# An incremental garbage collector for embedded real-time systems

*Malcolm Wallace*
*Colin Runciman*

## Abstract

Embedded real-time systems often have a single processor and small memory. This combination runs contrary to frequent architectural assumptions in the context of functional programming—parallel processing and/or large memory. The dichotomy is especially a problem when it comes to memory management, for an incremental garbage collection scheme is needed to achieve real-time, but present algorithms require either an extra processor or additional memory—the very thing an embedded system cannot afford. We present an incremental garbage collector of the mark-sweep type, which minimises structural overhead without sacrificing performance. It uses a very small stack for marking the live heap, with 'safety bits' to ensure that stack overflow does not hinder collection. We report on how the algorithm performs and discuss its correctness.

## 1. Introduction

Traditional garbage collectors (GCs) for functional languages run on a *stop-and-collect* basis. When there is no free heap left, the computation is interrupted, a garbage collection is invoked to reclaim dead space, and then the computation resumes. Real-time systems, however, require some guaranteed throughput and cannot allow pauses for arbitrary lengths of time at unspecified intervals. An *incremental* approach must be taken. In other words, by doing some small, known, amount of GC work at frequent intervals, we ensure that the supply of free heap cells never dries up. Overall the amount of work expended on GC might be higher, but the performance of the system is more evenly predictable.

For more information on general uniprocessor GCs, see Wilson's review paper [Wilson92] which includes a number of real-time algorithms. Many such real-time algorithms for garbage collection have been suggested, employing incremental techniques. Most, however, assume that

Authors' affiliation: Department of Computer Science, University of York, YO1 5DD, United Kingdom. E-mail: {malcolm|colin}@minster.york.ac.uk

- an *additional processor* will be available to handle GC in parallel 'on-the-fly' (e.g. [Queinnec et.al. 89, Ben-Ari84, Appel et.al. 88]; see also the review paper [Abdullahi et.al. 92] on general distributed garbage collection),

and/or that

- *additional memory* can be used to offset the extra time costs of the incremental algorithm [Baker78, Baker92, Yuasa90, Brooks84, Lieberman & Hewitt 83].

Neither of these assumptions is necessarily valid for embedded real-time systems. Although many applications of functional languages can safely assume large memory spaces, in embedded systems the amount of memory available is severely limited by cost: when designing electronic instruments, for example, the microprocessor element is only one part of the overall device, and one wishes to keep the cost of each part as low as possible if the device is destined for mass production. For many embedded systems, a single processor with small memory is the ideal.

Section 2 briefly reviews two previous incremental GCs, Mark-During-Sweep [Queinnec et.al. 89] which assumes an additional processor, and Stack Collect [Yuasa90] which assumes additional memory. In section 3, we present an improved incremental GC algorithm for a uniprocessor, with small memory overhead, which can be viewed as a hybrid of the two algorithms discussed in section 2. Section 4 sketches a proof of the new algorithm. Section 5 concludes, reporting our experience using the algorithm in a real embedded system.

This new garbage collection algorithm is just one element of a larger project, in which we aim to show that a pure functional language can handle embedded applications hitherto only programmed in languages such as Modula, Ada, and C.

## 2. Real-time collectors

Most real-time garbage collectors require memory much larger than the size of the usable heap. Baker in his 1978 paper (and later variants) relies on *semi-spacing*, which requires twice the usable heap space; *generational* collectors [Lieberman & Hewitt 83] work on a similar principle; and the *treadmill* [Baker92] requires the layering of a doubly-linked circular list on top of the heap. In addition, many of these algorithms are designed to optimise virtual memory performance (the treadmill is a notable exception); an embedded system has no virtual memory, and often cannot afford the expense of much non-active-heap memory. In this respect, a *mark-sweep* [McCarthy60] based collector is more attractive for an embedded system, since its overhead is a single bit per heap cell.

