THE IMPACT OF PROGRAMMING STYLE ON THE PERFORMANCE OF SCHEME PROGRAMS

by

LARS THOMAS HANSEN

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

August 1992

APPROVED: ______________________________

Dr. William D. Clinger

An Abstract of the Thesis of

Lars Thomas Hansen for the degree of Master of Science

in the Department of Computer and Information Science

to be taken August 1992

Title: THE IMPACT OF PROGRAMMING STYLE ON THE PERFORMANCE OF SCHEME PROGRAMS

Approved: ________________________________

Dr. William D. Clinger

Conventional wisdom among programmers states that an imperative programming style necessarily gives better performance than a functional style; that iteration together with an explicit stack is faster than recursion; that assignment is faster than parameter passing; and that allocation of intermediate data structures is wasteful if the allocation could have been avoided.

We believe that these and other claims about the impact of style on performance are questionable in the presence of a modern run-time system and a generational garbage collector. We have therefore measured the actual performance of a number of Scheme benchmark programs under several garbage collectors.

Our findings show that a functional, allocation-hungry style is quite viable in a wide variety of cases, and that functional programs can have performance equal to or better than the corresponding imperative programs. We give recommendations for when a functional style is appropriate and when it is not.

VITA

NAME OF THE AUTHOR: Lars Thomas Hansen

PLACE OF BIRTH: Askim, Norway

DATE OF BIRTH: June 14, 1965

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Østfold Distriksthøgskole, Norway
Pacific Lutheran University

DEGREES AWARDED:

Master of Science, 1992, University of Oregon
Bachelor of Science, 1990, University of Oregon
DH-kandidat, 1987, Østfold Distriktshøgskole

AREAS OF SPECIAL INTEREST:

Programming Language Design and Implementation
Distributed Architectures and Programming Systems

PROFESSIONAL EXPERIENCE:

Graduate Teaching Fellow, University of Oregon, Eugene, 1991–1992

Systems Administrator, University of Oregon, Eugene, 1990–1992

Software Developer, Bergen, Norway, 1987–1988

Computer Programming Teacher, EDB-skolen, Bergen, Norway, 1987–1988

ACKNOWLEDGEMENTS

I would like to thank my advisor, Will Clinger, who provided me with not only the original idea for this thesis but also with encouragement, insight, a lecture on Texas and Texans, and comments sufficiently enigmatic to send me off investigating previously un-thought-of issues.

Part of the work on "Larceny," the test system used in this thesis, was financed by a grant from OACIS.

I would also like to thank Sastry Akella and Bart Massey, who read drafts of the thesis and provided feedback which improved both the presentation and the discussion of our findings.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Page

Page

CHAPTER I

INTRODUCTION

Programming Style Issues

Programming style is not a new issue, having been the topic of heated debate in the computer science community since at least the inception of structured programming in the late 1960s. At that time, one debated which kinds of statements were aiding program comprehension and how a program should be structured (in terms of control and data abstractions) to make it easier to design, write, read, and maintain.

The topic of style got a new direction with Backus' 1977 paper on functional programming [3]. Backus argues that the imperative style of programming is fundamentally flawed and that a better paradigm is a functional style. Part of his argument is that the assignment statement, which is at the core of the imperative style, is hard to reason formally about, and therefore that programs written in an imperative style are difficult to understand. A functional style, emphasizing expressions and function application and avoiding statements altogether, on the other hand, lends itself readily to formal analysis and programs written in a functional style can therefore be easier to understand than imperative programs. Functional programming was not new with Backus; already in 1960 John McCarthy had written a paper on Lisp, promoting an essentially functional style [16]. However, Backus' paper brought the concept of functional programming to the attention of a larger audience.

A functional style is characterized by being expression oriented rather than statement oriented. Functional programs make heavy use of functional abstractions in the style of the lambda calculus [2], use recursion to express iteration (and use rebinding rather than assignment to update loop variables), and frequently allocate storage with unlimited extent to either hold temporary values or to build up data structures. All data objects (or values) in a functional-style program are typically first-class, meaning that each object can be bound to an identifier, passed as an argument to a procedure, returned as the result of a procedure, or be incorporated in another (new) object. As object identity is an issue only in the presence of assignment, the implementation may copy or share objects at will. Some functional languages are lazy: evaluation of an expression is delayed until the value of the expression is needed, and if the value is never needed, the expression is never evaluated at all.

An example of a functional program is shown in figure 1; the program computes the inner product of two vectors.

```
(define (inner-product vector1 vector2)

  (define (loop i sum)
    (if (negative? i)
        sum
        (loop (- i 1) (+ sum (* (vector-ref vector1 i)
                                (vector-ref vector2 i))))))

  (loop (- (vector-length vector1) 1) 0))
```

Figure 1: Functional program which calculates the inner product.

In contrast, languages emphasizing an imperative style have separate constructs for procedure calls and loops, often discourage or disallow recursion, rely heavily on the assignment statement to maintain the program state, do not make rebinding of identifiers easy (to the extent that block structure is at all allowed), and often make the allocation and timely freeing of temporary storage of unlimited extent (lifetime) difficult or impossible. In fact, most imperative languages do not support general objects with unlimited extent. Usually, only objects of primitive types, e.g. numbers and pointers, are first-class. Objects do not have identity: an assignment makes a copy of the object (value) being assigned, and subsequent changes to one copy of the assigned object are not reflected in the other copies.

Figure 2 shows the inner-product procedure written in an imperative language.

```
int inner_product( int a[], b[], size )
{
  int sum, i;

  sum = 0; i = 0;
  while (i < size) {
    sum = sum + (a[ i ] * b[ i ]);
    i = i + 1;
  }
  return sum;
}
```

Figure 2: Imperative program which calculates the inner product.

The functional paradigm is now quite popular in universities and research laboratories. Functional programming has not, however, been widely adopted in the mainstream programming culture, mostly due to a perceived or actual performance penalty incurred by a purely functional style.

Some of the actual penalties incurred by a functional style may be ascribed to the current state of the art in computer architecture. The imperative paradigm, with the assignment statement at its center, fits the von Neumann computer[1] and its state transition semantics very well. A statement in an imperative language is merely a high-level description of a (typically small) number of instructions in the machine's native language. It is therefore a relatively simple task to make imperative programs run quickly on today's computers, and new computers are being built which make it easier to run programs written in imperative languages.

The functional paradigm, on the other hand, is not based on the state transition semantics of the von Neumann computer but rather on the notion of a function in mathematics, e.g. on the reduction semantics of the lambda calculus. Functional languages do not, as a rule, map nicely to modern computer architectures, and these languages can be hard to implement on conventional hardware, at least if one wants run-time efficiency comparable to that of imperative languages.

Functional languages also require fairly elaborate run-time systems. For example, the implementation of objects with unlimited extent requires a form of automatic storage reclamation, for example garbage collection. Garbage collection can be expensive, however. While it is the case that garbage collection is often less expensive than explicit memory management in programs that allocate and free much memory, imperative programs allocate and free such dynamic memory sparingly exactly because dynamic memory is hard to manage correctly and efficiently. When we compare existing programs in both styles it will therefore look like garbage collection is more expensive than explicit allocation and deallocation.

Some languages give the programmer a choice of which paradigm to use: the

[1]The term "von Neumann computer" was coined by Backus [3] and refers to the computer architecture devised by John von Neumann and his associates; these same architectural principles are still pervasive in modern digital computers.

language supports both an imperative and a functional style. Scheme [11] is such a language. In Scheme, which is based on the strict lambda-calculus, assignment to variables and mutation of compound objects is possible. On the other hand, all objects in Scheme (even lexical closures) are first-class, iteration is expressed via (tail-)recursive calls, objects have identity and unlimited extent, and Scheme therefore has much of the flavor of a functional language. Another mostly-functional language is Standard ML [18].

In a language such as Scheme, the question arises: which style should the programmer use for a particular program? In some cases, the answer depends on the algorithm one wishes to implement; in others, the choice is unclear. Consider a typical procedure, `map`. `Map` (in this simplified form) takes two arguments, a procedure and a list. The procedure passed to `map` must take one argument. `Map` applies the procedure to each element in the list in an unspecified order and returns a new list containing the results. There are at least four mostly obvious ways to implement `map`, shown in figure 3.

All implementations of map run in O(n) time and use O(n) space, but the exact amount and type of space allocated varies between implementations and makes the comparison interesting; for example, some implementations allocate extra list cells and others allocate extra stack space.

The first program, `map-1`, recurs down the input list, accumulating unfinished computations on the program stack. When the end of the list is reached, the new list is constructed on the way back out of the recursion. `Map-1` allocates no extra list cells, and it is written entirely in a functional style. However, `map-1` uses stack space in an amount proportional to the length of the input list.

