# Introduction

Programming language implementations have incorporated *garbage collection* since at least 1960. In a garbage collected language implementation, the garbage collector discovers memory that is no longer in use (the garbage) and returns it to the free memory pool. Garbage collection is important because the alternative, explicit deallocation of dead objects, is a source of many program bugs and program maintenance problems, including storage leaks, dangling pointers, and module interfaces cluttered by deallocation protocols. Garbage collection makes it easier to write correct and maintainable programs.

The garbage collectors I will concern myself with are the precise, tracing collectors. A *tracing* collector discovers garbage by traversing the graph of reachable objects starting with a set of *root pointers* that usually includes the processor registers and the program stack. An object is *reachable* from another if the latter contains a pointer to the former, and any object not transitively reachable from the root pointers is garbage. By abuse of terminology, the reachable objects are said to be *live*[1]; objects are said to *die* when they become unreachable. A *precise* collector knows with certainty whether a memory location visible to it contains a pointer or not. A collector can be precise without being tracing (*eg*, reference counting collectors) or tracing without being precise (*eg*, so-called conservative collectors).

A garbage collection must be performed every time no memory is available for allocation. Tracing collectors do not examine the dead objects, so the first-order cost of garbage collection is the cost of tracing live objects. Thus, the first-order overhead of garbage collection is the ratio of the number of objects traced to the number of objects allocated. Traditionally, tracing has been called *marking* and allocation has been called *consing*; the ratio is therefore called the *mark/cons ratio*. The smaller the mark/cons ratio, the less work is spent on collecting garbage.

Figure 1 shows an example of copying garbage collection. The memory is split into two spaces. Initially the upper space is used for allocation. When that space is full, all reachable objects are evacuated from the upper space to the lower space, and both inter-object pointers and root pointers are updated to reflect the new locations of the objects. Program execution can continue, with allocation taking place in the lower space.

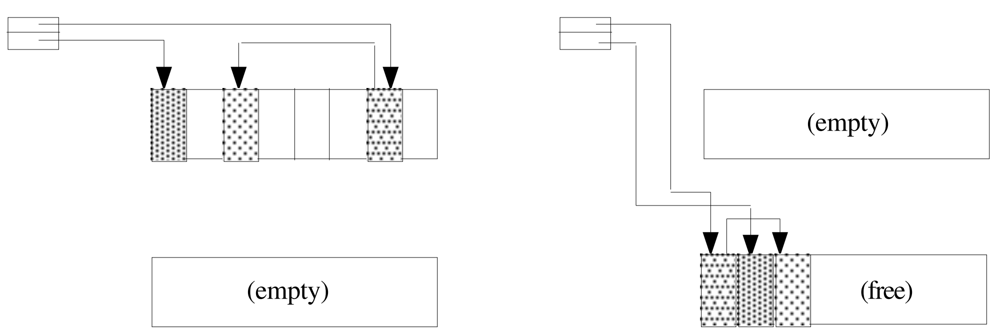


Figure 1: Simple copying garbage collection, before and after collection. The small box outside the heap is the set of root pointers. White cells in the heap are garbage.

Garbage collectors can lower the mark/cons ratio by using more memory or by collecting cleverly selected subsets of the heap. The *weak generational hypothesis* says that dynamically allocated objects exhibit a regularity: most objects die young. This hypothesis seems to hold not just for functional or quasi-functional languages, but also for languages like C and C++. In fact, not only do objects tend to die young, but the vast majority of objects die very quickly.

[1] Properly, a "live" object is one that will be used in the future; however, certain conservative approximations to liveness that are more precise than reachability are sometimes useful.

*Generational garbage collectors* take advantage of the weak generational hypothesis by observing that there will be a higher proportion of garbage among young objects than among the old and that collecting among young objects therefore lowers the mark/cons ratio. Generational collectors segregate objects by age (where each group of objects of similar age makes up a generation), collect younger generations more frequently than older generations, and thereby reduce the cost of allocating short-lived storage by an order of magnitude. Generational collectors have been spectacularly successful in reducing the overall cost of garbage collection.

Figure 2 shows a simple generational collector with two generations before and after a collection in the youngest generation (a *minor* collection). Allocation takes place in the young generation (the upper area). When the young generation is full, all reachable objects in the young generation are evacuated to the old generation (the lower area), and the root pointers and inter-object pointers are updated. Again, the cost of collection is proportional to the number of objects evacuated. As we can see, there are live objects that were not evacuated, as they already are in the old generation; this example shows how a generational collector can be more efficient than a non-generational collector.