A traditional stop-and-collect mark-sweep algorithm is generally invoked when the *freelist* becomes empty. The *marker* recursively traces the graph reachable from the computational roots, marking cells as it goes, after which the *sweeper* scans linearly through the heap, linking garbage (unmarked) cells into a new freelist. To convert mark-sweep to an incremental scheme, we effectively take a snapshot of the roots at a point in time before the freelist is exhausted, and interleave the operation of the garbage collector with the operation of the *evaluator/mutator*. We must add some extra functionality into the mutator to ensure that any new sections of graph that are created will also be traced and marked. However, no special action is needed to cater for sections of graph which become garbage after the marker has started: they will either be picked up in the current cycle (if not yet marked), or the next (if already marked).

Queinnec et.al.[89] describe such a real-time GC algorithm that performs incremental work in both the mark and sweep phases simultaneously, with each phase operating on distinct sets of mark bits. When both phases terminate, the set of mark bits just created by the marker is passed to the sweeper; meanwhile, the marker re-starts with a fresh set of mark bits. Hence, the sweeper is always working on an older set of mark bits than the marker, while both are interleaving their activity with the mutator. In this way, *Mark-During-Sweep* is a cautious algorithm: some cells which are not part of the live heap may be marked, with the implication that some garbage may not be recycled immediately—there may be up to two GC cycles after a cell becomes garbage but before it is added to the free list.

The state of the on-going marking operation must be recorded explicitly in any incremental algorithm, since the marker must be interleaved with the mutator. Queinnec et.al. use *grey colour* bits (originally due to [Dijkstra75]) for this purpose. Marked (black) cells can only point to unmarked (white) cells indirectly, through half-marked (grey) cells. The set of grey cells is the ‘wavefront’ dividing the set of black cells from the set of white cells. As a grey cell is examined by the marker, the cells it points to are shaded grey (if they are not grey or black already), and the cell itself is darkened to black. Marking commences with only the computational roots shaded grey, and all else white. As marking continues, the wavefront gradually moves into the white region until the entire reachable graph has been marked black and there are no grey cells left, at which point marking is complete and the sweeping phase may begin.

The important aspect of Mark-During-Sweep is that the marker can take a new snapshot of the roots and continue working (with a fresh set of mark bits and grey bits) whilst the sweeper reclaims as garbage those cells unmarked in the previous cycle.

Mark-During-Sweep has a low memory overhead, requiring just 3 bits per heap cell—two denoting black/white and grey for the marker, and one denoting black/white for the sweeper. The inefficiency of the algorithm is that every invocation of the incremental marker must search the heap for grey cells. This has a worst-case behaviour where the time for a full mark phase is quadratic in the heap size. Queinnec et.al. propose that efficient implementation of the marker should be possible with an additional parallel processor.

Yuasa [90] on the other hand describes an incremental scheme where the state of the marking phase is recorded in an explicit stack (rather than using grey bits). In colour terms, the grey cells are those currently on the stack—a cell on the stack has been marked, but its child pointers have not yet been examined. As cells are popped off the stack, their pointers are followed and the cells pointed at are pushed onto the stack, unless they have already been marked.

This *Stack Collect* algorithm avoids the time inefficiency of Mark-During-Sweep—worst-case behaviour of the marker takes time proportional to the size of the live heap—but introduces a space inefficiency: the additional memory needed for the stack in the worst case grows linearly with the size of the heap. Also, Yuasa's algorithm does *not* take advantage of the possibility of running both the marker and sweeper together incrementally.

## 3. A hybrid algorithm

Our algorithm combines the best time and space properties of Mark-During-Sweep and Stack Collect. We perform marking and sweeping simultaneously, using an explicit stack to record the progress of the mark phase. However, we set a *very small* upper bound on the depth of the stack (for instance, one ten thousandth of the heap size). When the stack overflows, we bring the grey bits into action as 'safety' bits. Once the stack is empty again, we search for any grey cells and place them back on the stack. We call the algorithm *Stack-Safety*.

The idea is that for most graph structures, marking can be accomplished entirely through the stack. The occasional expense of an (incremental) search through the heap for a grey cell will be tolerable. Worst-case time behaviour is still quadratic in principle, though in practice most marking can be achieved in time proportional to the size of the live heap. As for memory costs, we require three bits per heap cell (as in Mark-During-Sweep) plus a small constant for the bounded stack.