The second program, `map-2`, uses an iterative strategy and accumulates

```
(define (map-1 proc list)
  (if (null? list)
      '()
      (cons (proc (car list)) (map proc (cdr list)))))


(define (map-2 proc list)

  (define (map-helper list1 list2)
    (if (null? list1)
        (reverse list2)
        (map-helper (cdr list1) (cons (proc (car list1)) list2))))

  (map-helper list '()))


(define (map-3 proc list)

  (define (map-helper list1 list2)
    (if (null? list1)
        (reverse! list2)
        (map-helper (cdr list1) (cons (proc (car list1)) list2))))

  (map-helper list '()))


(define (map-4 proc l)

  (define (map-helper l last-pair)
    (if (not (null? l))
        (let ((new (cons (proc (car l)) '())))
          (set-cdr! last-pair new)
          (map-helper (cdr l) new))))

  (if (null? l)
      '()
      (let ((first (cons (proc (car l)) '())))
        (map-helper (cdr l) first)
        first)))
```

Figure 3: Four strategies for the implementation of `map`.

results in reverse order in a helper list. When the input list is exhausted, the helper list is reversed, and the resulting list is returned. `Map-2` uses no extra stack space, but it creates an "unnecessary" list: the helper list. Hence, `map-2` allocates twice as much space for the list as it needs to. Like `map-1`, `map-2` is written in a functional style.

The third program, `map-3`, is much like `map-2`, but rather than reversing the helper list to produce a new list, the helper list is reversed in-place, and the reversed helper list is then returned. `Map-3` allocates no extra stack space and no extra cells for the resulting list, but it is written in an imperative style.

The fourth program, `map-4`, is simply `map-3` with the call to `reverse!` expanded in-line, in order to gain performance. The algorithm is now getting quite obscure. Like `map-3`, `map-4` is written in an imperative style.

What has not been mentioned already is that `map-3` and `map-4` are in fact incorrect! If the procedure passed to `map` captures the continuation and saves it in some data structure (as it may, in Scheme), and later reinvokes the continuation, the imperative implementations of `map` will cause illegal behavior in the system.

Consider for example the simple procedure shown in figure 4. The procedure `p` has the outward appearance of the identity procedure, but for a certain argument, `p` captures the continuation and saves the continuation object in a global variable. When we later invoke the continuation, already-defined data structures will be changed because the `map` procedure uses side effects. The problem is illustrated in figure 5.

One might ask: "But who would do that?" We believe that this question is irrelevant. In the presence of first-class continuations, assignments can have very unexpected effects on the program's behavior, and a functional style provides us

```
(define secret #f)

(define (p x)
  (if (eqv? x 3)
      (call-with-current-continuation
        (lambda (k)
          (set! secret k)
          x))
      x))
```

Figure 4: A nasty procedure to pass to an imperative `map`.

with a sense of security in the behavior of our programs. We will return to this topic in the next section.

```
> (define result (map-4 p '(1 2 3 4)))
result
> result
(1 2 3 4)
> (define foo result)     ;; "foo" and "result" are the same
foo
> (secret 7)              ;; should change "result" only
result
> result
(1 2 7 4)                 ;; as it should be
> foo
(1 2 7 4)                 ;; wrong!
```

Figure 5: A demonstration of the problem with an imperative `map`.

This Thesis

Conventional wisdom tells us that among the four implementations of `map` shown above, `map-4` is the most efficient in terms of running time and space usage. It is supposedly the most efficient program because (we are told) assignments are cheap and space allocation is expensive, and `map-4` clearly allocates no more space than it needs to, neither does it perform any extraneous assignments.

However, in the presence of a complex run-time system, and especially modern garbage collection technology, conventional wisdom is no longer a satisfactory basis on which to build one's programs, as the trade-offs in the run-time system affect the cost of primitive operations in the language. With the use of generational garbage collection algorithms, allocation and garbage collection have become inexpensive, while assignments, at least on stock hardware, have become more expensive. In addition, most compilers are able to make procedure calling (and hence recursion) inexpensive without affecting negatively other operations in the language.

It is therefore the objective of this thesis to challenge conventional wisdom and to quantify the actual tradeoffs in choosing one style over another for a particular computation. We perform our investigation in the context of several different garbage collection algorithms, observing how the tradeoffs change with the type of garbage collector. The language chosen for the investigation is Scheme.

In the `map` example, `map-4` is difficult to understand. The use of `reverse!` in `map-3` versus the use of `reverse` in `map-2` seemingly does not add much complexity, but the fact that side effects are used will make us stop and think about possible problems with shared data, and, as we noted in the previous section, a subtle bug was introduced by the use of side effects. One can therefore argue that `map-3`, too, is more difficult to understand than its functional counterparts.

In summary, a functional style is desirable, and we hope to be able to show that with sufficiently advanced compiler and run-time support, a functional style is a viable alternative to an imperative style.

In addition to determining the relative costs of the functional and imperative styles, we investigate style issues in a broader sense, looking for alternative ways to implement common algorithms and measuring the performance of the differing implementations.

A secondary goal of our investigation is to find how much the availability of first-class continuations to the programmer impacts the performance of a Scheme system. Scheme has been criticized for providing first-class continuations because they cause implementation difficulties, and as we saw above, the behavior of many procedures can become more difficult to understand in the presence of both side-effects and first-class continuations. We believe that having first-class continuations is a desirable feature of the Scheme language, and hope to show that the fact that the run-time library cannot always be coded imperatively "behind the scenes" does not impact performance as badly as some believe.

Our approach is perhaps a little unconventional. Most performance studies take a set of real programs (often tuned to particular language implementations) and then attempt to discover how a language implementation should behave to run all of these programs well. We, on the other hand, start with a particular language implementation which was designed to efficiently support a functional style without penalizing imperative programs too much. We then write benchmark programs in different programming styles and measure their behavior, in the last instance discovering tradeoffs in style for our language implementation. Since our Scheme system in many ways is typical of modern Scheme systems, we expect our findings to generalize to other Scheme implementations.

Previous and Related Work

It is a folk theorem in the Lisp community that an assignment operation, in the presence of a generational garbage collector, is as expensive as or more expensive than a `cons`, some say by as much as an order of magnitude more expensive. Yet, we are not aware of any published studies which measure the actual effect of programming style on performance.

Some relevant work exists on general performance measurements in Lisp and other symbolic languages, however:

1. Richard Gabriel's book [8] is the standard reference for performance studies of Lisp systems. The benchmarks are designed to measure particular aspects of a Lisp implementation, but do not consider alternative implementations of any particular program.[2]

2. David Ungar's dissertation [22] discusses performance measurements in the context of SOAR, a microprocessor specialized for running the Smalltalk language [9], and also in the context of a generation-scavenging garbage collector like the one used by one of our test systems.

3. Robert Shaw [19] analyzes a particular Lisp system on four large benchmarks in order to determine how data objects are created, used, and collected, with the ultimate goal of determining how a Lisp system should be implemented to support these kinds of programs well. For the most part, our test system (described in Chapter IV) is similar to his recommended system.

[2] For each of the benchmarks in the book, Gabriel presents a table of the primitive operations executed by the benchmark and their frequency. Assignments are not listed in the tables even when some benchmarks perform many of them, yet assignments are, as we will see, expensive in many modern Lisp systems. We believe that the omission reflects the notion that assignments are inexpensive.

4. Benjamin Zorn [27] has evaluated several garbage collection algorithms in the context of several large Lisp programs in order to determine the most efficient type of garbage collector. He reaches the rather unexpected conclusion that a generational mark-and-sweep algorithm is superior to the more common copying algorithms.

5. Paul Wilson has done work on improving the virtual memory locality of reference in garbage collectors [24] and on determining which cache configurations are most suitable for Lisp-like languages [25]. His conclusions are that the copying algorithm used by the garbage collector can have a large impact on locality of references, and that large and set-associative caches are very beneficial for performance.

Organization of the Thesis

The remainder of this thesis is organized as follows. Chapter II discusses garbage collection. Chapter III outlines the specific issues we are challenging and the behavior we wish to quantify. Chapter IV describes our test system. Chapter V presents methodology and all our test programs, and Chapter VI presents results from the tests and an analysis of the results. Chapter VII concludes the thesis and points to further work. The appendix contains the full source code for all our benchmarks.

CHAPTER II

AUTOMATIC STORAGE RECLAMATION

This chapter presents an overview of several common automatic storage reclamation algorithms, including those used by our test systems. We distinguish between reference counting and garbage collecting algorithms; the former will be described only briefly, as they are not of interest in this thesis.