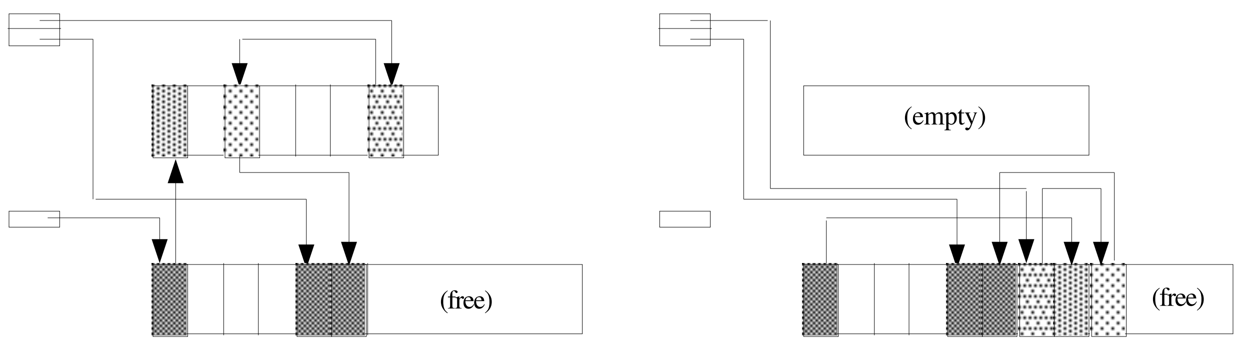


Figure 2: Generational collector with two generations, before and after a minor collection. The upper space is the young generation, the lower space is the old generation. The upper box outside the heap is the set of root pointers, and the lower box is the remembered set.

A generational collector can avoid scanning all objects in the old generation for pointers into the young generation by keeping track of those old objects that do contain pointers to young objects; the set of those old objects is called the *remembered set* and is shown as an additional box outside the heap in Figure 2.

When the old generation is full, a *major collection* must be performed; it collects the old generation using the algorithm of the simpler collector (Figure 1).

Generational collectors make fast garbage collection the rule, not the exception. However, the weak generational hypothesis holds only for very young objects, and objects that survive the first short while after their birth should be managed using other strategies.

Generational collectors typically manage objects that survive their first garbage collection by *promoting* them quickly from young to middle-aged status, and then more slowly from middle-aged to old status.[2] Objects that have lived long enough to have old status are said to have become *tenured*. Collecting among middle-aged objects is usually more expensive than collecting among younger objects, but not nearly as expensive as collecting among tenured objects: in a large system, there will be a large number of tenured objects, and many of them will usually be live.

Methods for performing promotion and their relative efficiency is the subject of the rest of this paper, and of my thesis.

[2] In the simple collector shown in Figure 2, there are no middle-aged objects, and all objects are promoted immediately from young to old status.

# Motivation and Scope

A central dilemma of generational garbage collection is inherent in the promotion process. This is the *tenuring problem*. If objects are promoted quickly, then objects of intermediate lifetimes will become tenured before they die, yet will die not long after becoming tenured, and the amount of tenured garbage will build up, increasing the frequency of (expensive) collections among tenured objects. If objects are promoted slowly, then each long-lived object will be traced a number of times before it is finally tenured, increasing the amount of work the garbage collector spends on it and reducing the advantage of generational collection over non-generational collection.

The program logic responsible for determining how objects are promoted is called the *promotion policy*. Promotion policies differ in how they choose to promote objects that survive a garbage collection. Most promotion algorithms are capacity-based (they promote when the amount of live data in a generation exceeds some threshold) or frequency-based (they promote at every collection).

Slow promotion–as in a system where objects are promoted along a pipeline of generations until they finally reach the oldest generation–is an ideal solution if the *strong generational hypothesis* is true. That hypothesis says that even among middle-aged and old objects, relatively younger objects have a higher mortality rate than relatively older objects. In a pipeline of generations, collecting among the younger objects more often than among the older will thus be less expensive (in terms of the mark/cons ratio) than collecting among both younger and older; a pipeline will reduce both the mark/cons ratio and the amount of tenured garbage. To date, however, no data that support the strong generational hypothesis are available, and the data that are available suggest that the strong generational hypothesis does not in fact hold for most programs. This suggests that a deep pipeline of generations will not be helpful in reducing work for most programs.

In fact, most garbage collection researchers do not worry about overall work and consider generational collection mainly a way to attain short garbage collection pauses [Ungar84, Appel89, Ungar92, Barrett95], to reduce memory consumption and paging [Ungar84, Moon84, Courts88, Zorn89], or to postpone major garbage collections temporarily or indefinitely [Moon84, Ungar84, Caudill86, Shaw88, Wilson89b, Ungar92]. Each design idiosyncratically trades off the risk of premature tenuring against the amount of tracing that is done among middle-aged objects. Some designers present data in support of their approach [Wilson89b, Zorn89, Ungar92], some admit that they have a good feeling about their policy but that it may need to be tuned [Caudill86], but most do not address the issue of the promotion policy's effectiveness at all.