Let us assume that the heap is a collection of `H` binary cells holding a `car` and `cdr`, which can either be pointers to other cells in the heap, or some other sort of value. See Figure 3.1. For each cell we also store two mark bits (one for the marker, one for the sweeper) and a grey bit (for the marker). Global variables, `m` and `s`, record which bits each of the marker and sweeper is currently using. The stack has depth `D`.

```
value    :: tagged item
          | ptr to cell
cell     :: record of
             car  :: value
             cdr  :: value
             mark :: array [A,B]
                     of bit
             grey :: bit
heap     :: array [1..H] of cell
gcStack  :: array [1..D] of cell
gcSp     :: integer 0..D
m, s     :: A,B
freelist :: ptr to cell
```

**Figure 3.1**

Note that a **`ptr to`** `cell` is effectively an index `1..H` into the `heap` array. That is, `hptr->car` is synonymous with `heap[hptr].car`.

### 3.1. The marker

The marker performs its work in small bursts of activity: using the stack as a scratchpad which persists between the bursts, it recursively traces the graph from the roots. When a cell is marked, it is placed on the stack. Having been marked, its sub-graph is traced by removing the cell from the stack and placing its `car` and `cdr` on the stack (and marking them), unless they are non-cell-pointers, or have already been marked. See Figure 3.2.

```
Km :: integer constant

marker =
    repeat Km times
        c := gcPop
        if c =/= EMPTY then
            gcPush(c->car)
            gcPush(c->cdr)
        else
            c := searchForGreyCell
            gcPush(c)
```

**Figure 3.2**

The stack is bounded, so we handle overflow through the push operation: specifically, the grey bit records an item that cannot fit on the stack due to overflow. Such cells will be pushed onto the stack again at some later time when there is room for them, either when the stack is empty, or through being reachable from another cell on the stack. The push operation checks that its argument is a cell pointer, and that the cell in question is unmarked. See Figure 3.3. If the pop operation revealed an empty stack, the marker looks for a grey cell to push onto the stack.

The mark bit operations are straightforward. See Figure 3.4. It is an invariant that a cell cannot be both black and grey—the operations `markCell` and `shadeCell` enforce this. We keep a count of the number of grey cells to ease testing for termination of the marker.

The operation to search for a grey cell is shown in Figure 3.5. The search is incremental, performing a fixed small amount of work on every invocation (determined by the constant `Kg`). A persistent pointer records the current search position.

```
gcPush(c) =
    if not isCell(c) then
        return
    if c->mark[m] then
        return
    if gcSp < D then
        markCell(c)
        inc(gcSp)
        gcStack[gcSp] := c
    else
        shadeCell(c)

gcPop =
    if gcSp > 0 then
        c := gcStack[gcSp]
        dec(gcSp)
    else
        c := EMPTY
    return c
```

**Figure 3.3**

```
greyCount :: integer 0..H

markCell(c) =
    if c->grey then
        c->grey := FALSE
        dec(greyCount)
    c->mark[m] := TRUE

shadeCell(c) =
    if not c->mark[m]
    and not c->grey then
        c->grey := TRUE
        inc(greyCount)
```

**Figure 3.4**

```
search :: ptr to cell
Kg     :: integer constant

searchForGreyCell =
    if greyCount > 0 then
        repeat Kg times
            inc(search)
            if search > H then
                search := 1
            if search->grey then
                return search
    return EMPTY
```

**Figure 3.5**

### 3.2. The sweeper

The sweeper also performs a burst of work on every call, using a pointer into the heap as its persistent state between calls. See Figure 3.6. Mark bits which were set by the marker in the previous cycle are reset again for the next pass of the marker; unmarked cells are joined onto the freelist. Cells already on the freelist must not be swept to the freelist again—they are identified by their tag.

```
scan :: ptr to cell
Ks   :: integer constant

sweeper =
    repeat Ks times
        if scan > H then
            return
        if scan->mark[s] then
            scan->mark[s] := FALSE
        else
        if scan->car =/= FREE then
            scan->car := FREE
            scan->cdr := freelist
            freelist  := scan
        inc(scan)
```

