Structured Programming (1990) 11: 117-127

Structured Programming

A Comparison of Three Garbage Collection Algorithms[1]

Pieter H. Hartel

Computer Systems Department, University of Amsterdam[2], Kruislaan 409, 1098 SJ Amsterdam, The Netherlands

Abstract. The running cost of garbage collection is studied as a function of the amount of available store. A performance model originally proposed by Hoare is modified to support experiments with three garbage collection methods: reference count, mark/scan and two-space copy. By also taking the boundary effects into account, we show that adding more store to a list processing system with a non-reference counting garbage collector does not always make it run faster. We present an explanation for this anomaly in the behaviour of garbage collection algorithms and discuss some of its consequences.

Key Words: garbage collection, performance modelling, combinator graph reduction, anomalous behaviour, subinstruction-level timing, SASL

1. Introduction

The implementation of a list processing system requires a garbage collector if storage space is limited and the removal of the last reference to an object is implicit. The major problem is to control the cost at which the act of removing the last reference to an object can be detected. Many algorithms have been proposed to implement garbage collection [5]. Each algorithm has specific advantages and disadvantages. For instance, reclamation of the space occupied by cyclic structures is more difficult with reference counting algorithms than with mark/scan or two-space copying algorithms [2, 7]. The space requirements and running costs of the algorithms are also vastly different. Based on certain assumptions, mainly about the structure of the graphs involved, theoretical work provides insight in the asymptotic complexity of garbage collection algorithms. A systems architect wishing to select an appropriate garbage collection algorithm for a particular list processing system first chooses the algorithms with the best asymptotic complexity that fit the boundary conditions the list processing system imposes. The practical implementation of these algorithms (all with the same asymptotic complexity) may turn out to be rather different because of the constants that have been ignored in the asymptotic analysis. However, it is precisely this difference that allows the architect to decide which of these algorithms to use. Therefore some authors have made a more practical analysis of garbage collection, usually based on an implementation in a (hypothetical) language. This makes it possible to base more detailed comparisons on counting memory references, executed instructions, etc. [1, 14]. In this paper the analysis and measurement of the cost of garbage collection is based on subinstruction-level timing. This provides the most detailed means of comparing (implementations of) algorithms. This approach was chosen to test the frequently made assertion that counting instructions at a particular level leads to the same conclusions as measuring execution time. Our analysis is also different from most other studies because the structure of the graphs on which we perform garbage collection is specific to applications of a list processing system. This allows us to derive realistic performance data on garbage collection algorithms.

The choice for using real graphs directs us towards using a generator of real graphs and garbage. We have chosen to use as a list processing system an implementation of the functional programming language SASL [19], together with a benchmark of application programs written in SASL [10]. In selecting the application programs we have tried to avoid the unusual ones and thus create a sample of application programs which represents average behaviour. The benchmark set contains four small programs and four medium-size programs. All are run on small input data

1 This work is supported by the Dutch Ministry of Science and Education, dienst Wetenschapsbeleid.

2 The authors present address is the Department of Electronics and Computer Science, University of Southampton, England.

sets: "*fib* 7" prints the seventh Fibonacci number, "*quicksort*" (9, 0, 1, 8, 4, 8) sorts a list of 6 numbers, "*hamming* 100" calculates in ascending order the first 100 natural numbers whose prime factors are 2, 3 and 5 [21], "*paraff* 5" enumerates in order of increasing size the first five paraffin molecules [20], "*wave* 5" predicts the tides in a rectangular estuary of the North Sea over a period of 5 × 20 minutes [22], "*em script*" runs a simple script through a functional implementation of the UNIX text editor [15], "*lambda (S K K)*" evaluates to *I* on an implementation of the λ–K calculus [8] and "*yacc yacc-grammar*" generates a parser for yacc input in SASL [17].

The implementation of SASL is based on Turner's method of graph reduction, which uses a fixed set of combinators [18]. As an implementation technique of a functional programming language with normal order semantics, this method has been surpassed by more efficient methods such as the G-machine [13], CLEAN [3], and the TIM machine [6]. These methods in essence derive their improved efficiency from avoiding using the graph whenever possible, which implies avoiding garbage collection, and is thus contrary to our intention to study garbage collection. There are even applications that do not need the graph at all. Admittedly such programs do not represent the most interesting list processing applications, but unfortunately they are often used in comparing the performance of implementations of functional languages. Because Turner's reduction method places heavy demands on the garbage collector, we consider it suitable as a generator of graphs and garbage. The method has the advantage that it is simpler to reason about than other methods. We know, for instance, that a reduction step never requires more than three new nodes. This is important to know, because during a reduction step the graph need not be connected. This is the case if a new internal node is added to the graph because that requires updating at least two pointers: one to the new node and one emanating from it. Garbage collection algorithms would cause a disaster if applied to a graph in disconnected state. With program-derived super combinators a threshold on the required number of available nodes is application-dependent and therefore more difficult to use than with Turner's method.

2. Selection of Garbage Collection Algorithms

We have chosen three garbage collection algorithms that seemed most promising with respect to their run time efficiency. This means that we are prepared to expend a number of extra bits or bytes per object to support rapid storage allocation and reclamation. The mutator that we use is always the implementation of SASL based on fixed combinator graph reduction. It is assumed that both the mutator and the collector run on the same processing element, which consists of a sequential processor and a fair amount of local store. The mutator and the collector run in an interleaved fashion. Normally the mutator executes, but whenever necessary, the mutator calls the collector, which, after completing its task, returns control to the mutator. Hence we are not concerned with issues of parallel garbage collection. Another problem that is ignored here is the use of virtual memory.