Programs vary a great deal in how much they allocate and in how long their data live. Zorn reports on several large Lisp programs; the numbers in the following illustrative example were computed from results in his thesis [Zorn89].

One of Zorn's test programs is a commercial compiler for Common Lisp. That program has an average allocation rate of about 400KB per second, so a 400KB youngest generation requires one collection every second. At that time, about 94KB is still live and will be promoted to the second generation, so an 800KB second generation will fill up in 800/94=8 collections. When the second generation is full and must be collected, 3200KB will have been allocated, and some 309KB will still be live.

To simplify calculations, say that 300KB must be traced, reclaiming 500KB of memory and giving a mark/cons ratio of .38. The second generation will then fill up after only another 5 collections in the youngest generation, at which point about 440KB will be live, and the mark/cons ratio for the second collection will be .88, with an overall ratio of .57.

After the first collection the collector could instead elect to expand the second generation by 300KB, allowing it to grow to 1100KB before the next collection, when 480KB will be live. This gives a mark/cons ratio of .6 for the second collection and .49 overall.

Alternatively, the 300KB of extra memory can be used for a third generation. If all the live data are promoted from the second generation to a third the first time the second generation is full, then the overall second-generation mark/cons ratio remains at .38 also after the second collection.

Several points are worth making about the preceding example.

First, the promotion rate out of the youngest generation–nearly 25%–is high compared to the other test programs in Zorn's thesis, although not as high as we have observed in several other programs [Clinger97].

Second, adding a third generation reduces the amount of collection work, because the data that survives until the time of the second collection is only traced once, not twice.

Third, the reduction in work achieved with the use of the third generation does not come for free: after the second collection there is 600KB-480KB=120KB of garbage in the third generation, and it will only be reclaimed when the third generation is collected. If the third generation is the oldest generation, then collecting it may be quite expensive, as long-lived objects accumulate in the oldest generation, and may all have to be traced during a collection. The data that were live after the second-generation collection may only remain live for a little while longer, and the second generation can then be collected at very low cost. If so, promoting the data into the oldest generation will have been a bad choice. That is not far-fetched: it is well-known that programs have spikes in their use of memory, as temporary data structures are built up and then die [Wilson95].

Fourth, there are other strategies that can be used to cope with rapid allocation. *Nonpredictive collection* tries to minimize the mark/cons ratio by collecting the old generation where data have had the longest time to die. *Feedback strategies* attempt to detect gross program behavior and adapt the collection policy: for example, if the collector can determine that the generational hypothesis still holds for a group of older data, then that data can be moved along a pipeline of generations; otherwise, the data can be managed differently.

In summary, I therefore propose to examine the following open problem:

> How should we perform promotion in a generational garbage collector so that the collector's effectiveness is maximized?

The issue can be phrased assertively in the form of a thesis:

> A promotion policy that takes information about the survivors of a garbage collection into account can be more effective than one that does not.

but that is conjecture at this point.

Specifically, I propose to design and implement two new promotion policies, one based on a feedback model and the other based on a nonpredictive model, and to compare the feedback algorithm and the nonpredictive algorithm to several previously proposed and implemented promotion mechanisms.

An important part of this work will be analysis of object behavior in programs. Object behavior analysis differs from object lifetime analysis. Lifetime analysis results in information about average object lifetimes, or about the amount of live data over time. Object behavior analysis combines these and reveals facts about how groups of objects mature and die.

The feedback algorithm will be allowed to use information about the survivors of a single garbage collection, and any other information available to it, to decide how to perform promotion. This contrasts with previously implemented systems, which have only used frequency or capacity exhaustion to guide promotion.

The non-predictive algorithm, following the model of non-predictive garbage collection [Clinger97], will not use any information about survivors to guide it, but will collect the generation least recently collected and promote the survivors.

A generational garbage collector has to collect the oldest generation occasionally, yet I will not examine how this should be done. The methods section addresses this problem.

There is some closely related work (general related work is discussed at length below). Wilson [Wilson89a] has suggested that a high survival rate should trigger more aggressive promotion; Ungar and Jackson [Ungar92] have implemented and studied this for the case of a two-generation collector with a very high penalty on premature tenuring. Barrett and Zorn [Barrett95] have implemented a collector that additionally demotes objects when survival rates are low, thus reducing the penalty of premature tenuring. In both collectors, the emphasis was on achieving reasonable pause times, not on increasing garbage collector effectiveness. Hudson *et al* [Hudson91] have speculated that survival rates can be used to modify the promotion policy dynamically.

# Contributions

My dissertation will make three contributions. First, I examine the viability of the non-predictive and feedback-based promotion strategies by comparing them to several existing strategies. Second, I establish a framework for discussing the effectiveness of promotion algorithms. Third, I characterize the non-predictive promotion strategy theoretically with an aim to establish its effectiveness.