**Figure 3.6**

### 3.3. The collector

A routine is needed to co-ordinate the marker and sweeper, ensuring that they do not interfere. When the marker and sweeper have both completed all their work, the marker's bits are swapped with the sweeper's. The roots of the computation are marked to start the new 'wave-front'. See Figure 3.7.

```
incrementalGC =
    if scan > H
    and gcSp == 0
    and greyCount == 0 then
        swap (m,s)
        scan := 1
        for c in rootset
            gcPush(c)
    else
        marker
        sweeper
```

**Figure 3.7**

### 3.4. The allocator

A burst of GC is invoked on each allocation of heap storage. See Figure 3.8. Every newly allocated cell is marked immediately.

```
cons(l,r) =
    incrementalGC
    c        := freelist
    freelist := freelist->cdr
    c->car   := l
    c->cdr   := r
    gcPush(c)
    return c
```

**Figure 3.8**

### 3.5. The mutator

For every destructive alteration to a cell pointer performed by the mutator, the mutator must guarantee to mark the destination cell of the new reference. See Figure 3.9. In addition, any destructive update to the rootset must lead to a marking operation.

```
rplaca(c,l) =
    c->car := l
    gcPush(l)

rplacd(c,r) =
    c->cdr := r
    gcPush(r)
```

**Figure 3.9**

### 3.6. Discussion

When a cell is shaded grey, it does not necessarily follow that a search through the heap will be needed to find that cell again later. Even if one path to that cell from a root overflows the stack, another shorter path may exist from the same root or from a different root altogether. The cell, though grey, may still be marked through the stack, avoiding the need to search for it.

Ideally, the marker and sweeper should reach the end of their cycles together. The constants `Km` and `Ks`, determining how much work will be done by the marker and sweeper in each call, must therefore be chosen carefully to give acceptable performance. The most appropriate values may depend on the program under evaluation. One possibility is to make them variables, and extend the garbage collector to adjust their values dynamically to redress any imbalance at the end of each phase. This approach may, however, have an unreasonable effect on the predictability of the algorithm.

The stack depth is another constant which may need to be tuned to suit the computation. If a single perfectly balanced tree fills the entire heap, a stack of $\log_2 H$ obviates any need for grey bits. In the worst case, a tree with a backbone stem and a single leaf at each node requires a stack of $H/2$ cells to obviate grey bits. As the stack depth approaches this upper limit, we approximate Yuasa's Stack Collect. As it approaches zero, we approximate Queinnec et.al.'s Mark-During-Sweep. A compromise must be found which keeps both the stack and the grey count small.

The constant `Kg` will not depend on the particular program being evaluated - its value should be chosen solely to balance the two branches of the `if` statement in `marker`. A balanced conditional ensures minimum variation in the cost of a call to `marker`.

### 3.7. Optimisations

Possible optimisations of the algorithm (besides compiler techniques such as in-lining function calls) include:

- The mark bits associated with cells can be compacted into separate bit-maps, making bit operations more efficient, especially since virtual memory performance is not an issue. In particular, searching for a grey cell will be faster.
- The range in which grey cells lie might be marked by maximum and minimum pointers. If the grey count indicates a single grey cell, the max and min will be equal and point to that cell. If there are two grey cells, max and min will point to them both. Otherwise, searching need only be attempted within the indicated range. However this technique might interact badly with the marking of grey cells due to sharing rather than searching, as described above.

## 4. Sketch of proof

We must show that the marker has marked (at least) every cell which is reachable from the root at the end of the marker cycle, and therefore that the sweeper collects no live cells. We must also show that every unreachable cell is eventually reclaimed as garbage.

### 4.1. Unbounded stack

Assuming an unbounded stack, a static examination of the code allows us to make the following statements about the marker, within a single GC cycle:

- Once a cell has been marked, it cannot be unmarked again. (The only operation to clear a mark bit is in the sweeper.)
- For a cell to be marked it must be placed on the stack. Being on the stack implies a cell has been marked. (The only operation to set a mark bit is in `markCell`, which is called only from `gcPush`.)
- The roots are initially on the stack. (See the code for `incrementalGC`.)
- If the cell at the top of the stack points to another cell, the latter cell will be placed on the stack or already be marked. (`marker` calls `gcPush` on the first cell's `car` and `cdr`.)