Two more technical problems for which many intricate solutions have been proposed are resolved in favour of simplicity. Firstly, all objects are assumed to occupy the same amount of space (we make no distinction between an object and the space it occupies). Secondly, reclamation of the space occupied by redundant cyclic structures is assumed to be rarely necessary. These restrictions pose no problems to the SASL implementation or the benchmark programs. An interior node of the graph that is processed by the SASL interpreter represents the application of a function to a single argument. Leaf nodes represent constants such as (floating point) numbers and combinators. A small object that can accommodate two pointers or a floating point datum suffices. The second restriction is justified because we have observed that the vast majority of the cycles are due to the use of recursive functions and not cyclic data structures [10]. The graphical structure that represents the function definitions of a program is rarely a good candidate for garbage collection. Though some of the function definitions may not be used any more after the application has progressed to a certain point, it appears that in particular because of lazy evaluation few functions definitions can be garbage collected. Furthermore the implementation of a functional language is often embedded in a conversational environment that requires the function definitions to be protected from the garbage collector. Instead it is the rapidly expanding structure that represents applications of the user-defined functions to their arguments, which needs to be controlled by the garbage collector. Cycles in the structure that represents the function definitions can be detected at low cost [9]. The remaining cycles are in the data structures. Vree has shown that such cycles can often be avoided by suitable program transformation [23]. In general we cannot avoid having to reclaim space occupied by circular structures. Rather than to ignore reference counting just because its simplest and most efficient form cannot cope with cycles, we include the method in our experiments with a special provision discussed below.

There are in principle two ways to identify redundant objects. The first method, reference counting, keeps a count of the number of references to an object. When

the count drops to zero, the object can be recuperated. This method has a strong intuitive appeal because it "never leaves to tomorrow what can be done today". The disadvantage of reference counting lies in the additional information that must be maintained for each creation or destruction of a reference. The particular algorithm in this category that we choose to study is referred to as *RefCount*. It uses a reference count field that occupies the same number of bytes as a pointer field, such that a reference count cannot overflow. *RefCount* always reclaims the maximum number of objects.

We use the simplest and fastest form of reference counting, i.e., the algorithm that cannot reclaim the space occupied by circular structures. Should the reference counting storage allocator run out of free cells because all remaining garbage cells are tied up with hitherto unrecoverable cyclic structures, then we resort to one of the other garbage collection methods. Our hope is that this will rarely be necessary, and indeed in the experiments to be reported in Section 5, we found that our reference counting storage manager never ran out of space. As a result our analysis with respect to reference counting storage management should be interpreted as a lower bound on its performance.

The second method to identify redundant objects is by elimination. The objects that are still in use are marked, such that unmarked objects are deemed to be garbage. We selected two algorithms from this category. A mark/scan algorithm first marks all the used objects, then makes a scan over the entire store, picking up all redundant objects. The particular algorithm we chose, referred to as *MarkScan*, delays the scan until objects are actually required [12]. This obviates the need for building a free list and taking it apart again, which would increase running time. The third algorithm (*Queue*) copies the objects that are still in use from the current partition, which will be called to a fresh area of store (called *to-space*). The roles of *from-space* and *to-space* are interchanged after each *Queue* garbage collect [4].