# Literature Review

*General*

Garbage collection was first described by McCarthy [McCarthy60]; Knuth [Knuth73] and Cohen [Cohen81] survey early work. Generational collection was first described in 1983 by Lieberman and Hewitt [Lieberman83], although earlier work foreshadowed the technique [Deutsch76, Bawden77, Hanson77]. Recent works include the book by Jones and Lins [Jones96] and the survey by Wilson *et al* [Wilson92].

*Promotion policies*

Lieberman and Hewitt first described generational garbage collection [Lieberman83]. Their system performs no promotion at all: when the youngest generation is somewhat full after a garbage collection, a new youngest generation is created for allocation. Each generation can be collected independently of the others.

Moon's ephemeral garbage collector [Moon84] has a number of "ephemeral" (young) generations that can be collected independently and a "dynamic" (old) generation for older data. Objects are allocated in the youngest ephemeral generation; when it fills up, all full ephemeral generations are *condemned*, and a garbage collection is started. Live objects are evacuated from each condemned generation to the following generation. Thus, if there are $n$ ephemeral generations, each object that lives to become old is copied $n$ times. Moon's algorithm promotes objects slowly and prefers to work more to avoid promoting an object too soon. At the same time, it bounds the amount of work that is expended on each object before it is promoted to the dynamic generation.

Ungar's generation-scavenging garbage collector [Ungar84] has two generations, where the youngest generation consists of a large creation area and a smaller area for surviving objects. When the creation area is full, live objects are evacuated from the creation area and the survivor area into a new survivor area. Each object contains a count of the number of collections the object has survived; when the count reaches a fixed limit, the object is promoted into the old generation. Ungar's algorithm promotes objects only reluctantly, because the old generation is never collected while the system is in use. However, if the new survivor area fills up during a garbage collection, then the algorithm promotes all objects that won't fit. This bail-out strategy tends to promote young objects, and the amount of tenured garbage can increase rapidly.

Caudill and Wirfs-Brock [Caudill86] combined Moon's and Ungar's algorithms into a multi-generational collector that maintains a collection count for each object; an object is promoted to the next generation after its count reaches a preset limit. Instead of having a large limit like Ungar, the limit is small and the number of generations has instead been increased (to eight). The intuition is that the older generations are collected less often, hence the overall work relative to Ungar's algorithm should be reduced, yet premature promotion should not be more of a problem. The oldest generation is not collected at all.

Shaw [Shaw88] proposed a simplification of Moon's algorithm where objects are aged in buckets that are akin to generations, but unlike Moon's generations that are collected independently when full, all buckets are collected together, with survivors from a bucket promoted to the next bucket; this system is called a "bucket brigade". Shaw then implemented his system by grouping buckets into "areas", trading some age precision for implementation simplicity.

Appel's garbage collector for Standard ML of New Jersey [Appel89] has two generations. Objects are allocated in the young generation, and when it is full, live objects are promoted *en masse* to the old generation. When the old generation is full, the entire heap is collected at once. The algorithm is notable for its simplicity.

Wilson and Moher [Wilson89a, Wilson89b] simplified Shaw's algorithm further by reducing the number of buckets to two. Thus, objects are promoted from the creation area (the first bucket) into the second bucket, and from the second bucket into the old generation; objects that reach the old generation have survived two collections. If the creation area is large, then objects in the second bucket have time to die and chances are fair that the data promoted into the old generation will not die soon. The main benefit of their system is that it guarantees that very little work is spent on each promoted object—each object is copied twice—yet no just-created data are ever tenured.

Ungar and Jackson [Ungar92] improved on Ungar's algorithm by selectively promoting only objects with the highest collection count when the amount of data in the young generation reaches a limit during a garbage collection. They found that this strategy was effective in reducing the amount of tenured garbage.

Reppy [Reppy94] took Wilson and Moher's algorithm one step further and used it in a multi-generational collector for Standard ML of New Jersey. In this collector, each young generation is a two-bucket bucket brigade. This system is therefore similar to that of Caudill and Wirfs-Brock but objects do not need explicit copy counts. When a particular generation is collected, all younger generations are also collected; this simplifies the collector.

Barrett and Zorn [Barrett95] adapted Ungar and Jackson's algorithm by allowing objects to be demoted. If there are few live objects in the youngest generation after a collection, some of the previously promoted objects will be included in the next collection, thereby allowing some tenured garbage to be reclaimed.

*Survival rates and object behavior*

Many researchers have characterized object size distributions, object allocation rates, and object survival rates for various programming languages, both garbage collected and not.