Why Automatic Storage Reclamation?

Sophisticated programs often need to allocate memory (in the form of data objects) dynamically to hold data structures and, in some systems, procedures. The memory cannot be allocated statically either because the compiler cannot know at compile time how much memory will be needed or because the lifetime of a data object cannot be determined statically. Once memory can be allocated dynamically, however, returning the memory to the pool of free memory in a timely fashion can be a problem. The memory which was just allocated has to be freed when it is no longer used, yet determining when a chunk of memory is no longer in use can be difficult.

If the programmer is left to manage the freeing of dynamically allocated memory (i.e. if the language supports objects with controlled extent), two bugs are common: freeing an object several times, and not freeing an object at all. If an object is freed several times, a corruption of the free memory pool may result, often causing strange behavior in apparently unrelated parts of the program. If objects are not freed at all, memory will "leak" away until there's none left, causing the program to crash due to a lack of memory.

An alternative to programmer-managed memory is for the language to support objects with unlimited extent. Under this scheme, the user program merely allocates memory as it needs it, and once all references to an object have been destroyed, the programming language implementation is free to reclaim the memory occupied by the object and return it to the pool of free memory. Algorithms performing automatic reclamation is generally known as garbage collectors. The above-mentioned bugs with programmer-managed memory cannot happen when memory is automatically reclaimed, and in addition, having objects with unlimited extent in the language lets the language support general first-class data types, adding significantly to the expressive power of the language.

In summary, objects with unlimited extent and associated automatic storage reclamation are very desirable features in a programming language.

Reference Counting

In the reference counting scheme of automatic storage reclamation, each object has an additional data field which contains a count of the number of pointers (references) to the object. Whenever a reference to the object is stored somewhere, the reference count of the pointed-to object is incremented. Similarly, when a pointer to an object is destroyed (because a new value is stored in the location where the pointer exists), the reference count of the previously pointed-to object is decremented. When the reference count of an object drops to zero, the object is unreferenced and its memory can be returned to the pool of available memory; since returning the object to the free memory pool effectively destroys the object's contents, all objects to which the newly freed object refers must have their reference counts decremented as well, in general causing a recursive freeing process.

The advantage of reference counting is that it is simple and that the use of memory is minimized, since an object is freed as soon as it dies.

The disadvantages of reference counting are several, however. First, performance is poor because of the frequent manipulation of the reference count fields. Second, the reference count fields themselves occupy valuable space in the objects. If the count fields are small, they may fill up, and if a count field fills up, it cannot be subsequently decremented because there is no accurate count of the number of references to the object. The consequence of a filled-up count field is that the object and all the objects it references (recursively) cannot be freed, and we have a storage leak. Third, circular structures cannot be freed because all objects in a circular structure are referenced by at least one other object in the structure.

Several improvements on the basic reference counting algorithm have been suggested, for example Deutsch and Bobrow's deferred reference counting scheme [7], which delays the manipulation of the reference count fields and thereby gains much performance. Reference counting has also been augmented with more traditional garbage collectors to deal with the problem of circular structures and filled-up reference count fields.

Despite these improvements, reference counting does no longer seem to be considered a viable technique for general-purpose automatic storage reclamation.

Garbage Collection

All garbage collection algorithms work in roughly the same fashion. Starting from a small, fixed set of pointers called the roots, and traversing objects recursively, all live objects are found and somehow marked. Then, dead objects are reclaimed, and execution can continue. There are several noteworthy details.

First, there are usually very few roots, sometimes only one, and at most a few machine registers and global variables.

Second, a live object is defined as one which is (transitively) reachable from the roots, so it is illegal for the run-time system to store all pointers to a live object in non-roots or in dead objects if there is a chance that the garbage collector might run before a pointer to the live object can be stored in a root or in a live object. This invariant restricts to some extent the implementation: certain traditional compiler optimizations may be harder to incorporate into languages with garbage collection.

Third, the nature of "marking" and "reclaiming" varies widely between collector types, as we shall see.

Mark-and-Sweep Collection

The earliest garbage collectors were proposed for the LISP 1.5 system [17]; the collector for LISP 1.5 was of the type known as mark-and-sweep. A traditional mark-and-sweep collector marks all live objects by setting a bit in each reachable object; the collector then sweeps the entire heap from beginning to end and reclaims all non-marked objects by putting them on a list of free blocks. A more efficient variation of mark-and-sweep keeps a table of mark bits separate from the objects; this can make the sweeping phase much faster on modern hardware due to improved cache and virtual memory locality. A traditional mark-and-sweep collector will dirty all virtual pages with live objects on them, triggering a potentially large number of write-backs if memory is tight.

Mark-and-sweep, if implemented naively, can have very poor performance in virtual memory systems. The scan of the entire heap will result in all pages being brought into memory if they are not already there; this may cause the system to thrash. In addition, mark-and-sweep collection does not compact memory without extra work. Compaction is generally desirable because it speeds allocation and improves locality of reference.

Despite these problems, a new mark-and-sweep technique developed by Zorn [27] appears to have performance comparative to the generational collectors described below, and the non-compacting behavior of mark-and-sweep is sometimes desirable, especially in languages which were not initially designed to have garbage collectors, like many imperative programming languages.

Stop-and-Copy Collection

Rather than traversing all live objects and then sweeping the heap to find garbage, a copying collector copies each live object to a different area of memory as soon as the live object is found during the traversal phase. All memory which has not been copied is dead (garbage). The simplest form of copying collection is known as stop-and-copy. In a stop-and-copy collector, memory is divided into two equal-sized spaces, known as fromspace and tospace. All allocation is performed in fromspace. When fromspace is full, all live objects are copied from fromspace and into the initially empty tospace, with appropriate pointer adjustments. Then, fromspace and tospace are logically exchanged ("flipped"), and the program continues with the old fromspace as the new tospace, and vice versa.

Stop-and-copy collection corrects many of the problems of mark-and-sweep collection. First, only live objects are touched by the collector, so if there are entire pages of garbage, they will not be referenced or brought in from disk. Second, the heap is automatically compacted. Third, the traversal can be performed using only a constant amount of extra memory, utilizing a technique described by Cheney [4]; in Cheney's technique, tospace is used as a queue of objects still to be traversed.

The major problem with stop-and-copy collection is that only half of the available memory may be in use at any time; the rest is set aside for tospace. Also, with much live data, copying costs go up, as may the page fault rate.

Generational Collection

As it happens, objects in symbolic programming languages like Lisp and Smalltalk tend to either die young or go on to live for a long time. Exploiting this (entirely empirical) property, Lieberman and Hewitt [14] suggested that memory should be divided into generations and that each object should be located in a generation which reflects the age of the object.

In these generational collectors, objects are moved through generations as they get older, with the "young" generations being collected more often than the "older" generations. Since most objects die young, collecting the young generation (using a stop-and-copy method) takes little time, and an object surviving a collection in the youngest area can be moved on to the next generation, on the assumption that it will live for a while; since the next youngest generation is collected less often than the youngest generation, an object will be undisturbed in the next youngest generation for some time, giving it more time to die than it would have had in the youngest generation.

As an object gets older it is collected (and hence copied) less and less frequently. Contrast this with the traditional stop-and-copy collection scheme, in which a live object is copied once per collection, regardless of the object's age.

Because copying costs are reduced, the generational collectors can be much more efficient than the traditional methods. In addition, because the pages of the older generations are referenced relatively rarely, locality of reference improves, boosting performance even more.

Ungar [23] describes a form of generational collection known as generation scavenging. A generation scavenging collector has only two generation: "old" and "new". The "new" generation contains the young objects and is collected using the common stop-and-copy method. Each object contains a count of its age (in number of copies it has survived); when an object has survived a pre-set number of copies, it is copied into the "old" area, which is never collected.

An important factor in the performance of a generational collector is the tenuring policies used: when is an object tenured, i.e. moved from a younger to an older generation? Intimately related with this question is the problem of how to keep track of the age of an object. Explicit generation counts can be expensive because they occupy space and because they must be updated on every collection. An alternative approach is suggested by Shaw [19], who proposes a bucket brigade scheme: each generation is divided into buckets, and objects are copied from younger buckets to older buckets, finally being copied out of the oldest bucket to the "old" area. Buckets differ from generations: generations are collected at different intervals, but all buckets in a generation are collected at the same time. The bucket in which an object is located indicates the age of the object. A simplified version of the bucket brigade scheme uses only two buckets: the oldest bucket contains all objects which were copied during the last collection, and the youngest contains all objects created since the last collection.