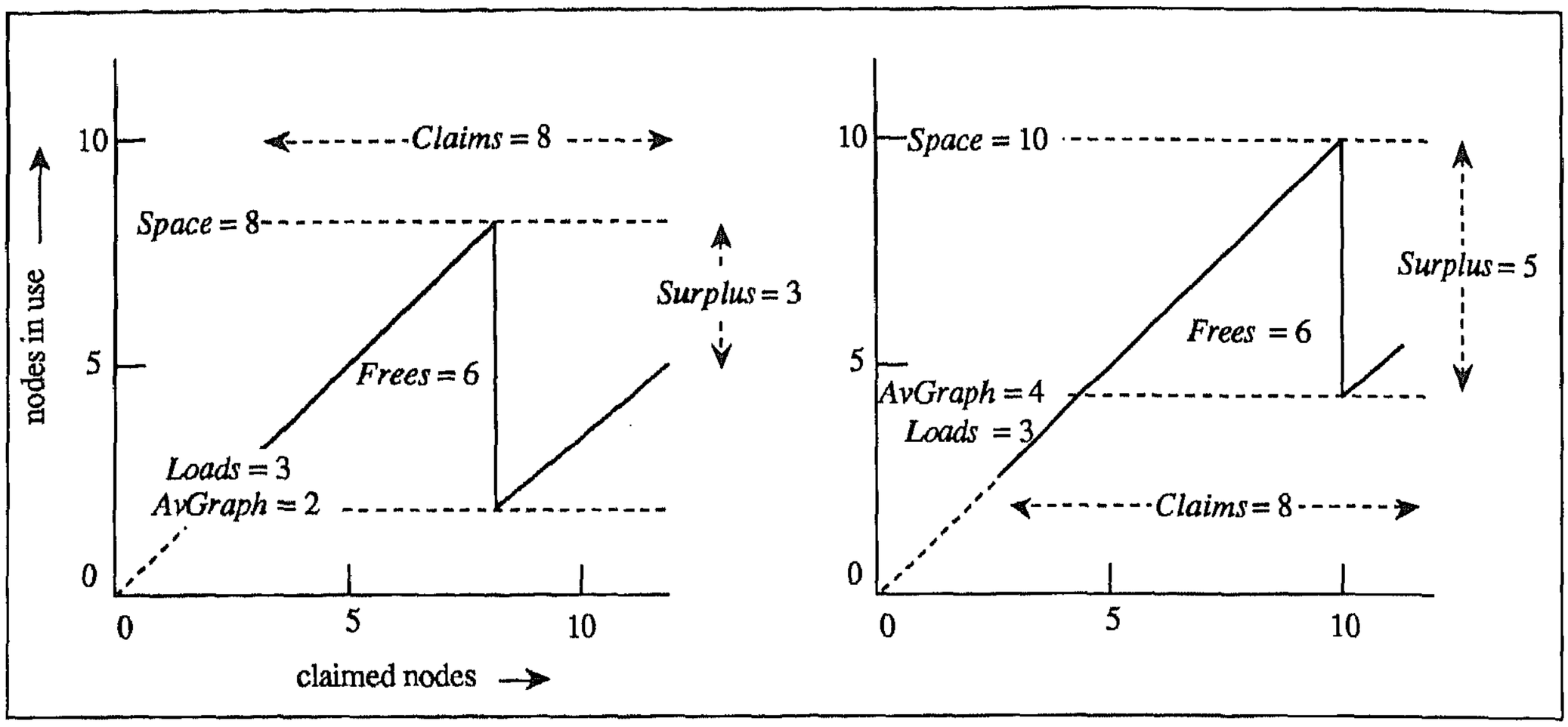
3. System Model

We will develop an analytic model to capture the essence of the three different garbage collection methods in a number of parameters. The parameters will be chosen such that the three algorithms can be compared. The behaviour of the algorithms is primarily determined by the size of the store and the number of objects that an application of the list processing system requires to execute. Let the size of the store be *Space* objects and let *Claims* represent the number of objects claimed during the execution of a particular application program. We do not consider loading the application into the store as part of the execution because the loading process does not require services of the garbage collector. The number of objects claimed to load an application is represented by the parameter *Loads*. The total number of objects ever required by an application program is therefore *Loads* + *Claims*, but the cost analysis pertains to the *Claims* objects claimed during execution.

If based on reference counting, the collector must be called for each reference that is deleted, but the *MarkScan* and *Queue* algorithms require the services of the collector only when a certain threshold on the use of the store is reached. As a consequence the behaviour of these systems is rather different from the reference counting scheme. A reference counting algorithm makes objects available to be claimed again at the moment they become unreachable. The cost of reference counting is therefore independent of *Space*. We will define the cost of reference counting garbage collection as the total number of ticks spent in the allocation of new objects and collection of redundant objects divided by *Claims*. We use a "tick" as the unit cost of work. When the implementations of the garbage collection algorithms are discussed, we will introduce a more concrete measure to use instead of the tick.

The cost of storage management with *MarkScan* and *Queue* does depend on *Space*, hence the analysis will have to be more involved. The behaviour of *MarkScan* and *Queue* can be shown in a "storage profile" that represents the number of objects that are in use as a function of the number of claims. We call an object "in use" if it is either part of the graph or when it has become garbage without having been discovered as such. Objects that are in use can therefore not be claimed until the next time garbage is collected. Figure 1 shows the storage profiles of a program with two different values of *Space* (8 and 10 respectively). After loading the program into store, which requires *Loads* = 3 objects, the number of claims rises until the store is full. As a result of garbage collection a number of objects are recuperated. After garbage collection all objects that are still in use are part of the graph. We use *AvGraph* to represent the average size of the graph immediately after a *MarkScan* or *Queue* garbage collection. The number of garbage collections is *Collects*, and *Frees* indicates the total number of objects recuperated by *MarkScan* or *Queue* garbage collections. In Figure 1a *AvGraph* = 2 and in 1b *AvGraph* = 4. In both cases *Frees* = 6 and *Collects* = 1.

When an application terminates, a number of objects remain available to be claimed that are not needed any more. Let *Surplus* represent this number of objects. The number of objects that are in use when the application terminates is thus *Space* - *Surplus*. In Figure 1 the values of *Surplus* are 3 and 5. In discovering the *Surplus* objects a number of ticks are spent by the garbage collector that must be taken into account as part



(a) *Space* = 8;
cost = 8 × *AllocCost* + 2 × *MarkCost* "ticks"

(b) *Space* =10;
cost = 8 × *AllocCost* + 4 × *MarkCost* "ticks"

Figure 1. Two storage profiles of the same application with the *Queue* garbage collector.

of the running costs of garbage collection, in particular if *Claims* + *Loads* is a little larger than *Space*. Hoare's analysis [11], upon which the current one is based, ignores these boundary effects, since he assumes that *Space* is small in comparison to *Claims*. (The second major difference with Hoare's analysis is that we do not account for the size of the store as a cost factor.) In the next section we will show to what extent boundary effects influence the cost of garbage collection.

The parameters that we have introduced so far characterise the storage profile of an application. They represent a number of objects in a particular state. What we need to add is a number of cost factors that can be multiplied by one of the parameters to calculate the total cost involved in processing objects in that particular state. The most interesting factor is *MarkCost*: the average number of ticks spent to traverse an object during the mark phase of *MarkScan* or the copy phase of *Queue* garbage collection. Included in the *MarkCost* are the ticks spent to discover whether the object being traversed is a leaf or an interior node and the ticks necessary to mark the object as being traversed (*MarkScan*) or to mark it and copy the object to *to-space* (*Queue*). The value of *MarkCost* is thus sensitive to the composition of the graph: the more objects are shared, the higher *MarkCost* will be. The second factor, *AllocCost*, represents the number of ticks required to allocate an object. Both for *MarkScan* and *Queue* garbage collection, *AllocCost* is a constant because it does not depend on the structure of the graph that garbage collection operates on. The *AllocCost* for a particular implementation of a garbage collection algorithm can be determined statically from the program text; the *MarkCost* must be measured. The behaviour of the *Queue* algorithm can now be fully described. For the *MarkScan* algorithm we need one more parameter and associated cost factor, because when claiming an object, we must skip (and unmark) the marked objects that we find on our way to an unmarked object. The parameter *Skips* represents the total number of marked objects that must be skipped and the cost of skipping a single object is represented by *SkipCost*. Like *AllocCost*, the *SkipCost* can be determined statically from the program text of the implementation of the garbage collection algorithm. In general *Skips* cannot be determined from the other parameters because its value depends on the way marked objects are distributed over the store. The only case where *Skips* can be computed is when *Surplus* = 0 (then *Skips* = *Collects* × *AvGraph*).