The most comprehensive work is Zorn's dissertation [Zorn89]. Zorn presents lifetime data for eight large and medium-size Lisp programs. His data show several interesting trends. First, objects die very quickly: only between 0% and 26% of the objects survive for one second. Yet at the same time, between 0% and 12% survive four seconds, and between 0% and 7% survive one minute. Zorn's data have one peculiar problem: he measured the time each object was (properly) live—from birth to last reference—rather than the time it was reachable. Garbage collectors cannot in general compute liveness accurately, so the time an object is reachable must be expected to be longer than the lifetimes reported by Zorn.

Hayes [Hayes91] presents lifetime data for objects in Cedar programs and shows that even in Algol-like languages most objects die quickly. He also points out that, unlike what is the case for young objects, there appears to be no correlation between object age and future life expectancy for older objects.

Ungar and Jackson [Ungar92] present lifetime profiles for four Smalltalk-80 programs. These profiles show that most objects die quickly, but that a substantial volume of data can live for a long time, and also illustrate that programs have data structures of "intermediate" lifetime.

Stefanovic and Moss [Stefanovic94] investigate the survival rates and object lifetimes for three programs in Standard ML of New Jersey. They report lifetime results for objects' relative youth, up to 3MB after allocation. In SML/NJ, which allocates as much as 16MB per second because all continuation frames are allocated on the heap, 3MB is a short time and their results are not particularly useful for evaluating a promotion strategy.

Wilson, Johnstone, Neely, and Boles [Wilson95] present live storage profiles for several substantial and very different C programs. These profiles show that the amount of live storage can vary greatly over time, with rapid heap expansion and contraction as the program builds up a data structure or deallocates it and moves on to the next phase. Other programs have a constant amount of live storage yet allocate rapidly, or have profiles that are quite unpredictable.

Clinger and Hansen [Clinger97] present lifetime data for three fast-allocating programs, and argue that generational garbage collectors derive their efficiency in part from predicting which generation to collect rather than from predicting object lifetimes. They show the effectiveness of this principle for a two-generation collector.

*Lifetime prediction*

A number of authors have suggested using lifetime prediction in one form or another [Moon84, Hanson90, Barrett93, Cohn97, Cheng98]. In a system that uses lifetime prediction, objects are allocated in a generation that will be collected with a frequency that approximates the object's expected lifetime so that on the average, the object will be subject to very few garbage collections. While this work is interesting and has its place in the language implementor's bag of tricks, it is not directly applicable to generational garbage collection. This is a somewhat subtle point. Lifetime-predictive systems can reduce the mark/cons ratio by effectively predicting object lifetime and using a frequency-based garbage collector. Generational collectors, on the other hand, can reduce the mark/cons ratio by collecting a generation with a high ratio of garbage to live data, hence they derive an advantage from heuristically predicting the generation to collect, not by predicting object lifetimes [Clinger97]. Lifetime prediction results in a very different kind of garbage collector than garbage prediction.

# Methods and Techniques

*Test system and platform*

I will use Larceny [Clinger94], our research implementation of Scheme [Kelsey98, IEEE], as the experimental system. Larceny is mature and has served us well as a research system for garbage collection for many years. I know its garbage collectors and other internals well, because I wrote them myself. Its collectors use most known important techniques for good garbage collector performance, and revisions planned independently of my thesis work will make it even more competitive. Good performance in the overall system is important for most garbage collection research: even though the primary measure of work is the mark/cons ratio, performance measurements that take all effects into account are also important, since second-order effects do matter [Hicks97].

*Comparison Algorithms*

This section describes the existing promotion algorithms that I want to compare the new algorithms to. I have chosen these algorithms because they together represent most of the approaches that have been used in past implementations.

Notable by its absence from the following list is Appel's algorithm [Appel89]. Appel's algorithm, although both simple and popular among implementors who are looking for a simple generational collector design [Sansom93, Clarke96], suffers badly from premature promotion [Sansom93, Reppy94], and I do not consider it interesting for my study.

Also absent are Ungar and Jackson's algorithm [Ungar92] and Barrett and Zorn's algorithm [Barrett95], because both are directed towards keeping pause times low, not towards reducing work. They are therefore not appropriate comparisons.

<u>Larceny's capacity-based algorithm</u>
This algorithm is simple and conservative, and seems to work well in practice. It may promote very young data prematurely because it insists on deciding, before a collection starts, whether the generation that receives the live data may overflow.

<u>Standard ML of New Jersey's multi-generational bucket brigades</u>
This algorithm is described by Reppy [Reppy94] and also above. The two-bucket bucket brigades used in each generation slows down promotion but may require a slightly more complicated (slower) collector than Larceny's collector, because a datum's destination generation depends on its source generation.

An added benefit of choosing this algorithm is that by setting the number of generations to three (where only the middle will have a bucket brigade), I get a system that is very close to that of Wilson and Moher [Wilson89a, Wilson89b], yielding another comparison point if desired.