A problem common to all generational collectors, regardless of configuration or policies, is that side effects require special treatment. Consider the collection of a young generation: to collect this generation, we copy all live objects from its fromspace into its tospace. If, at the time of the collection, there are references to objects in the young generation from objects in the older generations, then these latter references must be counted among the roots, or they will be not be updated correctly.

One simple solution to the intergenerational pointer problem is to consider all the objects in the older generation(s) as roots during a collection of the younger generation, scanning the older generations for references into the younger ones. However, this mostly-generational method can be costly: scanning the older generation(s) may incur a large number of page faults if memory is tight, and if the tenured area has much data in it, much time will be spent on scanning the tenured heap. A better approach is to keep track of a set (called the remembered set) of those objects in the older generations which may have pointers into the younger generation. The insight which lets us update the remembered set inexpensively is this: the only way an object in an older area may point to one in a younger area is if an assignment has been made to the object in the older area. This is true because all new objects are created in the youngest generation, and later copied to an older generation. Since an object can only point to objects older than itself when it is created, the object can point to younger objects only if it has been changed after it was created. We can therefore keep track of the set of remembered objects by checking, at every assignment, whether the value stored is a pointer to an object younger than the object which is stored into.

Variations on maintaining the remembered set exist, notably ones where the system does not remember the objects stored into but rather the locations in the old generations which were assigned to. Shaw [19] describes a scheme which uses the virtual memory hardware to keep track of pages which have been changed since the last collection; Sobalvarro [20] and Wilson [26] describe a software implementation of this scheme known as card marking.

No matter how the intergenerational pointer handling is implemented, it adds cost and complexity to assignments. Besides that and the implementation complexity (which is not very high), there are few problems with generational collection schemes.

CHAPTER III

STYLE ISSUES

In this Chapter, we present the style issues we want to investigate along with illustrative examples. The full benchmark suite is presented in Chapter V.

Imperative Implementation vs. Functional Implementation

As the `map` example in Chapter I showed, we can sometimes choose to implement an operation imperatively or functionally, without affecting the surrounding algorithm. In the implementation of `map`, we could exchange `reverse!` for `reverse` without making any other changes to the `map` procedure.[1] Since the use of `reverse!` makes the algorithm harder to understand, `reverse` is to be preferred style-wise. We need to investigate how much, if anything, the extra allocation done by `reverse` will cost us in execution time.

Imperative Iteration vs. Functional Recursion

Consider again the `map` procedure in Chapter I. The procedure `map-1` recurs down the input list and then builds the resulting list on its way back out of the recursion. The program is simple, but, as pointed out earlier, it uses stack space proportional to the length of the input list. The procedure `map-4`, on the other hand, uses constant extra space, but is much harder to understand than the `map-1`. What we need to know is how much, if anything, the recursion in `map-1`

[1]Note, however, the subtle error this introduced into `map`.

will cost us in execution time over the iteration in `map-4`.

A related issue is the trade-offs between the two functional procedures `map-1` and `map-2`, as they both use extra space in proportion to the length of the input list.

Explicit Stacks vs. Implicit Stacks

Almost any programmer will tell you, when asked, that "iteration is faster than recursion, because you can manage your own stack." The idea is that the recursion adds so much overhead to the "real" data saved on the stack that it is more efficient to allocate some memory to use for a stack and manage the pushing and popping of data words explicitly. Many algorithms texts will present techniques for converting a recursive algorithm into an iterative one with an explicit stack, or at least will present an iterative implementation of `quicksort`.

Programmer-managed stacks are not very nice, though. The amount of stack space needed often depends on the size or even organization of the input to the procedure using the stack, and hence the programmer, if he wants his procedure to be general, must find ways to implement arbitrary-sized stacks efficiently. Such mechanisms tend to obscure the algorithm being implemented. In addition, converting a recursive procedure to an equivalent iterative one can be tedious and difficult.

Recursion, with its implicit stack, is therefore considered a much nicer (but, as already pointed out, slower) programming technique than iteration with an explicit stack. We must therefore investigate if, in the context of a decent Scheme implementation, recursion is in fact significantly more expensive than iteration with an explicit stack.

Creating New Data

As the `map` example in Chapter I showed, new data structures can be created in several ways, either by starting with the base case (the empty list) and then extending the structure (with `cons`), or by building up a partial structure (with `cons`) and then changing the existing structure to add new elements (with `set-cdr!`).

Another example of different ways of creating data is illustrated by the two vector addition procedures shown in figure 6. The imperative procedure `vector-add-1` creates the resulting vector first and then assigns values to the slots of the vector as the values are computed. The functional procedure `vector-add-2`, on the other hand, accumulates the values in a list and then calls the primitive `list->vector` which returns a vector containing the values in the list.

```
(define (vector-add-1 v1 v2)
  (let* ((len (vector-length v1))
         (v3  (make-vector len)))
    (let loop ((i 0))
      (if (< i len)
          (begin (vector-set! v3 i (+ (vector-ref v1 i)
                                      (vector-ref v2 i)))
                 (loop (+ i 1)))
          v3))))

(define (vector-add-2 v1 v2)
  (let loop ((i (- (vector-length v1) 1)) (l '()))
    (if (>= i 0)
        (loop (+ i 1)
              (cons (+ (vector-ref v1 i) (vector-ref v2 i)) l))
        (list->vector l))))
```

Figure 6: Imperative and functional vector addition.

While the functional vector addition procedure allocates extra space proportional to the length of the resulting vector, it is not clear that it is much more expensive than the imperative procedure. The issue of how to create new data therefore falls squarely within the scope of this thesis.

Rebinding vs. Assignment

Consider a typical imperative-style program `sieve-1` which finds the first `n` prime numbers using the Sieve of Erathostenes, shown at the top of figure 7. The assignments in this program can be removed by adding two parameters to the loop procedure, giving us the functional program `sieve-2` at the bottom of figure 7.

Conventional wisdom tells us that the `sieve-1` is cheaper because parameter passing is "expensive" and because the assignment statements do not get executed often, indicating extraneous overhead in the loop. It is our task to discover the actual costs here.

Explicit Data Structures vs. Implicit Data Structures

Some procedures can be made to operate on vastly different data structures by hiding not only the implementation of a data structure, but also its meaning. Hiding the meaning of a data structure can be accomplished by the use of generators. In this context, a generator is a procedure of no arguments which when called will return the next element of a data structure, for some appropriate definition of "next". Using generators, we can proceed to write general comparison routines like the one in figure 8, which takes as its arguments two generators which when called generate the next element of their respective data structures, or `#f` when there are no more elements.

We can now apply `same?` to e.g. binary trees, arbitrary n-ary trees, or indeed to arguments which have different structure, as long as their generators behave according to the postulated rules.

The alternative to using generators is to either generalize our (explicit) data structures sufficiently to make it possible to write general comparison routines, or to write special-case comparators when we need them. In either case, our programs become larger and we get more code to debug and maintain.

Another interesting data structure is the stream [1]. A stream is in essence a list whose tail has not yet been computed; it is an on-demand data structure. Streams are useful data abstraction devices and can among other things be used to model state without assignment. There is a fair bit of overhead associated with implementing on-demand (lazy) evaluation in Scheme, however, and we would like to see how much more expensive streams are than the corresponding "eager" data structures. The Sieve of Erathostenes is again a good example: we can implement a stream of prime numbers which will deliver prime numbers on demand. Pitted against the stream implementation would be the traditional implementation of the Sieve, using a vector for keeping track of previous primes.

```
(define (sieve-1 n)

  (define primes '())
  (define count 0)
  (define (loop trial)
    (cond ((= count n)
           primes)
          ((prime? trial primes)
           (set! primes (cons trial primes))
           (set! count (+ count 1))
           (loop (+ trial 1)))
          (else
           (loop (+ trial 1)))))

  (loop 2))


(define (sieve-2 n)

  (define (loop trial primes count)
    (cond ((= count n)
           primes)
          ((prime? trial primes)
           (loop (+ trial 1) (cons trial primes) (+ count 1)))
          (else
           (loop (+ trial 1) primes count))))

  (loop 2 '() 0))
```

Figure 7: Implementations of the Sieve of Erathostenes. The procedure `prime?` has been omitted.

```
(define (same? g1 g2)
  (let ((x1 (g1))
        (x2 (g2)))
    (cond ((not x1) (not x2))
          ((not x2) #f)
          ((= x1 x2) (compare g1 g2))
          (else #f))))
```

Figure 8: Generator-based comparator.

CHAPTER IV

THE TEST SYSTEMS

Introduction

Our test system is "Larceny," a native-code Scheme system for the SPARC architecture [21], developed by William Clinger and the author. The compiler used by Larceny is "Twobit," an experimental compiler for MacScheme [15].

