose a heap management scheme and an optimization technique faces a dilemma. Simplifying the choice of heap management schemes, we have:

- "Mark-scan": Here the time per gc is proportional to the size of the heap; thus, for programs which do not occupy much memory relative to the heap size, it is an expensive option. However, since the scheme involves a free list, *both* deallocation *and* re-use optimizations are possible. It is not widely used at present.

- "n-space copying": Here time per gc is proportional to the number of accessible cells; thus, for programs which do not occupy much memory relative to the heap size, it is a fast option. Most copying schemes do not employ a free list, and so *only* re-use optimization is possible; thus we would expect that less optimization could be carried out than with a mark-scan scheme. Currently, this gc scheme is widely used.

We now see the implementer's dilemma: "slow gc/high optimization" *vs* "fast gc/low optimization". It is pertinent to ask the question:

How does the performance of
mark-scan + deallocation optimization
compare with
copying (with no optimization) ?

Note: The referees suggested that, since there exist mark-scan gcs *without* free lists (e.g. [4]), and also copying gcs *with* free lists, the question could be more generally phrased as:

How does the performance of
a gc with free list + deallocation optimization
compare with
a gc without free list (with no optimization) ?

This paper then addresses the question in one particular context.

An analytical assessment of deallocation optimization

In [5] Tyas presents an analytical model for predicting the total time involved recycling storage cells — that is in returning cells which have no references to the free list. The analysis is performed for mark-scan with an adjustable percentage of the released cells returned to the free list via explicit deallocation, and for copying with no deallocation optimization (because it is not possible).

The models assume that the cells are of a uniform, fixed size (so that the model does not need to deal with fragmentation), and that the active heap size, that is the actual number of accessible cells, is at a steady state. The copying heap scheme comprises two semi-spaces; therefore, since the total available memory in both cases is assumed to be the same, each semi-space is half the size of the mark-scan heap (the formulae are easily adjusted to handle the case of them being the same size).

The models are expressed in terms of the following parameters,:

H	The total number of heap cells available (maximum memory size).
R	The mean number of cells in use (accessible in the hypothetical steady state).
N	The total number of cell allocations that occur in a given program.
P	The percentage of released cells that are explicitly deallocatable.

and the following time constants are required :

C_1	Time to copy a cell.	(Estimate: 48μs)
C_2	Time to add a cell to the free list.	(Estimate: 5μs)
C_3	Time to mark & unmark a cell.	(Estimate: 16μs)
C_4	Time to determine the mark on a cell.	(Estimate: 2μs)

The values estimated for the time constants were obtained on an HP9000/375 (details are given in [5]). They agree in their magnitudes with those given in [1], which reports a similar, but simpler analysis comparing copying collection with stack allocation. (These values must be accepted in a qualified way, since there may be cache effects involved.)

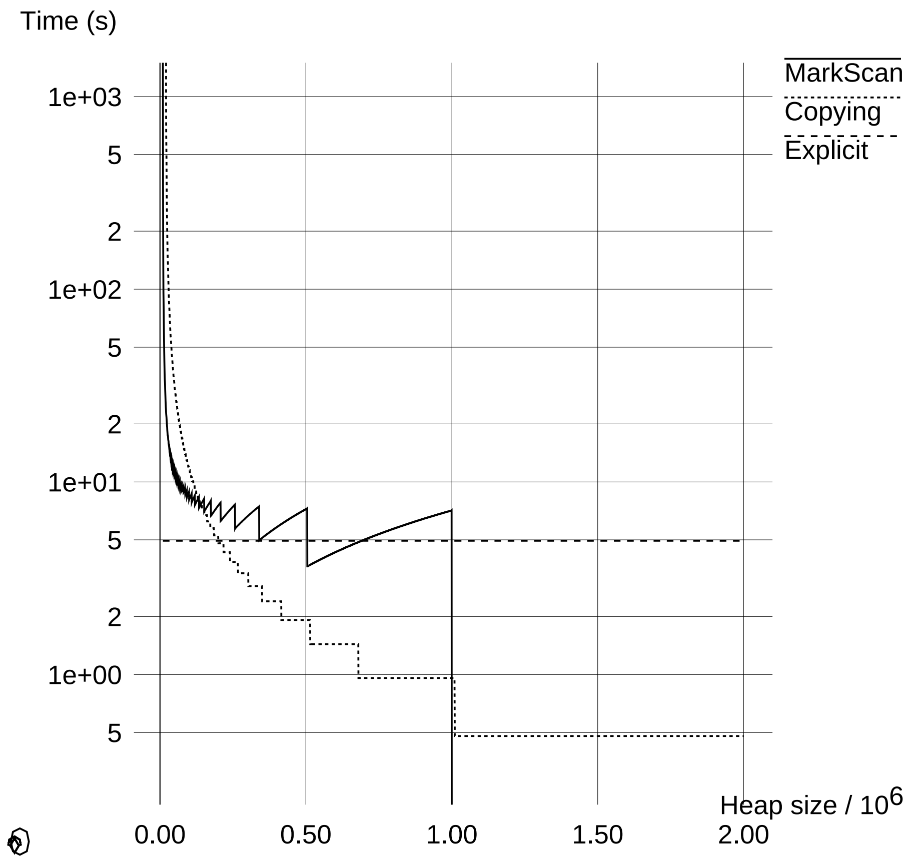
[5] derives the following formulae for the total time taken in recycling cells with mark-scan gc, T_{ms}, and copying gc, T_{cp} :

$$\begin{aligned} T_{ms} &= (C_2(H-R)+C_3R+C_4H)\left\lfloor \frac{\frac{100-P}{100}(N-R)}{H-R} \right\rfloor + C_2\frac{P}{100}(N-R) \\ T_{cp} &= C_1R\left\lfloor \frac{N-R}{H/2-R} \right\rfloor \end{aligned}$$