*Test programs*

A set of test programs that will be used both for analysis in preparation of designing the feedback algorithm and for comparing all the promotion algorithms must be selected. We already have a number of programs that we have used in the past and that may still be useful [Clinger97], but one of the problems with Scheme's research emphasis is that the number of substantial and realistic programs is small. On the other hand,many Scheme programs are available on the Internet. I could write some programs myself, but substantial and realistic programs are not written in a week, and if I write them myself they may biased in favor of my promotion algorithms, or vice versa.

My plan is therefore to solicit programs from people in the Scheme community; leads so far include compilers and other programming tools (Stalin, SoftScheme), a symbolic algebra system (Jacal), a parser generator (Early), an agent system (Julia), and a rule-based expert system (Min).

I expect that about 10 test programs is adequate, although I need to examine a larger number than that. All the selected programs need to be reasonably difficult for the garbage collector, hence, they should not be "typical" programs (for which garbage collection is easy), and there needs to be diversity in the group with respect to storage use and object lifetimes.

*Storage profiles*

Larceny has a mechanism for computing object populations. This mechanism (obscurely called SRO, for "Standing Room Only") is useful to me in two ways.

First, SRO can be used to compute the live data profile over time for any program. This method was previously used by Clinger to compute several novel live data profiles [Clinger97]. Live data profiles for the benchmark programs will make it easier to understand benchmark results, and may also help in tuning the design for the feedback algorithm.

Second, using an extension of the existing SRO mechanism, it is possible to compute exactly the set of live objects in any generation without affecting future decisions by the garbage collector. I can therefore compare the collector's heuristic decisions to decisions that would be made on the basis of exact data.

*Theoretical work*

Of particular interest is the behavior of the non-predictive algorithm when it is faced with some known patterns of object behavior, such as objects among which the generational hypothesis holds, and object structures that result from program phase behavior (ramps, spikes, and continually-growing data structures).

The non-predictive algorithm will be analyzed in an idealized setting to try to understand its behavior. Clinger has hypothesized that the non-predictive algorithm "is robust in the sense that randomized versions of it frustrate attempts to create a worst-case benchmark", and that game theory may be used to explore this conjecture.

To the extent it is possible, it would be interesting to analyze the other promotion algorithms' performance in idealized settings as well.

*Promotion algorithms*

The feedback algorithm will be designed based on the analysis of some of the test programs. It is important that not all test programs are used in the design; the test programs not used should serve as a control on the design.

The non-predictive algorithm will be designed based on previous experience with nonpredictive collection and on the theoretical analysis, if that gives us any insights.

Reppy's algorithm will be implemented based on the description in his paper [Reppy94].

Some or all of the algorithms may require extensions to Larceny's garbage collector structure. For example, Moon's algorithm requires that the remembered set–the set of objects that contain pointers from an older generation to a younger–contain also objects that contain pointers from younger objects to older objects. Larceny's garbage collector is implemented as a set of cooperating agents and should be able to accept this extension without much trouble; the existing nonpredictive collector uses a similar remembered set.

*Results*

Obtaining results is not merely a matter of running the test programs in Larceny with the appropriate promotion policy in effect. For better or for worse, Larceny has several tuning knobs, in particular the number of generations, their sizes, and the overall load factor (ratio of live data to storage in use) can be adjusted. A reasonable set of parameter values must be chosen, and each program must be run with each parameter setting for each promotion strategy. To get meaningful results, each program must be run several times, and representative results selected.

What do we want to measure? Certainly the overall mark/cons ratio, the overall memory consumption, and running time are important, but the mark/cons ratio for each generation might be useful in interpreting the results. Larceny already has an instrumentation infrastructure and it should not be a problem to add instrumentation to obtain specialized data.

*Analysis*

A generational garbage collector has to collect the oldest generation occasionally, yet I will not examine how this should be done. Therefore, the problem statement must be qualified as follows. Effectiveness is measured by overall mark/cons ratio, not by the mark/cons ratio in only the youngest generations, and memory consumption is measured as the total memory in use by all generations. There are some subtleties still: for example, while one promotion strategy may trigger a full collection, another may avoid it yet would have triggered a full collection had only a few bytes more been promoted. Thus a notion of *amortized effectiveness* needs to be defined and used.

The analysis of the results is also complicated by the many parameters involved. Since garbage collection is about memory management, it seems reasonable to use comparative performance (both running time and mark/cons ratio) under equal memory use as the primary measure of goodness.

# Progress Plan

I plan to graduate by the end of Spring term, 2000. Thus, five terms are available (Winter, Spring, and Fall 1999; Winter and Spring 2000). Summer 1999 is left open for schedule adjustments, outside work, and/or vacation.

The progress plan is laid out by term. There is an implied milestone at the end of each term: the work scheduled for that and previous terms should be completed by the end of the term. That said, I expect that actual progress will be somewhat less linear than the outline suggests, and prefer, with the committies' indulgence, to present the following outline as a guide rather than as a rigorous plan.