Larceny was designed with an emphasis on simplicity and portability, and is written mostly in Scheme. It incorporates a straightforward generation-scavenging garbage collector; a stack cache; a small set of fundamental data types; simple data representations; a software timer to enable us to write time-sliced and threaded programs; fast generic arithmetic; and, in Twobit, a relatively advanced compiler. Each of these features are described in greater or lesser detail below.

For the purposes of the research leading to this thesis, we have implemented two more garbage collectors in order to be able to experiment with the effects of collector technology on performance. These collectors are described below as well.

The version of Larceny used for this thesis is v0.14, which is still a little unfinished and in need of tuning in odd places, none really critical to the results in this thesis. For a more complete description of Larceny, see [10].

The Stack Cache

Scheme provides the programmer with access to the current program continuation through the procedure `call-with-current-continuation` (abbreviated `call/cc`). In general, since the continuation object so obtained is a first-class object, continuation frames cannot be allocated solely on a stack. A simple way around this problem is to allocate all continuation frames directly in the heap. However, due to the frequency with which frames are allocated in a language such as Scheme where procedure calls are very frequent, allocating frames on the heap leads to poor cache locality and an increased number of garbage collections, in turn leading to poor performance.

In order to get good performance (but still allow for a reasonably fast implementation of `call/cc`), we provide a stack cache, a bounded area of memory in which frames are allocated and deallocated in a stack-like manner. This cache may overflow during deep recursions. When the cache overflows, or when a continuation is captured, each frame in the cache is spilled to the heap individually, forming a linked list in the heap just as if the frames were allocated in the heap in the first place. Later, when the cache underflows or a continuation is reinstated, a single frame is restored from the heap and placed in the cache. This stack cache scheme is known as the "incremental stack/heap strategy" and is described more fully by Clinger et al [6].

In Larceny, the stack cache is also flushed before every garbage collection, since having all live data in the heap simplifies the garbage collector. The performance impact of this, if any, is unclear, and would appear to depend very heavily on the codes being run.

It is important to observe that underflow checking takes zero instructions. The trick is to keep a dummy frame at the bottom of the stack; the return address in this frame points to the underflow handler, so when the cache underflows, the handler is called automatically.[1]

Data Types and Representations

Larceny has six fundamental data types: fixnums (immediate exact integers), non-fixnum immediates (characters, booleans, the empty list, and other odds and ends), pairs, vector-like structures, bytevector-like structures, and procedures. The three low bits of a word are used for the tag, with fixnums and immediates having two tags each. The low two bits of a fixnum are zero, letting us use the tagged arithmetic instructions on the SPARC. Any type information needed beyond the tag bits is stored in the header of the pointed-to data structure. Pointers may point to the start of a data structure only. All data structures except pairs consist of a header word (an immediate) followed by zero or more data words. The header word has a length field and additional type information, if any. Pairs are headerless and occupy two consecutive words. Bytevector-like structures contain untagged data, such as strings and executable code. All other data structures contain tagged data only.

The Garbage Collectors

The principles of garbage collection were presented in Chapter II. This section presents the specific garbage collectors used by Larceny.

Larceny's default garbage collector is a simple generation-scavenging collector, as described in Chapter II, with the exception that the tenured area is sometimes collected (not just expanded, as in Ungar's collector).

[1]Due to a bug in the current compiler which prevents the auto-detection scheme from working properly, stack frame pops are currently more expensive than they have to be because we perform explicit underflow checking. Our procedure calls are therefore more expensive than they need to be.

The tenuring policy employed in Larceny is simple. If, after an ephemeral collection, the amount of live data in the ephemeral area exceeds a user-set limit, then the next collection will tenure all live data.

Larceny also has a static area for keeping unchanging data, mostly the code vectors of procedures in the run-time system. The static area is not used by the test programs in this thesis.

The remembered set for intergenerational pointers in Larceny is known as the transaction area. The transaction area contains tagged pointers to objects in the tenured area which in turn may contain pointers into the ephemeral area. Transactions are added to the transaction area when an assignment is performed on an object in the tenured space; the value being assigned is not considered. Since several transactions may exist for the same tenured object, the collector goes to some lengths to compact the transaction area and to avoid scanning a tenured object more than once.

What Larceny's collector lacks in terms of age precision it makes up for in simplicity. It is also straightforward to make slight modifications to the collector to implement e.g. a two-bucket bucket brigade as described in Chapter II; future research will show whether this is in fact worthwhile.

For the investigations in this thesis we employ two additional garbage collectors.

The first experimental collector is a straight stop-and-copy collector. Under this collector, all data lives in the ephemeral area; there is no tenured area. When the ephemeral area fills up, an ephemeral collection is performed, and execution continues.

The second experimental collector is a mostly-generational collector. This collector differs from the default collector only in that it does not maintain a remembered set. Rather, during ephemeral and tenuring collections, the entire tenured area is scanned for references into the ephemeral area.

Generic Arithmetic

In Larceny, some operations on fixnums (notably addition and subtraction) are done in-line using the tagged arithmetic available on the SPARC architecture. Overflows as well as other fixnum operations and all operations on other numeric types are handled in assembly-language subroutines (which in turn call Scheme support procedures for some operations).

All test programs in this thesis use fixnum arithmetic exclusively.

The "Twobit" Compiler

The "Twobit" compiler combines a number of relatively simple and well-known optimizations with a friendly target architecture to generate good code for typical Scheme programs. The target architecture is the MacScheme architecture, a virtual architecture originally implemented as a bytecode interpreter in MacScheme. In Larceny, a back-end assembler takes MacScheme assembly language as input and produces SPARC object code as output.

The MacScheme architecture is an accumulator-based architecture with a large number of auxiliary registers and an instruction set specialized for executing Scheme. During procedure calls, as many parameters as possible are passed in registers, with extra arguments passed in an additional list.

Currently, the optimizations performed by Twobit include assignment elimination and recognition of local definitions, lambda lifting, optimization of calls to known procedures, register allocation, and register targeting.

Desirable optimizations not performed by the current version of Twobit include stack frame reuse and shrink-wrapping [5], type propagation, and type inferencing.

The SPARC Architecture

The SPARC architecture [21] is a 32-bit reduced-instruction-set (RISC) non-proprietary architecture. Like most RISC architectures, the SPARC has a simple instruction set with all instructions being 32 bit wide, a large general-purpose register file (32 32-bit integer registers and 32 32-bit floating-point registers; the floating point registers can be grouped to form wider floating-point registers), and a flat address space. Memory accesses are via load and store instructions only, and there are only two addressing modes: register-plus-offset and register-plus-register. Immediates and offsets in load and store instructions are 13-bit signed, and constants larger than 12 bits must in general be built up using two instructions. All branches are PC-relative with a 24-bit signed offset. There is also an “jump absolute” instruction and a PC-relative “call” instruction.

Branches, jumps, and calls are delayed: the instruction following the control-flow instruction is executed whether the flow of control is changed or not. In the case of the branch instruction, the following instruction can be annulled by setting a bit in the branch instruction: if the branch is not taken, and the annul bit is set, the following instruction is not executed.

The SPARC also has several features which are not very common among current RISC architectures. The primary of these features is the register windows. Register windows provide an efficient means of caching a small number of stack frames on-chip. During a procedure call, the program can execute a special instruction to save the current registers and to get a fresh set of registers. At the time of the return, the program executes the inverse instruction, restoring the old set of registers.

Another unusual feature is the architectural support for tagged integer arithmetic. The tagged-arithmetic instructions will perform addition or subtraction, but will also check that the "tags" (low two bits) of the data being operated on are both zero; if they are not (or if they are, but the operation created an overflow condition), the overflow flag can be set or a trap can be taken. These instructions make it simpler to implement fast generic arithmetic.

Finally, the semantics of the annul bit are unusual; other architectures which let the instruction following the branch instruction be annulled will annul the delayed instruction if the branch is taken.

Larceny uses the tagged arithmetic instructions, but does not trap on a tag fault; rather, the overflow flag is checked explicitly. The reason for this decision was that the cost of a trap on the SPARC was deemed too high to allow us to implement truly efficient generic arithmetic.

Larceny does not use the register windows, however. Register windows are fine if a program does not have "deep" calls, because the saved windows can fit entirely in the microprocessor's on-chip cache of saved windows. Once the call depth goes beyond 6 or 7 (on current implementations of the architecture), however, the window cache overflows and will have to be spilled. The overflow causes a trap, which is expensive, and since the trap handler has no knowledge of the procedure whose window has to be spilled, it has to save all the registers in the window, which is needlessly expensive. The bottom line is that once the procedure call depth gets interesting, using register windows becomes more expensive.[2] Furthermore, since the register windows imply (but do not require) a callee-saves discipline rather than a caller-saves discipline, implementing tail recursion properly becomes a little more difficult. Finally, we anticipate porting Larceny to other architectures. A straightforward port appears more feasible if we do not use the windows in the SPARC version.