3.1 Anomalous Behaviour in Garbage Collection

Garbage collection performance at values of *Space* ≤ *Claims* ≤ 2 × *Space* is of interest because it represents the performance of a system with garbage collection performed under favourable circumstances. The case where *Claims* < *Space* is even more favourable, but since it does not require garbage collection, we do not consider it here. When *Claims* is just a little larger than *Space*, the garbage collector has to perform much work to recuperate redundant objects, while few of these will actually be (re)allocated. The average cost per object will therefore be relatively high. One would expect this cost

to drop as the size of the store is enlarged. We will show that this is not necessarily the case. Figure 1 shows two storage profiles of the same application with the *Queue* garbage collector. In both cases *Loads* and *Claims* have the same values, but the sizes of the stores are different. Since in both examples *Collects* = 1, we find for the total cost of allocation and garbage collection $Claims \times AllocCost + AvGraph \times MarkCost$. As we can see from the diagram, the value of *AvGraph* rises along with that of *Space* from 2 to 4. We also know that *AllocCost* is a constant of the implementation of the garbage collection algorithm and that *Claims* is a constant determined by the application. The *MarkCost* values may be different because they are sensitive to the structure of the graph being copied. However, a difference of a factor two is virtually impossible because that would imply that in the graph traversed in Figure 1a many more nodes would be shared than in Figure 1b. In practice we have observed that on average the percentage of shared nodes is less than 10% [10]. The increase of *AvGraph* cannot be compensated by a decrease of *MarkCost*, hence the total cost of allocation and collection is higher in Figure 1b than it is in Figure 1a. We must conclude that adding more store does not always make a list processing system faster. The real problem is that there is no easy way to choose a favourable moment for garbage collection. The only way to know when a graph is small is by performing garbage collection.

The effects of the "garbage collection anomaly" are not restricted to the case where *Collects* = 1, but the effect at higher *Collects* values has a tendency to average out. In the experiments discussed in Section 5 we will show the effect of garbage collection anomaly on a real application.

3.2 Analysis of *Queue* and *MarkScan* Model Parameters

Not all the parameters for the *Queue* and *MarkScan* algorithms are independent. We will show that *Collects* and *Frees* can be eliminated. The first observation we can make is that an object must either be in existence when the application is started (*Loads*), or claimed (*Claims*), or still available upon termination (*Surplus*). Their sum must be equal to the total heap space (*Space*) plus the number of collected objects (*Frees*):

$$Space + Frees = Loads + Claims + Surplus \qquad (1)$$

The second observation is that garbage collection cannot alter the number of objects in the system. The average number of collected objects is $Space - AvGraph$. Therefore the total number of collected objects is:

$$Collects\ (Space - AvGraph) = Frees \qquad (2)$$

Combination of (1) and (2) yields:

$$Collects = \frac{Loads + Claims + Surplus - Space}{Space - AvGraph} \geq 0 \qquad (3)$$

The total cost of marking or copying the objects that are part of the graph is $Collects \times AvGraph \times MarkCost$. For the *Queue* algorithm, the total cost of claiming objects is $Claims \times AllocCost$. The average number of ticks necessary to claim an object for the *Queue* algorithm is therefore:

$$TicksPerClaim_{Queue} = \frac{Claims \times AllocCost + Collects \times AvGraph \times MarkCost}{Claims}$$

The *MarkScan* case is slightly complicated by the delay of the scan. While the cost of claiming an object is a constant for *Queue*, *MarkScan*, when claiming an object, must skip the marked objects on its way to an unmarked object. This requires a total of $Skips \times SkipCost$ ticks. For the *MarkScan* algorithm we find as the average cost per object:

$$TicksPerClaim = \frac{Claims \times AllocCost + Collects \times AvGraph \times MarkCost + Skips \times SkipCost}{Claims} \qquad (4)$$

To facilitate references to the formulae we will use (4) also for the *Queue* algorithm, but with $SkipCost_{Queue} \equiv 0$. To compare the costs incurred by the different application programs, we will express the cost of allocation as a function of the store capacity, which is defined as the number of times the average graph fits into the store: $AvCapacity = Space / AvGraph$. With the value of *Collects* found in (3) we obtain from (4):

$$TicksPerClaim = AllocCost + \frac{Loads + Claims + Surplus - AvCapacity \times AvGraph}{(AvCapacity - 1)\,Claims} MarkCost + \frac{Skips}{Claims} SkipCost \qquad (5)$$

This formula states that there are three components to the cost of claiming an object. The first is the cost of allocating it (e.g., to remove it from the free list). The second and third components represent a fraction of the garbage collection costs attributed to the object. The fraction multiplied by the *MarkCost* represents the effort of the garbage collector to mark the objects that are still in use. The fraction multiplied by *SkipCost* accounts for the cost of skipping objects that are marked by *MarkScan*. Equation 5 will be used to calculate *MarkCost* from the other quantities, which we will determine in a series of experiments to be described below.

Algorithm	Extra field(s) per object	maximum space in bytes for			MC68010 cycles	
		Heap	Stack	Total	*AllocCost*	*SkipCost*
RefCount	reference count	12 × *Space*	4 × *Space*	16 × *Space*	–	–
Queue		2 × 8 × *Space*	0	16 × *Space*	22	–
MarkScan	visited flag	9 × *Space*	4 × *Space*	13 × *Space*	36	50

Table 1. Static characteristics of the algorithms on the MC68010 processor.

4. Implementation of Garbage Collection Algorithms

The SASL interpreter, which drives the storage allocation and reclamation, operates on a rooted directed cyclic binary graph. The graph is transformed from its initial state to the final state in small "reduction" steps. On average a step claims one node and alters a few edges. No reduction step requires more than three new nodes. With *Queue* or *MarkScan* garbage collection we use this property to check at the beginning of each step whether enough objects are still available. If not, the garbage collector is started with the root of the graph as the first object to be marked or copied. At that point, the graph is connected, which is usually not the case while a reduction step is in progress. Because of its close connection with reduction, we have decided not to account for the test on the available space as part of the cost of allocation and collection.

Apart from the pointer to the root, the SASL interpreter maintains a stack of pointers into the graph in its "left ancestors" stack. The average depth of this stack is of the order $\sqrt{AvGraph}$ [10]. The contents of the left ancestors stack have to be updated by the *Queue* algorithm and the cost of these updates is accounted for. The left ancestors stack is of no concern to the *MarkScan* and *RefCount* algorithms.