I will be writing the dissertation as I go along.

*Winter 1999: Further analysis of the problem*

- Further analysis of the problem
- Select benchmark programs
- Create object lifetime profiles of benchmark programs
- Revise PLDI '97 paper [Clinger97] for journal

*Spring 1999: Design and implementation*

- Design and implement feedback algorithm
- Design and implement non-predictive promotion algorithm
- Implement Reppy's algorithm

*Summer 1999: No assigned work*

*Fall 1999: Analysis of the results*

- Run benchmarks
- Compare results across benchmark programs and promotion strategies
- Relate results to lifetime profiles and other data

*Winter 2000: Wrap-up*

- Conclude, finish writing

*Spring 2000: No assigned work*

One of the things I would like to do is to submit the work for publication while I'm still in school. A conference like ACM Programming Language Design and Implementation (PLDI) would be suitable. The PLDI deadline is usually in early November, requiring work scheduled for Fall 1999 to be performed during Summer 1999.

# Acknowledgements

Will Clinger, Doug Orleans, Johan Ovlinger, Jon Blow, and Amit Patel all read the proposal and provided me with useful feedback.

# Bibliography


[Agesen98] Ole Agesen, David Detlefs, and J. Eliot B. Moss, *Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines*. In ACM '98 Conference on Programming Language Design and Implementation, 269–279, Montreal, Canada. SIGPLAN Notices 33(5), May 1998.

[Appel89] Andrew W. Appel, *Simple Generational Garbage Collection and Fast Allocation*. Software—Practice and Experience, 19(2), 171–183, February 1989.

[Appel92] Andrew W. Appel, *Compiling with Continuations*. New York, NY 1992. Cambridge University Press.

[Baker93] Henry G. Baker, *'Infant Mortality' and Generational Garbage Collection*. SIGPLAN Notices 28(4), 55–57, April 1993.

[Barrett93] David A. Barrett and Benjamin G. Zorn, *Using Lifetime Predictors to Improve Memory Allocation Performance*. In ACM '93 Conference on Programming Language Design and Implementation, 189–198, Albuquerque, NM. SIGPLAN Notices 28(6), June 1993.

[Barrett95] David A. Barrett and Benjamin G. Zorn, *Garbage Collection Using a Dynamic Threatening Boundary*. In ACM '95 Conference on Programming Language Design and Implementation, 301–314, La Jolla, CA. SIGPLAN Notices 30(6), June 1995.

[Bawden77] A. Bawden, R. Greenblatt, J. Holloway, T. Knight, D. A. Moon, and D. Weinreb, *Lisp Machine Progress Report*. MIT AI Memo 444, August 1977.

[Bekkers92] Y. Bekkers, O. Ridoux, and L. Ungaro, *Dynamic Memory Management for Sequential Logic Programming Languages*. In [IWMM92], 82–102.

[Caudill86] Patrick J. Caudill and Allen Wirfs-Brock, *A Third Generation Smalltalk-80 Implementation*. In ACM OOPSLA '86 Conference Proceedings, 119–130.

[Cheng98] Perry Cheng, Robert Harper, and Peter Lee, *Generational Stack Collection and Profile-Driven Pretenuring*. In ACM '98 Conference on Programming Language Design and Implementation, 162–173, Montreal, Canada. SIGPLAN Notices 33(5), May 1998.

[Clarke96] Charles L. A. Clarke and David V. Mason, *Compacting Garbage Collection can be Fast and Simple*. Software—Practice and Experience, 26(2), 177–194, February 1996.

[Clinger94] William D Clinger and Lars Thomas Hansen, *Lambda the Ultimate L, or, a Simple Optimizing Compiler for Scheme*. In 1994 ACM Conference on Lisp and Functional Programming, 128–139. LISP Pointers VII(3), July–September 1994.

[Clinger97] William D Clinger and Lars T Hansen, *Generational Garbage Collection and the Radioactive Decay Model*. In ACM '97 Conference on Programming Language Design and Implementation, 97–108, Las Vegas, NV. SIGPLAN Notices 32(5), May 1997.

[Clinger98] William D Clinger, *Proper Tail Recursion and Space Efficiency*. In ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, Montreal, Canada. SIGPLAN Notices 33(5), 174–195, May 1998.

[Cohn97] David A. Cohn and Satinder Singh, *Predicting Lifetimes in Dynamically Allocated Memory*. In M. Mozer, M. Jordan, and T. Petsche, eds., *Advances in Neural Information Processing Systems 9*, MIT Press, 1997.

[Cohen81] Jaques Cohen, *Garbage Collection of Linked Data Structures*. ACM Computing Surveys, 13(3), 341–367, September 1981.

[Courts88] Robert Courts, *Improving Locality of Reference in a Garbage-Collecting Memory Management System*. Communications of the ACM 31(9), 1128–1138, September 1988.