The Sun SPARCstation

Our test systems have been SPARCstation machines from Sun Microsystems. There are currently two major classes of SPARCstations, the Sparc 1 and the Sparc 2. These two classes are again subdivided into models, depending on clock speed, memory capacity, number of processors, and so forth.

Our test machine was a SPARCstation ELC (Sun 4/25, a Sparc 2 class machine) with 8 MB of physical memory, 32 MB of virtual memory, a 33 MHz clock, and running SunOS 4.1.2.

The following is an informal discussion of some of the interesting (in the context of this thesis) implementation characteristics of the Sparc 2; specific numbers are taken from [12]. The Sparc 2 has a 64 KB direct-mapped unified instruction and data cache. The cache is write-through with no-write-allocate; a cache miss on a write will not bring a new line into the cache. The cache lines are all eight words (32 bytes) wide. The cache has a write buffer which is two doublewords deep; when this buffer fills up, the CPU is stalled until the buffer

[2] Some very early benchmarks comparing Larceny with a commercial Common LISP which uses register windows as well as with the Sun C compiler on a simple recursive benchmark indicated a performance advantage of a factor of 7 in favor of Larceny in both cases. Since Larceny has slower calls than either of the systems we were comparing it with, we conclude (as have others before us) that the register windows can be very expensive on deep recursions.

drains again.

In the absence of a cache miss, a single-word load takes two cycles, and a single-word store takes three cycles. If there is a cache miss, there is an additional miss penalty. In the case of a load, the penalty is 24 or 25 cycles to bring the entire line into the cache, and the CPU stalls until the line has been fetched in its entirety. In the case of a store, the miss penalty is four or five cycles.

Effects of the Cache Strategy on Performance

A 64 KB cache is much too small to hold all of the ephemeral area of a generational collector, so on the face of it, cache locality is not going to be very good for programs which use large amounts of memory.

Because the cache is no-write-allocate on the Sparc 2, however, allocation itself has no impact on how data is cached: a datum is brought into the cache only when read, never when initialized or modified.

Also, when a chunk of memory is allocated for a data structure, that memory is likely to be uncached due to effects of the garbage collector, so for new data, we will always incur a miss penalty during initialization. It may therefore be that explicit prefetching is beneficial. The following outlines some of the issues and tradeoffs; we will not investigate prefetching any further in this thesis.

Consider the `cons` operation: it allocates two words for the cell and initializes each word. The initialization takes a total of 14-16 cycles, because each store takes 7-8 cycles (3 cycles to store plus 4-5 cycles penalty). When we later take the `car` of the cell, another cache miss will occur, for a total of 26-27 cycles for the one reference. However, the entire cell becomes cached, so another `car` or `cdr` operation takes only two cycles, and storing a value into the cell takes only three cycles.

As a concrete example, to create a cell and then take its `car` and its `cdr` once each, takes (on the average) $15 + 26.5 + 2 = 43.5$ cycles. Now, if we instead prefetch the cache line in which the words are allocated; this would then cost $26.5 + 6 + 2 + 2 = 36.5$ cycles, roughly a 20% savings.

Furthermore, if two cells are allocated next to each other, the cost for the two allocations plus two `car` and two `cdr` operations would be $(26.5 + 6 + 2 + 2 + 2 + 6 + 2 + 2)/2 = 24.2$ cycles per cell because the second prefetch would hit the cache; the cost for the non-prefetching case is still twice the cost of a single allocation if the second cell is allocated before the first is referenced, and in this case, prefetching saves almost 50%. For larger structures or for more pairs in a row (up to four, for an 8 word cache line), the savings would be even greater, for as little as 18 cycles per cell for four cells.

We don't really know anything about the reference or allocation patterns, so the prefetch, while not wasteful as such, may in fact evict useful data, causing the program to run slower overall. We would also expect that the more times a cell is referenced after it has been created, the smaller the benefits from prefetching (since reference costs dominate, even when the cell is cached).

As caches get bigger, more memory will be in the cache most of the time, reducing the effects of tricks like prefetching. On the other hand, cache misses will get more expensive as machines get faster. Also bear in mind that the cost of a `cons` is more than the cost of simply storing the values in memory, and that an implementation would have to have very streamlined allocation to get significant speedup from prefetching. Finally, the benefits of prefetching are strongly dependent on the exact details of the memory hierarchy of the computer running the program; the Sparc 1, for example, differs quite a bit from the Sparc 2 in its memory access costs.

Considering the possible benefits from prefetching, and all the tradeoffs, the issue is worth an investigation (but again, not here).

CHAPTER V

METHODOLOGY AND BENCHMARKS

We first outline which parameters of the test systems we are able to manipulate. We then present the quantities which will be interesting to either derive or measure for each program. Finally, we present all of the benchmark programs and all directly derivable measurements for each.

Test Parameters

Even though our benchmark programs are small, essentially toy programs, we can vary several conditions under which they are run. The goal of varying the conditions is of course to determine how different running conditions affect the running time and other measurable quantities of the system.

The test parameters come in several classes.

1. The amount of live data can affect the time it takes to perform a garbage collection. While large amounts of live data will presumably penalize the non-generational collectors more than the generational ones, we need to determine what the actual costs are.

   The amount of live data in the test systems is about 2.75 MB at startup time. Since data occasionally gets tenured between benchmarks, some benchmarks may run with more live data than others, but the variations are expected to be small.

2. The heap sizes primarily affect the frequency of garbage collection, but can also have an impact on the page fault rate. For the generational collectors, a large ephemeral area is expected to give a longer interval between collections, enabling more objects to die, and thereby shortening the time it takes to perform the garbage collection.

3. The size of the stack cache influences the cost of heavily recursive algorithms, as stack cache overflow and underflow may be a significant cost. We would like to know something about how big it is necessary to make the stack cache to make recursive algorithms viable even on fairly large inputs.

4. The size of the input affects the running times in several ways. Large inputs often behave differently from small inputs because more temporary data is live during a collection (causing collection time to increase), or because the stack cache overflows, or because hardware cache locality is destroyed.

5. The location of objects affects the cost of primitive operations in the system; under the generation-scavenging collector, for example, a side effect is several times more expensive if the assigned-to object is located in the tenured area than if it is located in the ephemeral area. However, with the data residing in the tenured area, there's more space in the ephemeral area for temporary or new objects, resulting in less garbage collection activity.

   To measure the differences incurred by tenured objects, is therefore important to make sure that the test programs are run with test data in both the ephemeral and tenured areas.

6. The amount of physical memory available to our test system can have a significant impact on the page fault rate, which, as we will see, is one of the

primary factors in determining how long a program takes to run. We should therefore run our test programs both with enough physical memory to contain the entire heap, and with little enough physical memory to make page faults a significant factor; in the latter case, the differences between the different garbage collection algorithms will be more apparent.

How well do each of these configurations fit realistic environments? While it is true that modern workstations usually have significant amounts of physical memory, they are not (as are our test systems) idle, and in many cases at least some page faulting will occur. Also, our test system with its less than 3 MB of live data is small compared to many realistic systems; the author routinely runs Scheme programs requiring as much as 30 MB of memory.[1]

Quantities to Measure

The interesting variables in our measurements can be divided into two main groups: derivable and empirical. The values of the derivable variables are the expected costs of particular operations in the system, that is, the number of CPU cycles it takes to perform a `cons`, given that we know what instructions are executed by the operation.[2] These values are interesting mostly in the context of the empirical variables, which tell us the average real cost of the corresponding operations. Empirical measurements can also be made for larger programs; for these programs, deriving the expected costs is rarely practical.

We first present the derivable variables of interest together with the values of

---

[1]Whether a 30 MB heap is typical of "normal" programs is another issue altogether.

[2]While it may seem that the derivable variables are not variables at all, but rather constants, we prefer to think of them as variables since a different version of Larceny will likely have different values for the derivable variables.

some of them, based on the current version of Larceny. We then present the empirical variables; the values of the empirical variables is the subject of Chapter VI.

Derivable Variables

The most interesting derivable variables are the costs of the primitive operations in the test systems. In our case, these operations are `cons`, `car` and `cdr`, `set-car!` and `set-cdr!`, `vector-ref`, and `vector-set!`. Also of interest in our benchmarks is the speed of generic arithmetic, procedure call and return, and the cost of a stack frame creation. The expected costs for these operations are summarized in table 1. In this table, the best case time assumes no cache misses, and the worst case assumes as many cache misses as possible. In neither case are instruction fetch times taken into account. For the assignment operators, the data are assumed to be in the ephemeral area for the best case and in the tenured area for the worst case (for those collectors which have a tenured area).