Now, given termination, which says that the stack will eventually empty (see argument below), we make the inductive step: for every cell which is on the stack at some time, the cells it points to will also be on the stack at some time. Further:

- If a cell's `car`/`cdr` is altered before it appears on the stack, the cell previously referenced through the `car`/`cdr` is no longer reachable by that path, and need not be marked.
- Hence, a subset of the cells reachable from the roots at the *start* of the cycle appears on the stack during the course of the cycle, each element of which is marked. Those cells which were reachable from the roots at the start of the cycle but have not been marked by the end of the cycle are unreachable at the end of the cycle.

For *safety*, we must show that all cells reachable from the roots at the *end* of the cycle are marked. These cells will be a subset of the cells reachable from the roots at the *start* of the cycle, plus any cells made reachable by the mutator *during* the cycle. But all cells allocated by, and all pointers destructively updated by, the mutator are placed on the stack. Hence, at the end of a GC cycle, the set of marked cells is a superset of the reachable cells at that moment, and an unmarked cell is indeed garbage.

For *progress*, we must show that any unreachable cell not on the freelist will at some time be identified as garbage and be added to the freelist. Reachable cells may be made unreachable by the mutator during a GC cycle (by destructive update of pointers). If at the moment of being made unreachable in cycle $n$, a cell has not yet been marked, it will remain unmarked to the end of cycle $n$ and hence be reclaimed by the sweeper in cycle $n+1$. If it was already marked, it will remain marked to the end of cycle $n$. It will be unreachable from the roots at the start of cycle $n+1$, therefore it will be unmarked throughout cycle $n+1$, thereafter being reclaimed by the sweeper in cycle $n+2$.

### 4.2. Termination

As regards proof of termination of one cycle of the marker, we state:

- Each iteration removes one cell from the stack (via `gcPop`).
- Any cell can be placed on the stack once only. (`gcPush` ensures that if a cell is already marked, it will not be placed on the stack again.)
- There is an upper bound on the number of reachable cells, namely, the heap size.
- Therefore the stack will eventually empty.

### 4.3. Bounded stack

Extending our argument to allow for a bounded stack with safety bits, we now show that a grey cell can be treated for the purposes of proof as if it were on the stack.

- Once a cell is grey, it can only become black, not white. (The only operation to reset a grey bit is in `markCell`.)
- Once a cell is black, it can become neither grey nor white. (The only operation to set a grey bit is in `shadeCell` which checks that the cell is not already black.)
- If a cell is on the stack, it is black. (`gcPush` ensures this.)
- The roots are initially either grey or on the stack. (`incrementalGC` calls `gcPush` on the roots.)
- If the cell at the top of the stack points to another cell, that cell will be placed on the stack or become grey. (`marker` calls `gcPush`.)
- When the stack is empty, any remaining grey cell is found and placed on the stack (by the `marker`.)
- Hence (given termination) every grey cell eventually blackens by transferral to the stack, and by induction, every cell reachable from the roots at the start of the cycle appears on the stack, and therefore is blackened.

The *safety*, *progress*, and *termination* arguments continue to hold when every grey cell can be treated as if on the stack.

## 5. Results and conclusions

We have adapted the Gofer [Jones91] interpreter to run in an embedded real-time system built locally to assist with undergraduate teaching. The apparatus consists of a chute which sorts glass and metal marbles into separate collecting bins. There are four effectors and two sensors in addition to terminal and host connections. The sensors rely on interrupts to signal events, and there are timing deadlines on when to activate the effectors following an interrupt.

Before development of the alternative GC, we pushed this real-time application to its maximum speed, whilst still correctly meeting its deadlines. It ran at this speed only until the freelist was exhausted, at which point it collapsed due to the pause for mark-sweep GC. We then implemented our Stack-Safety algorithm as a replacement GC for the interpreter. The application can now run continuously through any number of GC cycles. However the Stack-Safety routines currently account for 30-45% of runtime! Since the system can now run at about only 55-70% of its previous speed, we have had to relax the timing accordingly so it can meet its deadlines once again. We hope to reduce the GC overhead significantly, although it is unlikely we can reduce the percentage of time chargeable to Stack-Safety below 20%. The percentage overhead of mark-sweep was only 7-10%, but the comparison is not on equal terms, since mark-sweep caused the application to run incorrectly.