The representation of a node includes a tag field that allows the reducer and the collector to perform case analysis on the type of the node. The collector must be able to distinguish leaf nodes from interior nodes, but the reducer needs more information. Since our interest is in the cost of garbage collection rather than the cost of reduction, we have decided to account for reading and writing pointer fields, not tag fields. At all times we assume the pointer to contain valid information (i.e., a pointer value or NIL). This may appear to give the *Queue* algorithm an advantage, since insufficient data is copied, but as we shall see in Section 5.1, the difference may safely be ignored. All algorithms would suffer in roughly the same way if discrimination on the tag value had been incorporated in the performance measurements, but there are also methods that do not require tags to distinguish between leaf and interior nodes. For instance leaf nodes could be kept in a separate region of store such that the pointer values carry the type information. Our objects can thus be considered as minimal objects.

Table 1 shows the composition of the objects that are processed by the algorithms (as far as storage allocation and reclamation is concerned). A pointer or a reference count field requires 4 bytes and a flag field requires 1 byte of storage. Mappings that are thriftier of space are conceivable, but on most architectures this would slow down execution.

The garbage collection algorithms were implemented in assembly language for a Motorola 68010 processor. In the measurements a processor clock cycle (≈ 0.1 μ sec) is taken as the unit of time (*tick*). Recursion is implemented via explicit use of a stack of pointers to the objects that remain to be processed. A garbage collector operates in a tight loop consisting of move, clear, add, increment, decrement, test and branch instructions. Often used data is kept in 8 of the processor's registers. None of the algorithms could be improved by using a few more registers. The flow of control has been structured such that at each decision point the condition that occurs the least frequently causes the branch to be taken. No procedure calls are used; the code to claim a new node is inserted as in-line code where needed. Since most information is kept in registers, care was taken to use the settings of the condition codes generated by "move to register" instructions rather than test instructions to steer the branch instructions. The fields of an object are accessed by using the "address register indirect with displacement" addressing mode and the stack is manipulated with "auto increment" and "auto decrement" addressing modes. The configuration of a processing element is assumed to be such that the instructions and the data in the stack and in the heap can each be accessed at uniform cost, so we assume no memory management unit to be interposed between the processor and the store. The effects of caches on the performance of the algorithms are included by simulation in our experiments.

The implementations of the algorithms are close to what may be considered "standard" [5], such that we need not provide the listings here. The information that has been determined from the program text of the implementations are the *AllocCost* and *SkipCost*. Their values expressed in MC68010 processor cycles are shown in the last two columns of Table 1. We use a memory that does not require "wait states", hence one memory reference requires four processor cycles [16].

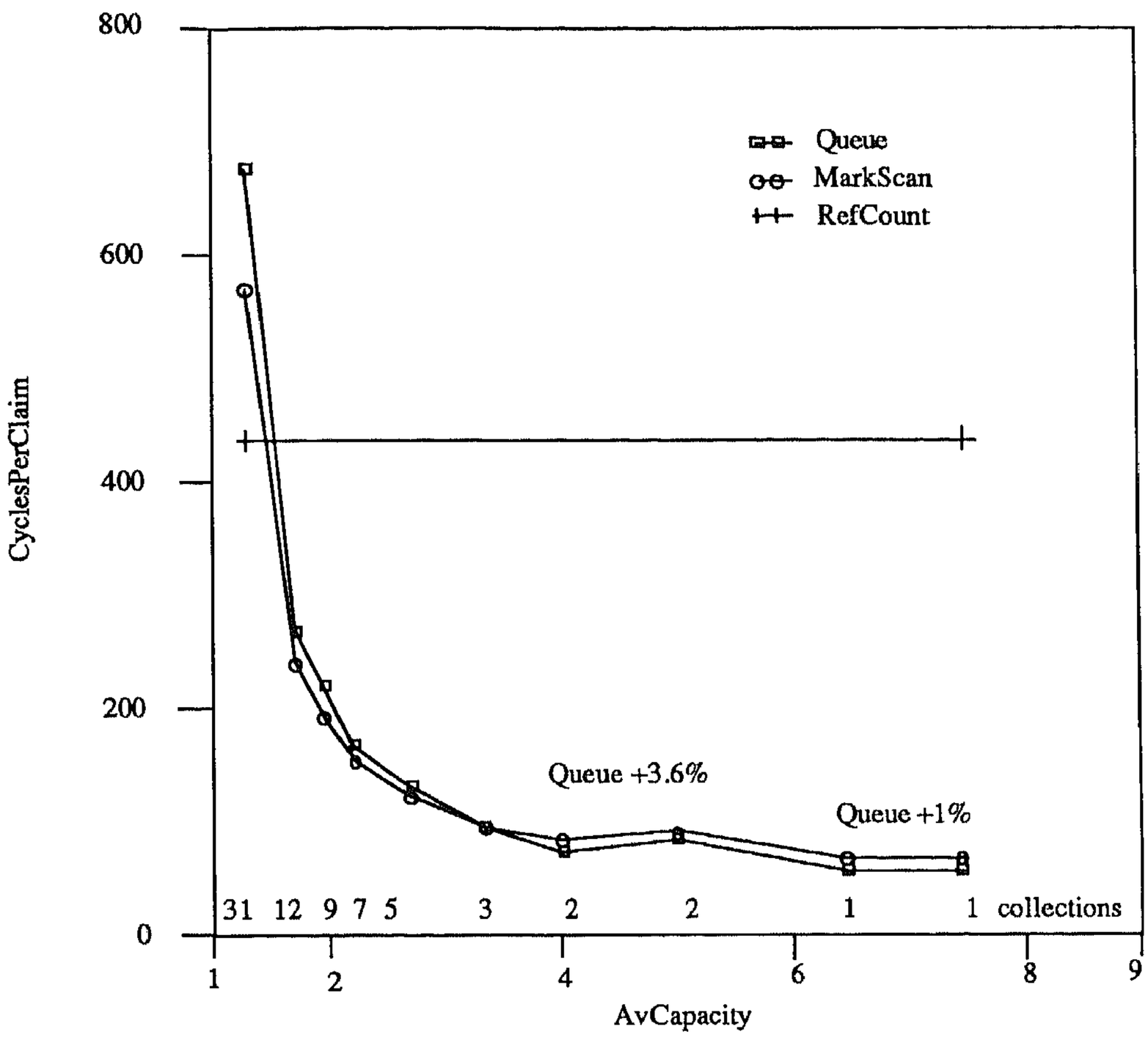


Figure 2. CyclesPerClaim.