Table 1: Derived costs (in cycles) of primitive operations in Larceny; the times in parentheses are for the mostly-generational and stop-and-copy collectors where there is a difference between those and those of the generation-scavenging collector.

Operation	Best case	Worst case
`cons`	18	50.5
`car`/`cdr`	3	27.5
`set-car!`/`set-cdr!`	19 (7)	79 (33.5)
`vector-ref`	20	69
`vector-set!`	36 (27)	177.5 (79)
Fixnum addition	3	30.5
Stack frame creation (n registers)	$15 + 3n$	$39.5 + 6n$
General procedure call + return	10+4	10+29.5
Call to known procedure + return	2+4	2+29.5

Empirical Variables

This section lists the quantities which were observed during each test run and from which we derive our conclusions. In general, we believe wall clock (elapsed) time to be more interesting than CPU time, mainly because the elapsed time differs significantly from the CPU time in the presence of page faults.

1. Running time, both elapsed and CPU time. The time it takes to run a program is of course the bottom line from the programmer's point of view.

2. The time spent in the garbage collector. As the collection algorithms are varied, we observe how (or if) collection times change, both in absolute terms and as a fraction of the total running time.

3. The number of data words allocated, collected, and copied during a test run. These quantities will tell us something about how memory-intensive a program is. Also, given the time spent in a garbage collection and the number of words copied, we are able to determine the efficiency of each garbage collector in terms of bytes copied per second.

4. The number of intergenerational pointers recorded. This will tell us how heavily a combination of an algorithm and test data placement uses "expensive" side effects.

5. The kinds of collections performed. For example, if a program causes a large number of tenuring collections, it will tell us something about the lifetimes and amount of its data. Such knowledge is valuable in the analysis of the program. An unexpectedly large number of expensive collections is also indicative of tight memory.

6. The number of page faults incurred by a run. Given this number, we can calculate the approximate cost of a page fault, and the number of page faults for a particular memory configuration may tell us how memory is being used.

   Relationships between the size of the heap and the number of page faults are also interesting.

7. The number of stack frames flushed to the heap and the amount of heap memory allocated to these frames tells us how the stack is being used. With specially designed programs (see the next section) we can find the actual cost of recursion when the stack is spilled and when it's not.

Unlike many other studies, we do not measure pause times incurred by garbage collection. While pause times are of interest in interactive settings, we are currently more interested in the overall behavior of our garbage collectors in extremely allocation-intensive benchmarks. We can calculate the average time per collection, however.

The Benchmark Programs

In the following, we present all the benchmark programs and the system parameters which we want to measure using a particular program or group of programs. The source code for the benchmark programs is included in the appendix.

`Memparams` and `Stackparams`

The `memparams` and `stackparams` benchmarks determine some basic parameters of the implementation, for use in the analysis of the other benchmarks.

`Memparams` determines the cost of the `cons`, `set-car!`, `set-cdr!`, `vector-ref`, and `vector-set!` operations. Where it is relevant, we measure both best-case (all memory references hit the cache) and worst-case (all memory references miss the cache) times.

The `cons` test works by consing up a number of pairs, all of which immediately become garbage. The number of pairs allocated must be large enough to outweigh any hardware cache locality on all the test systems, giving us a basis for finding the worst-case time for a `cons`. The best-case time for `cons` is not very interesting on our test systems because the caches are so small (64 KB); if the caches were significantly larger, the best-case time would be more relevant.

The tests for `set-car!` and `set-cdr!` come in four flavors, depending on where the word being mutated is located: in the cache or out of it, and in the tenured area or in the ephemeral area. The hard part is guaranteeing that a pair being mutated is not in the cache. Since the caches on all our test systems are write-through and no-write-allocate, we know that creating a pair with `cons` will not bring it into the cache. Consequently we `cons` 10,000 times and reference each pair to fill the cache with garbage cells; we then `cons` another pair and use the last pair so obtained in our tests, assured that it is not in the cache. The other cases are straightforward: to get a pair into the cache, just access one of its fields. To get a pair into the tenured area, do a tenuring collection. To ensure that a pair in the tenured area is not in the cache, perform `cons` 10,000 times after the tenuring collection and access a field of each of the newly-allocated pairs to fill the cache with garbage. Our tests are regrettably still not perfect: a garbage collection due to a full tenured area will bring the cell being worked on into the cache, polluting our measurements.

Caching considerations similar to those for `set-car!` and `set-cdr!` apply to the testing of `vector-ref` and `vector-set!`; however, for the vector tests, the cases are simpler. Since the cache on the test system is direct-mapped, a single long vector referenced cyclically will ensure that we have uncached data, and a single short vector referenced cyclically will ensure that we have cached data.

`Stackparams` determines the cost of using a stack cache of a given size on a particular recursion; this cost is interesting because the cost per frame goes up significantly when the stack cache overflows. The `stackparams` benchmark also determines the cost of flushing a frame as well as the cost of restoring it.

The cost of flushing the stack cache and of restoring the stack frames can be found by capturing the continuation at the outset of the recursion and then recurring until we reach the base case. At this point, we reinstate the continuation, bypassing any restoring of stack frames. With appropriate adjustment for the procedure calling overhead (which can be found by performing the same test with a shallow recursion, avoiding flushing the cache), we are left with the cost of flushing the frames. The cost of restoring the frames is simply the total running time minus the running time of the stack flushing benchmark.

`Reverse` and `Reverse!`

The `reverse` procedure takes as its input a list and returns a list which is the reverse of the input list. The new list is newly allocated. `Reverse!`, on the other hand, reverses the input list in-place by changing the `cdr` fields, and returns the last pair of the input list. After the link-reversal, the last pair of the input list is the first pair of the reversed list.

Since `reverse` allocates a new pair for each list cell, it uses more memory than `reverse!` (which runs in constant space). What we wish to measure is the relative cost of `reverse` to `reverse!` under various conditions. What we hope to find are some general rules for when each should be used for better performance. Since the algorithms are very similar in complexity, we expect the performance ratios to closely resemble those of `cons` and `set-cdr!`; the latter ratios are derived from the `memparams` benchmark.

In particular, we are interested in knowing whether there is significant advantage to using `reverse!` rather than `reverse` in creating fresh data structures.

The `Boyer` Benchmark

The `boyer` benchmark was written by J. Moore and B. Boyer and a Lisp version was published by Gabriel in 1985 [8]. Our version of `boyer` is a straightforward translation of Gabriel's version to Scheme.

The `boyer` benchmark is a simple theorem prover and does a lot of procedure calls and operations on pairs: `car`s, `cdr`s, and `cons`es. We use `boyer` as a basic garbage collection benchmark in order to determine some parameters of our test systems. What we are primarily looking for is the comparative costs of a garbage collection, such as how many words a particular test system can copy per second during a collection. `Boyer` is a good benchmark for this because it runs for a while and has more realistic data allocation patterns than many of the other test programs.

Two Implementations of `Append`

The standard Scheme procedure `append` takes one or more arguments, all lists, and returns a new list which is the concatenation of all the input lists. The resulting list shares structure with the last of the input lists. For our purposes, `append` is restricted to two arguments.

`Append` is typically implemented in one of two ways.

The first variation, `append-1`, is recursive. It recurs down the first input list, and when the end of the list is reached, the procedure returns up the call chain, consing up a new list on the way, with the second argument to `append` as the `cdr` of the first list cell.

The second variation, `append-2`, iterates over the first input list, creating new list cells as it goes along, and changing, using `set-cdr!`, the previously made list cell to point to the new one, and making the last cell point to the second input list.

Both variations of `append` have equivalent semantics. They both allocate the same number of pairs. `Append-1`, however, uses stack space proportional to the length of the first input list; `append-2`, on the other hand, performs a number of `set-cdr!`s proportional to the number of elements in the first input lists.

The relative cost of the two should be roughly equivalent to the ratio of the cost of creating a small stack frame to the cost of doing a `set-cdr!` in the ephemeral area. On truly deep recursions there will be other effects as well, as the stack cache is of fixed size and may overflow.

We will use the `append` benchmark to study the cost of functional recursion relative to imperative iteration.

The Four Implementations of `Map`

In Chapter I, we presented four implementations of the `map` procedure. Ignoring for the moment that the two imperative implementations `map-3` and `map-4` had incorrect semantics due to obscurities with the use of `call/cc`, the `map` benchmarks can be used to tell us whether the added complexity of the imperative procedures gives significant performance advantages over the functional ones.