In our experiments with our algorithm we varied the constant values for the number of increments in each burst `Km` and `Ks`, the stack depth `D`, and the heap size `H`. The value of the search constant `Kg` was fixed at 3 to balance the branches of the marker. We recorded the minimum length $f$ of the freelist during a cycle; the number of cells marked and reclaimed in a cycle (giving the approximate live heap size); the number of stack overflows; the maximum number of grey cells; the number of grey cells which are marked by sharing rather than searching, and so on. The following tables show measurements of such values during a typical GC cycle in one of our test programs which has a live heap of fairly constant size—about 2,250 cells (±40).

Unfortunately, the particular version of Gofer we used (v2.21) stored a character cache, CAFs and other information in the heap, leading to a large rootset (about 650 root cells), so in fact about 1,500 cells of the live heap of this test program was static data. We have excluded the roots from the stack overflow count for this reason. Tests on a more recent version of Gofer (v2.28) have reduced the static overhead, but made little difference to the other measures.

The *work ratio* is a measure of parity between the work being done by the marker and the sweeper. Each figure is the number of calls to the marker during which there still remained work for the marker to do, normalised against the number of calls to the sweeper during which there remained work for the sweeper to do. Values above unity show the marker did more work; values below unity show the sweeper did more work. Ideally (for cost prediction), we want the marker and sweeper to complete their work in the same number of bursts. GC cycle time—and hence the minimum freelist size—is also affected by any disparity.

Table 1 shows that (for this test program) only a very small stack is needed to eliminate stack overflows completely when traversing the live graph.

| H=40000, Km=4, Ks=20, Kg=3 | | | |
|---|---|---|---|
| D | overflow | $f$ | work ratio |
| 1 | 396 | 16120 | 5.361 |
| 2 | 75 | 22175 | 3.669 |
| 3 | 1 | 28756 | 2.002 |
| 4 | 0 | 28343 | 1.999 |

**Table 1**

Table 2 illustrates that varying the number of iterations in each burst of marking brings the work ratio closer to parity—achieved for this program when `Km`/`Ks` is 8/20.

| H=40000, D=3, Ks=20, Kg=3 | | | |
|---|---|---|---|
| Km | grey | $f$ | work ratio |
| 4 | 650 | 28756 | 2.002 |
| 5 | 650 | 30871 | 1.601 |
| 6 | 648 | 31649 | 1.334 |
| 7 | 650 | 32244 | 1.143 |
| 8 | 652 | 32699 | 1.001 |
| 9 | 652 | 32685 | 0.890 |

**Table 2**

Table 3 indicates that it is the ratio `Km/Ks` which determines parity. The absolute values of `Km` and `Ks` determine the time taken for a complete cycle, and hence the minimum heap space required.

| H=40000, D=3, Kg=3 | | | |
|---|---|---|---|
| Km | Ks | work ratio | $f$ |
| 2 | 5 | 1.001 | 21759 |
| 4 | 10 | 1.000 | 29648 |
| 6 | 15 | 1.001 | 31419 |
| 8 | 20 | 1.001 | 32699 |
| 10 | 25 | 1.001 | 33627 |

**Table 3**

Keeping the ratio `Km/Ks` constant, we varied the heap size downwards, and discovered that because the live heap size (and the marking task) stays constant whilst the sweeper has less work to do, the work ratio is affected. See Table 4.

| D=3, Kg=3 | | | |
|---|---|---|---|
| H | $f$ | Km/Ks | work ratio |
| 40000 | 32699 | 8/20 | 1.001 |
| 30000 | 23822 | 8/20 | 1.016 |
| 20000 | 15039 | 8/20 | 1.116 |
| 15000 | 10645 | 8/20 | 1.277 |
| 15000 | 10598 | 8/16 | 1.021 |

**Table 4**

We found that the amount of sharing (which enables grey cells to be marked without the cost of a search) varied with the depth of the stack: $D=1$ gave an occasional single sharing, whereas $D=2$ gave 18-30 sharings per cycle, this amongst an overflow count of 60-80 cells—a significant result.