5. Experiments

Each application from the SASL benchmark set was run once with *RefCount* and about ten times with each of the other garbage collection algorithms. *Space* was varied from run to run in order to measure the dependency of the *CyclesPerClaim* on *AvCapacity* with the *MarkScan* and *Queue* garbage collectors. The values of *Space* close to *AvGraph* cause many garbage collections, whereas large values of *AvCapacity* cause few or no garbage collections. Equation 5 states that *CyclesPerClaim* is roughly proportional to the reciprocal of *AvCapacity*. This is confirmed by the measurements. As an example Figure 2 shows the experimental data that have been gathered with the "*lambda*" program. The cost of *RefCount* does not depend on *AvCapacity* and is about an order of magnitude higher than the *AllocCost* values. Hence for both non-reference counting algorithms, there is a crossover point where *RefCount* has the same cost as non-reference counting. *RefCount* is cheaper below this point.

From the rise of the *CyclesPerClaim* between *AvCapacity* = 4 and 5 and again between *AvCapacity* = 6.3 and 7.4 we can see that in a medium-size application the performance of the *Queue* algorithm suffers somewhat from the garbage collection anomaly. The increase in the *CyclesPerClaim* are 3.6% and 1% respectively. The cause of the anomaly lies in the growth of the graph that is copied. For instance, at *AvCapacity* = 6.3, the size of the copied graph is *AvGraph* = 7866 objects, whereas it has 8080 objects at *AvCapacity* = 7.4. (In both these cases *Collects* = 1). With the *CyclesPerClaim* we have a measure that can be used to compare the performance of garbage collection algorithms at specific values of *AvCapacity*. The *Queue* and *MarkScan* algorithms are better compared using the *MarkCost*, *AllocCost* and *SkipCost*, because these do not depend on *AvCapacity*. The values of *Claims*, *Loads*, *Surplus*, *AvGraph* and *Skips* were recorded with the *CyclesPerClaim*, such that the coefficient *MarkCost* could be calculated from (5) using the *AllocCost* and *SkipCost* as shown in Table 1. Theoretically the values of *MarkCost* should be invariant to the choice of *AvCapacity*. However, the structure and composition of the graphs is not constant during a run and the garbage collector is activated at different stages of the computation depending on the value of *AvCapacity*. For example, the first garbage

parameter	fib	quicksort	hamming	paraff	wave	em	lambda	yacc
Claims	270	1035	16639	25606	389082	314520	68406	644682
Loads	45	134	141	1021	2113	4103	5241	14949
$\overline{AvGraph}$	62	202	593	1958	20191	14219	7650	30007
RefCount								
CyclesPerClaim	392	365	374	394	389	382	431	412
Queue				*AllocCost* = 22				
MarkCost	202	193	189	192	200	188	189	188
standard deviation	3	1	0.4	0.8	0.1	0.1	0.6	0.04
MarkScan			*AllocCost* = 36		*SkipCost* = 50			
MarkCost	115	113	105	105	106	98	103	100
standard deviation	1	1	0.3	0.3	0.1	0.03	0.3	0.01

Table 2. The benchmark applications on an MC68010 using Equation 5.
Claims, *Loads* and $\overline{AvGraph}$ in objects; *MarkCost*, *AllocCost* and *SkipCost* in MC68010 processor cycles.

collection occurs at different instants with respect to the status of the application for the different values of *AvCapacity*. Subsequent garbage collections are even more incomparable in this respect. Table 2 shows that the results are satisfactory, since the standard deviation of *MarkCost* is generally less than 1% of the mean. The small programs *fib* and *quicksort* are less reliable predictors for the value of *MarkCost*, because small programs exhibit a larger variation in the percentage of shared nodes. The average values of *AvGraph* that were measured are presented in the row $\overline{AvGraph}$ to give an idea of the average size of the graphs.

The *MarkCost* values for *Queue* give a lower bound for the case where more than a minimal object is copied (see Section 4). On processors that provide an efficient block move instruction, only a few more cycles would be required to copy an extra word of information per object. With the MC68010, for each extra word (4 bytes) to be copied, *MarkCost* must by incremented by 8.

A noticable difference between our results and those generally accepted is the relative proximity of *MarkCost* and *AllocCost*. This may be just a matter of interpretation, since some authors are vague in this respect. Cohen [5] writes "it is not unreasonable to assume that *AllocCost* is considerably smaller than *MarkCost*". We find that the ratio of *MarkCost* to *AllocCost* does not exceed 10.