[Deutsch76] L. Peter Deutsch and Daniel G. Bobrow, *An Efficient Incremental Automatic Garbage Collector*. Communications of the ACM 19(7), 522–526, July 1976.

[Hanson77] David R. Hanson, *Storage Management for an Implementation of SNOBOL4*. Software—Practice and Experience, 7, 179–192, 1977.

[Hanson90] David R. Hanson, *Fast Allocation and Deallocation of Memory Based on Object Lifetimes*. Software—Practice and Experience, 20(1), 5–12, January 1990.

[Hayes91] Barry Hayes, *Using Key Object Opportunism to Collect Old Objects*. In ACM OOPSLA '91 Conference Proceedings, 33–46, Phoenix AZ, October 1991.

[Hicks97] Michael W. Hicks, Jonathan T. Moore, and Scott M. Nettles, *The Measured Cost of Copying Garbage Collection*. In 1997 ACM International Conference on Functional Programming, 292–305, Amsterdam, The Netherlands. SIGPLAN Notices 32(8), August 1997.

[Hudson91] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight, *A Language-Independent Garbage Collector Toolkit*. COINS Technical Report 91-47, Department of Computer and Information Science, University of Massachusetts, Amherst, September 1991.

[IEEE] IEEE Standard for the Scheme Programming Language. IEEE Std 1178-1990.

[IWMM92] Y. Bekkers and J. Cohen (Eds.), *Memory Management–International Workshop IWMM 92*. Lecture Notes in Computer Science 637, St. Malo, France, 1992. Springer-Verlag.

[IWMM95] Henry G. Baker (Ed.), *Memory Management–International Workshop IWMM 95*. Lecture Notes in Computer Science 986, Kinross, UK, 1995. Springer-Verlag.

[Jones96] Rihard Jones and Rafael Lins, *Garbage Collection–Algorithms for Dynamic Memory Management*. New York, NY, 1996. Wiley.

[Kelsey98] Richard Kelsey, William D Clinger, and Jonathan Rees, eds., *Revised$^5$ Report on the Algorithmic Language Scheme*. SIGPLAN Notices 33(9), 26–76, September 1998.

[Knuth73] Donald E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Reading, MA, 1973. Addison-Wesley.

[Lieberman83] Henry Lieberman and Carl Hewitt, *A Real-Time Garbage Collector Based on the Lifetimes of Objects*. Communications of the ACM 26(6), 419–429, June 1983.

[McCarthy60] John McCarthy, *Recursive functions of symbolic expressions and their computation by machine*. Communications of the ACM 3, 184–195, 1960.

[Moon84] David A. Moon, *Garbage Collection in a Large Lisp System*. In ACM 1984 Conference on Lisp and Functional Programming, 235–246.

[Reppy94] John H. Reppy, *A High-performance Garbage Collector for Standard ML*. AT&T Bell Laboratories, 1994.

[Sansom93] Patrick M. Sansom and Simon L. Peyton Jones, *Generational Garbage Collection for Haskell*. In 1995 ACM Conference on Functional Programming and Computer Architecture, 106–116.

[Shaw88] Robert Allen Shaw, *Empirical Analysis of a Lisp System*. Doctoral thesis, Stanford University, February 1988.

[Stefanovic94] Darko Stefanovic and J. Eliot B. Moss, *Characterisation of Object Behaviour in Standard ML of New Jersey*. In 1994 ACM Conference on Lisp and Functional Programming, 43–54. LISP Pointers VII(3), July–September 1994.

[Ungar84] David Ungar, *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm*. In ACM SIGSOFT-SIGPLAN Practical Programming Environments Conference, Pittsburgh PA. SIGPLAN Notices 19(5), 157–167, April 1984.

[Ungar92] David Ungar and Frank Jackson, *An Adaptive Tenuring Policy for Generation Scavengers*. ACM Transactions on Programming Languages and Systems 14(1), 1–27, January 1992.

[Wilson89a] Paul R. Wilson, *A Simple Bucket-Brigade Advancement Mechanism for Generation-Based Garbage Collection [working paper]*. SIGPLAN Notices 24(5), 38–43, May 1989.

[Wilson89b] Paul R. Wilson and Thomas G. Moher, *Design of the Opportunistic Garbage Collector*. In OOPSLA ’89 Conference Proceedings, 23–35, October 1989.

[Wilson92] Paul R. Wilson, *Uniprocessor Garbage Collection Techniques*. Expanded version of paper appearing in [IWMM92].

[Wilson95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, *Dynamic Storage Allocation: A Survey and Critical Review*. In [IWMM95], 1–116.

[Zorn89] Benjamin Goth Zorn, *Comparative Performance Evaluation of Garbage Collection Algorithms*. Doctoral Thesis, University of California at Berkeley, December 1989.