For our algorithm to deserve the tag of ‘real-time’ we must be able to predict within small margins exactly how much work is to be charged to GC, and when. It is clear that calls to Stack-Safety GC occur only on every cell allocation. The cost of a call is bounded by

$$\max\ ((\mathtt{Km}\,M + \mathtt{Ks}\,S),\, R)$$

where $M$ and $S$ are the time taken for one iteration of the marker and sweeper respectively, and $R$ is the time taken to mark the roots at the beginning of a new cycle. By aiming for parity in the work ratio, we are reducing any variation in the cost of a call, leading to greater stability. In real-time parlance, there is less jitter. For $R$ to be made roughly equivalent to the cost of marking/sweeping, techniques such as defining a ‘super-root’ from which the entire rootset is accessible, may be required for particular machines.

In conclusion, we have demonstrated a memory management scheme for functional languages, suitable for the real-time domain, and in particular, for embedded systems with small memory and a single processor. We must yet improve its low-level efficiency and evaluate it under a wider range of applications.

## Acknowledgements

Thanks to Niklas Röjemo and Thomas Hallgren for helpful comments on the Stack-Safety algorithm, and to Christian Queinnec for further information on his Mark-During-Sweep algorithm.

# References


[Abdullahi et.al. 92]**Collection schemes for distributed garbage**, Saleh E Abdullahi, Eliot E Miranda, Graem A Ringwood, in *Proceedings International Workshop on Memory Management*, St. Malo, France, Springer-Verlag, LNCS **637**, pp.43-81 (September 1992)

[Appel et.al. 88]**Real-time concurrent garbage collection on stock multiprocessors**, Andrew A Appel, John R Ellis, Kai Li, in *Proceedings ACM Sigplan Symposium on Programming Language Design and Implementation*, Atlanta, Georgia, pp.11-20 (June 1988)

[Baker78]**List processing in real-time on a serial computer**, Henry G Baker, Communications of the ACM **21***(4)*, pp.280-294 (April 1978)

[Baker92]**The treadmill: real-time garbage collection without motion sickness**, Henry G Baker, ACM Sigplan Notices **27***(3)*, pp.66-70 (March 1992)

[Ben-Ari84]**Algorithms for on-the-fly garbage collection**, Mordechai Ben-Ari, ACM Transactions on Programming Languages and Systems **6***(3)*, pp.333-344 (July 1984)

[Brooks84]**Trading data space for reduced time and code space in real-time garbage collection on stock hardware**, Rodney A Brooks, in *Proceedings ACM Symposium on Lisp and Functional Programming*, Austin, Texas, pp.256-262 (August 1984)

[Jones91]**Gofer functional programming environment version 2.21**, Mark P Jones, Yale University (1991)

[Lieberman & Hewitt 83]**A real-time garbage collector based on the lifetimes of objects**, Henry Lieberman, Carl Hewitt, Communications of the ACM **26***(6)*, pp.419-429 (June 1983)

[McCarthy60]**Recursive functions of symbolic expressions and their computation by machine.**, John McCarthy, Communications of the ACM **3***(4)*, pp.184-195 (April 1960)

[Queinnec et.al. 89]**Mark DURING sweep rather than mark THEN sweep**, Christian Qeinnec, Barbara Beaudoing, Jean-Pierre Queille, in *Proceedings PARLE '89*, Springer-Verlag, LNCS **365**, pp.224-237 (1989)

[Wilson92]**Uniprocessor garbage collection techniques**, Paul R Wilson, in *Proceedings International Workshop on Memory Management*, St. Malo, France, Springer-Verlag, LNCS **637**, pp.1-42 (September 1992)

[Yuasa90]**Real-time garbage collection on general-purpose machines**, Taiichi Yuasa, Journal of Systems and Software **11**, pp.181-198, Elsevier (1990)