5.1 Other Ways of Calculating the Cost of Garbage Collection

One reason to collect detailed statistics is the possibility to verify certain claims that advocates of other analysis methods make about the validity of their methods. Since our information is fine grained, it is possible to derive more coarse grained information from it. We have modelled the boundary effects that are commonly ignored as the parameters *Surplus* and *Skips*. We will show that if an estimate for these is derived from the other parameters, much of the accuracy of the results is lost. *Surplus* represents the number of objects that could have been allocated, but were not needed any more when the application terminates. Hence it must hold that $0 \leq Surplus < Space - AvGraph$. There is no reason why *Surplus* should have a particular value, so we will assume its value to be average:

$$Surplus \approx \frac{Space - AvGraph}{2} = \frac{AvCapacity - 1}{2} AvGraph, \quad Collects \gg 0 \qquad (6)$$

Skips represents the total number of objects that are skipped during the scan phase of *MarkScan*. Let us assume, that when the application terminates, the remaining *Surplus* free objects are distributed uniformly over the unscanned storage space. Then apart from *Surplus* objects that can be claimed, there are *Surplus* objects that must be skipped during the scan. With (3) we find:

$$Skips \approx \left(Collects - \frac{Surplus}{Space - AvGraph}\right) AvGraph = \frac{Loads + Claims - AvCapacity \times AvGraph}{AvCapacity - 1}, \quad Collects \gg 0 \qquad (7)$$

Substitution of both approximations (7) and (6) in (5) yields:

$$TicksPerClaim \approx AllocCost + \frac{Loads + Claims - 1/2(AvCapacity + 1)AvGraph}{(AvCapacity - 1)Claims} MarkCost + \frac{Loads + Claims - AvCapacity \times AvGraph}{(AvCapacity - 1)Claims} SkipCost, \quad Collects \gg 0 \qquad (8)$$

	eq (8)	eq (5)						
parameter	MC68010 approx.	MC68010 exact	free stack	free heap	free code	free code&stack	uniform	Modula-2
RefCount								
CyclesPerClaim	455	455	455	357	98	98	81.5	65.3
Queue								
MarkCost	193	189	189	155	33.4	33	30	24.3
standard deviation	28	0.6	0.6	0.6	0.1	0.1	0.2	0.1
AllocCost	22	22	22	22	0	0	2	2
MarkScan								
MarkCost	113	103	101	85	19.5	18	18.5	13.8
standard deviation	24	0.3	0.2	0.2	0.1	0.1	0.1	0.1
AllocCost	36	36	36	32	4	4	5	4
SkipCost	50	50	50	42	8	8	7	5
$\frac{MarkCost_{Queue}}{MarkCost_{MarkScan}}$	1.70	1.83	1.87	1.82	1.71	1.83	1.62	1.76

Table 3. The *lambda* program with various measurement methods.
MarkCost, *AllocCost* and *SkipCost* in MC68010 processor cycles (except the last two columns, see text).

The first column (MC68010 approx.) of Table 3 has been calculated using (8). It shows a larger standard deviation in the averages than the second column, which presents the same results but calculated using the exact equation (5) that takes the boundary effects into account. We must conclude that ignoring boundary effects gives result of significantly lower precision.

In the next four columns of Table 3 the calculations with the exact equation (5) are repeated based on zero access times to various segments of the store. This provides an indication of the benefits that hardware support may bring to garbage collection. The "free stack" column gives the results that would have been obtained with an architecture that supports a large enough high speed scratch pad to contain the entire stack needed by *RefCount* and *MarkScan*. The data indicate that the improvement is insignificant. The fourth column applies to the situation where accessing and updating pointers in the heap is free of charge. Technically, this is a rather unrealistic assumption, but it gives some indication of the fraction of time that is spent on memory references (about 20%). The "free code" column presents the values of *MarkCost* and *AllocCost* that can be obtained on a processor that is equipped with an instruction cache and an instruction pipeline such as the MC68020. The column "free code&stack" assumes the presence of both an instruction cache and a high speed scratch pad for the stack. Actual performance figures would not be as good since, for example, a branch instruction may cause stagnation in the pipeline. We may conclude that with an instruction cache the speed of garbage collection on the MC68010 can be improved by at most a factor of 5.

The last two columns of Table 3 are included to verify the assumption that counting high level language statements leads to the same conclusion as measuring execution time. The data in the column marked "uniform" were obtained by counting the execution of each instruction as one cycle and using the exact equation (5). An average instruction (from the subset that is exercised by the garbage collection algorithms) normally consumes about 6 processor cycles. Hence the column "uniform" gives consistently lower figures than the first two columns. In the last column we report the figures that were obtained with equation (5) by counting each high level language (Modula-2) statement as a single *tick*. Most Modula-2 statements in the implementations of the algorithms correspond to single machine instructions, making extensive use of the features of the processor's instruction set. Some conditionals require two instructions: one to set the condition codes and the other to branch on the appropriate condition.

The last row of Table 3 allows for the various counting methods to be compared. The performance of the *MarkScan* and *Queue* algorithms is best compared using the values of *MarkCost*, because *MarkCost* represents the work performed by the traversal algorithm in the garbage collector. We have chosen to use the *MarkScan* algorithm as a reference point because it has the best performance. From the values obtained for the *MarkCost*-ratio we find that all counting methods that we have used lead to the same result $MarkCost_{Queue} / MarkCost_{MarkScan} = 1.76 \pm 8\%$. The *CyclesPerClaim* ratio of the two algorithms is generally less than 1.76, because $AllocCost_{MarkScan} > AllocCost_{Queue}$ and the

MarkScan algorithm also suffers from the cost of skipping marked objects. At high *AvCapacity* values the programs from our benchmark show a slight advantage for *Queue* (see, for example, Figure 2).

6. Conclusions

Experiments with garbage collection algorithms have been conducted in an unconventional way. The mutator is a fixed combinator graph reducer which is driven by a benchmark of SASL programs. We have thus used an average application to drive a worst case list processing system. As a consequence our selection of three garbage collection algorithms had to cope with the heaviest possible demand of free objects while the structure of the graphs that they had to process was real.

Three garbage collection methods were implemented with the same level of efficiency on a Motorola MC68010 processor. We have measured the performance of reference counting, mark/scan and copying garbage collection at subinstruction level. The performance of reference counting garbage collection was found to be superior to that of the other algorithms if the graph being manipulated fits snugly into the store (*AvCapacity* ≈ 1). The performance of all algorithms is roughly equal when the store provides about twice as much space as the average graph requires. A simple explanation for this effect is that reference counting algorithms visit mostly garbage, whereas the copying and to a lesser extent mark/scan algorithms visit objects that are still in use. Hence the costs should be roughly the same when the the amount of garbage is in equilibrium with the number of reachable objects. The mark/scan algorithm is a little faster than the two-space copy algorithm.