Here are some hints for interpreting these formulae:

- $N-R$ The total number of cells released by the program (since R is the steady state occupancy, we assume that this many cells remain allocated at the end of the execution).
- In mark-scan $\frac{P}{100}(N-R)$ cells are explicitly deallocated, and $\frac{100-P}{100}(N-R)$ are collected by ordinary garbage collection.
- $H-R$ The number of cells that must be recycled at each mark-scan gc. ($H/2-R$ for copying.)
- $\left\lfloor \frac{\text{\# of released cells}}{H-R} \right\rfloor$ The number of mark-scan gcs. ($H/2$ instead of H for copying.
- $C_2(H-R)+C_3R+C_4H$ Time for a single mark-scan gc: C_3R: mark and unmark accessible cells; C_4H: check mark on *all* cells in the heap; $C_2(H-R)$: add all released cells to the free list.

The figure (next page) graphs the formulae for (i) mark-scan with no optimization, i.e. T_{ms} with $P=0\%$ (the MarkScan line), (ii) mark-scan with *all* cells recycled explicitly, i.e. T_{ms} with $P=100\%$ (the Explicit line), (iii) copying, T_{cp} (the Copying line). In each case the total number of allocations, N, is 10^6 cells, and the steady state occupancy, R, is 10^4 cells. Practical experiments (reported in [5]) confirm the validity of the curves, but it is hard to reproduce them precisely. For intermediate values of P, in the case of mark-scan, the general trend of the curves is intermediate between the $P=0\%$ and $P=100\%$ cases shown (but their saw-tooth nature causes them to cross and re-cross as the heap size grows). Further, P must be substantially greater than 50% before the curve drops significantly below the $P=0\%$ curve.



The variation of total recycling time with heap size. $N = 10^6, R = 10^4$.
Unopt mark-scan (MarkScan), fully opt mark-scan (Explicit), and Copying.
Note: at heap size $1 * 10^6$ mark-scan time drops to zero,
and at heapsize $2 * 10^6$ the copying time drops to zero.
(*Reproduced from [5]*)

The most interesting and relevant part of this graph is for heap sizes between 0 and $0.5 * 10^6$ cells — this is where the executing program performs many gcs.

We can observe the following:

- Unoptimized mark-scan is faster than copying if $H/R < 12$
- Copying is faster than fully optimized mark-scan if $H/R > 20$
- Therefore, if the available heap is at least "15 or so" times the size of the average occupancy, then we ought to be using a copying collection scheme rather than a mark-scan scheme optimized for deallocation.

We can compare the results above with the conclusion of [1]. Appel analyses the cost per collected cell of gc for a copying collector and compares this with the cost of popping a cell from a stack. Re-casting in the notation of this paper, Appel's cost per cell in a copying gc, g, is:

$$g = \frac{C_1 R}{N-R} \left\lfloor \frac{N-R}{H/2-R} \right\rfloor = \frac{T_{cp}}{N-R}$$

This is consistent with Tyas' formula above. Copying is cheaper than explicit deallocation, and hence fully optimized mark-scan, if

$$g < C_2$$

With his time constants, Appel deduces that this condition is equivalent to

$$H/R > 14$$

This heap occupancy ratio is consistent with Tyas' predictions.

Conclusions

The results of the previous section are quite interesting: they predict that, if our programs will be running with a relatively small heap (less than roughly 15 times the average occupancy of the program), then we would be better with a mark-scan heap management scheme (whether optimized or not); otherwise a copying scheme will behave better. An important new result obtained from the analysis presented here is that, if the heap is small and we are using optimized mark-scan, then we need a *very high proportion* of optimized deallocations before the performance benefit is appreciable (well over 50%).

The threshold occupancy ratio observed above is probably too high. The analysis here does not take into account the full heap management costs: the time for the *allocation* of cells has been omitted (since it is not altered by the deallocation optimization). Heyman [2] has a more complex memory management scheme which incorporates allocation costs, but does not take into account deallocation optimization; his results place the crossover point at 2.8.

So, the choice of scheme depends on the expected heap occupancy ratio. Unfortunately we have no available data on typical heap occupancy ratios, so we are unable to make specific recommendations. It is hard to predict how large-scale functional application programs and large-scale functional programming systems are going to develop, and so it is hard to know what typical occupancy ratios are realistic. In the conventional software arena, programs have grown to fit the ever larger memories available, and there is no reason to assume that the same won't be true in the functional arena. Perhaps the results in this paper, and follow-up work on re-use optimization and other heap management schemes, may help to establish rules of thumb for tailoring heap managers and heap sizes to application programs, or for controlling the behaviour of heap managers which adapt the heap size dynamically.

References

[1] A.W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[2] J. Heyman. A comprehensive analytical model for garbage collection algorithms. *ACM SIGPLAN Notices*, 26(8), August 1991.

[3] S.B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 54–74, 1989.

[4] P. Sansom. Combining copying and compacting garbage collection. In R. Heldal, C.K. Holst, and P. Wadler, editors, *1991 Glasgow Workshop on Functional Programming.* Springer-Verlag, Workshops in Computing, August 1991.

[5] A.S. Tyas. An investigation into the optimization of garbage collection within functional languages. Final Year Dissertation, Department of Computing Science and Mathematics, University of Stirling, April 1993.