The conclusions that were drawn based on these measurements can also be reached if higher level events than cpu cycles are counted. One way to achieve this is to count machine instructions and memory references. Counting high level language statements requires more care, since statements of an arbitrary complexity are easily constructed. One must write the high level language code with the target processor in mind.

In our performance measurements we have taken all boundary effects into account and show that adding more store to a list processing system with a non-reference counting garbage collector does not always lower the cost of storage allocation and garbage collection. The explanation for this form of anomalous behaviour is that adding more store generally postpones the moment at which the garbage collector is activated. Since the size and composition of the graph to be traversed is not constant, how much time it will cost to traverse the graph entirely depends on the state of the graph. The garbage collection anomaly may also play a role when the number of garbage collections is greater than 1. For instance, we could strive at collecting garbage only when we know that the graph is relatively small. In a previous experiment we found that the size of the graph to some extent reflects the state of progression in the benchmark application [10]. Transitions phases in the application can be identified as sudden changes in the size of the graph. This indicates that benefits may be obtained by introducing explicit commands to start garbage collection at suitable moments.

In the experiments, we have imposed two restrictions on the graphs that require garbage collection; reclamation of cyclic structures is not often necessary and all objects are of the same size. Our conclusions remain valid if we lift the first restriction. The non-reference counting algorithms always properly deal with cycles, hence the costs involved cannot change. The cost of reference counting, which is already the most expensive garbage collection method, will yet increase. If the second restriction is dropped the cost of mark/scan and reference count will increase more than that of the two-space copy method because the latter requires less modification to cope with variable sized cells.

Simulated behaviour of garbage collection algorithms shows that no significant performance improvement is obtained if the stack required by the implementations is kept in an area of high speed memory such as a cache. Up to fivefold improvement in speed is possible if the instructions are kept in a high speed area of memory.

Acknowledgements: Discussions with Arthur Veen, Wim Vree and Betsy Pepels provided the stimulus to explore this area of research. Betsy made the suggestion to use Cheney's algorithm. Arthur, Wim and Koen Langendoen made valuable comments on draft versions of the paper.

References

1. Baer JL, Fries M (August 1977) On the Efficiency of Some List Marking Algorithms. In: Information Processing 77, Ed. Gilchrist B, North Holland, Toronto, Canada, 751–756
2. Brownbridge DR (September 1985) Cyclic Reference Counting for Combinator Machines. In: Second Conference on Functional Programming Languages and Computer Architecture. LNCS 201, Ed. Jouannaud JP, Springer Verlag, Nancy, France, 273–288
3. Brus TH et al. (September 1987) CLEAN: A Language for Functional Graph Rewriting. In: Third Conference on Functional Programming Languages and Computer Architecture, LNCS 274, Ed. Kahn G, Springer Verlag, Portland, Oregon, 364–384
4. Cheney CJ (November 1970) A Non-Recursive List Compacting Algorithm. CACM **13**(11), 677–678

5. Cohen J (September 1981) Garbage Collection of Linked Structures. Computing Surveys **13**(3), 341–367
6. Fairbairn J, Wray S (September 1987) Tim: A Simple Lazy Abstract Machine to Execute Supercombinators. In: Third Conference on Functional Programming Languages and Computer Architecture, LNCS 274, Ed. Kahn G, Springer Verlag, Portland, Oregon, 34–45
7. Friedman DP, Wise DS (January 1979) Reference Counting Can Manage the Circular Environments of Mutual Recursion. Information Processing Letters **8**(1), 41–45
8. Glaser H et al. (1984) Principles of Functional Programming. Prentice Hall, Englewood Cliffs, New Jersey
9. Hartel PH (October 1988) Performance Analysis of Storage Management in Combinator Graph Reduction. Department of Computer Systems, University of Amsterdam, PH.D. Thesis
10. Hartel PH, Veen AH (March 1988) Statistics on Graph Reduction of SASL Programs. Software-Practice and Experience **18**(3), 239–253
11. Hoare CAR (1974) Optimization of Store Use for Garbage Collection. Information Processing Letters **2**(1), 165–166
12. Hughes RJM (November 1982) A Semi-Incremental Garbage Collection Algorithms. Software–Practice and Experience **12**(11), 1081–1082
13. Johnsson T (June 1984) Efficient Compilation of Lazy Evaluation, Sigplan Notices **19**(6), 58–69
14. Knuth DE (1973) The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Addison Wesley, Reading, Massachusetts, 2nd Ed.
15. Koopman PWM (September 1987) Interactive Programs in a Functional Language: A Functional Implementation of an Editor. Software–Practice and Experience **17**(9), 609–622
16. Motorola Inc. (1986) MC68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual. Prentice Hall, Englewood Cliffs, New Jersey, 5th Ed.
17. Peyton Jones SL (August 1985) Yacc in SASL—an Exercise in Functional Programming. Software–Practice and Experience **15**(8), 807–820
18. Turner DA (January 1979) A New Implementation Technique for Applicative Languages. Software–Practice and Experience **9**(1), 31–49
19. Turner DA (August 1979) SASL Language Manual. Technical Report, Computing Laboratory, University of Kent at Canterbury
20. Turner DA (October 1981) The Semantic Elegance of Applicative Languages. In: Conference on Functional Progamming Languages and Computer Architecture, Ed. Arvind, ACM, Portsmouth, New Hampshire, 85–92
21. Turner DA (February 1984) Functional Programs as Executable Specifications. In: Mathematical Logic and Programming Languages, Ed. Hoare CAR, Shepherdson JC, Prentice Hall, London, 29–54
22. Vree WG (September 1987) The Grain Size of Parallel Computations in a Functional Program. In: Conference on Parallel Processing and Applications, Ed. Chiricozzi E, d'Amico A, Elsevier Science Publishing, L'Aquila, Italy, 363–370
23. Vree WG (August 1988) Parallel Graph Reduction for Communicating Sequential Processes. PRM Project Internal Report D-26, Department of Computer Systems, University of Amsterdam