The `map` benchmarks put the `reverse` and `append` benchmarks into a larger context, as `map-2` and `map-3` pit `reverse` against `reverse!`, and `map-1` and `map-4` compare algorithms which are similar to `append-1` and `append-2`, respectively. We therefore expect these benchmarks to behave in accord with their simpler relatives.

Merge Sort

Merge sort [13] appears to be the sorting method of choice in the Lisp community. Merge sort is attractive because of its guaranteed O(n log n) running time, because it can be made to be stable, and because it is simple to understand and implement.

We are measuring the running times of two implementations of merge sort. The first method, called `sort-1`, is a functional implementation of merge sort. The second algorithm, called `sort-2`, copies the list and then performs an in-place (imperative) sort of the copied list.

Both procedures take two arguments: the list to be sorted, and a predicate which imposes a total "less-than" ordering on the set of elements in the list. Since the predicate is an arbitrary user-defined procedure, `sort-2` has the same problems as do the imperative implementations of `map` (see Chapter I).

Both algorithms have the same time complexity, but the space complexities are different: `sort-1` allocates O(n log n) list cells, whereas `sort-2` allocates only O(n) list cells, and unless the list copy made by `sort-2` forces a collection which copies some of the new cells into the tenured area, no transactions will be recorded for side effects either. It would not be surprising, therefore, if `sort-2` is somewhat faster than `sort-1`.

The `Quicksort` Program

`Quicksort` [13] is literally the textbook example of a recursive algorithm which should be written iteratively using an explicit stack. For the purposes of testing this hypothesis, we have implemented five different versions of `quicksort`.

The first version, `quick-1`, is the standard recursive algorithm (in all cases, the choice of the median is the first element of the list to be sorted).

The second version, `quick-2`, uses an explicit stack in the form of a list onto which we `cons` unfinished computations.

The third version, `quick-3`, uses a doubly-linked list for a stack, which lets us avoid a bit of memory allocation. The "stack pointer" points into the middle of the list; the live stack cells are found by following the links in one direction, the free cells are found by following the links in the other direction. If the stack needs to be extended beyond the free list, new cells are simply added at the end; when the stack shrinks, the movement of the stack pointer both pops the stack and adds the stack element to the free list.

The fourth version, `quick-4`, allocates a single vector for the stack; this vector is large enough to handle the worst case (which is O(log n) on the condition that the smallest subfile after a partition is always sorted first).

The fifth version, `quick-5`, is the same as `quick-2`, but keeps the stack in a local variable rather than passing it a parameter.

All five programs sort a vector in-place; the `quicksort` benchmarks measure primarily the cost of recursion versus using an explicit stack and the cost of various stack implementations. `Quick-5` also measures the cost of rebinding versus assignment.

As the listings in the appendix show, the iterative versions are significantly more complicated than the straightforward recursive implementation, with `quick-3` being the most complicated one; it will therefore be good news if the recursive algorithm has performance approaching the performance of the iterative algorithms.

Three Implementations of `Fib`

The recursive Fibonacci procedure is a well-known program which is sometimes used to measure the performance of procedure calls. We will use it as another measurement of the tradeoffs of implicit and explicit stacks.

The stack depth in `fib` is proportional to the magnitude of its single argument, which, because of `fib`'s exponential behavior, has to be quite small. It is therefore reasonable to pre-allocate a vector and use it as the stack for unfinished computations. Another approach is to cons unfinished computations onto a list which functions as the stack.

This gives us three programs: `fib-1` is the original recursive procedure; `fib-2` is the iterative variation using a vector for a stack, and `fib-3` is the iterative variation using a list for a stack. We can also compare `fib-2` against `fib-3` to find the cost of a `vector-set!` relative to the cost of a `cons` in the presence of a very large amount of allocation.

Three Implementations of `Samefringe`

The various versions of the `samefringe` benchmark all solve the same graph problem: given two acyclic digraphs where the leaves are labeled, discover whether the two graphs have the same fringe (sequence of leaves).

We will call the programs `samefringe-1` through `samefringe-3`. `Samefringe-1` is the stupid implementation: it traverses both graphs, creating two lists of labels. It then compares the lists. `Samefringe-2` traverses both graphs simultaneously, using two explicit stacks (in the form of lists of traversal states) to keep track of nodes which have to be returned to. `Samefringe-3` does not operate directly on the graphs, but is rather handed two generators which will produce the next leaf value of their respective graphs when called.

For comparison purposes we have also implemented a version of `samefringe-1` in C; this version uses `malloc()` and `free()` to manage memory explicitly. We call this version `samefringe-C`. The object of this comparison is to discover, on a rudimentary level, the performance differences between a garbage collected system and one which does memory management explicitly.

The `samefringe` benchmarks are useful for several purposes. `Samefringe-1` is a nice garbage collection benchmark. By comparing `samefringe-3` with `samefringe-2`, we can compare a generic generator-based program to a program which knows about the shape of the data structure.

Two Implementations of `Vector-Add`

The two variations on `vector-add` were shown in Chapter III. This benchmark can tell us something about the cost of creating new data and whether a functional style is viable. Bear in mind that the functional program, `vector-add-2`, allocates three times as much memory as necessary: the resulting vector to hold n words occupies only n+1 words, but the intermediate list occupies 2n words as well.

The `Sieve` of Erathostenes

Two simple implementation of the Sieve of Erathostenes were shown in Chapter III; those programs exhibited the differences between using rebinding and using assignment to update state variables.

The `sieve` benchmark is more generally useful for our purposes and ideal for comparing explicit and implicit data structures. At least two ways of using explicit data structures come to mind: using a vector for the previously computed primes, and using a list. But the previously computed primes can also be represented by implicit data structures. Two approaches are streams and generators.

A generator is a procedure of no arguments which returns successive values on successive calls. The internal workings of the generator are undefined, and we can think of the generator as a function which computes the next element of a sequence given the previous ones. However, the generator has state, and if several consumers share a generator, each generated value will be seen by only one consumer.

Streams, on the other hand, have no state. Instead, a stream behaves much like a list, and the consumer of a stream can get consecutive elements from the stream using the `head` and `tail` operations. Because streams are stateless, several consumers can consume the same stream and see all the elements.

Streams and generators are both lazy, in that they only compute as much of the answer (in our case, a list of prime numbers) as has been requested.

`Sieve-1` is the imperative-style `sieve` program presented in Chapter I, and `sieve-2` is the corresponding functional style program. These programs mainly test the differences in the cost of rebinding vs. assignment. `Sieve-3` uses a vector to keep track of previous primes and their squares, while `Sieve-4` uses a generator approach, and `sieve-5` is a streams implementation. We compare these programs to find tradeoffs in data representation.

Nondeterministic Finite Automata

A nondeterministic finite automaton is a simple computational engine capable of recognizing a regular language. The language recognized by the automaton is described by means of a regular expression; for example, the expression `(((ab)|(ac))*c)` describes the set of strings ending in `c` and with a prefix consisting of an arbitrary number of repetitions of the tokens `ab` and `ac`. The strings `acc` and `abababacabc` are both in the language accepted by the automaton.

The expression given above is ambiguous in the sense that if the automaton has only one character of lookahead, it will not know, when an `a` is detected, whether it will find `ab`, `ac`, `aa`, or the end of the input. There is no notion of the automaton backtracking in case it guessed incorrectly; the recognition of the particular input, and therefore the expression itself, are said to be nondeterministic. Indeed, the automaton may reject the string even if it is in the language of the automaton. We can think of the automaton as always guessing correctly, however.

One can implement NFAs as computer programs. In doing so, we choose to specify the semantics of the NFA so that it is guaranteed to accept the input string if the string is in the language being recognized, i.e. our program simulates an automaton which always guesses correctly. This can be done by stacking choice points in such a way that when the automaton fails to recognize a string, it can backtrack to an earlier choice point and try a different route.

We compare two approaches to stacking choice points. The general strategy of both approaches is to implement each state in the state machine by a procedure, and to implement state transition by a procedure call which passes the rest of the input as a parameter. For example, if state 1 has a transition to state 2 on the input symbol `a`, we can write

```
(define (state1 input)
  (if (eq? (car input) 'a)
      (state2 (cdr input))
      ...))
```

The traditional approach to stacking choice points is to use the program stack and the implicit stacking of choice points inherent in non-tail-recursive procedure calls; we call this program `recursive-nfa`. For example, the choice `(a|b)` could be implemented as

```
(define (state1 input)
  (and (not (null? input))
       (or (and (eq? (car input) 'a)
                (state2 (cdr input)))
           (and (eq? (car input) 'b)
                (state3 (cdr input))))))
```

A different approach is to pass an explicit failure procedure (or failure continuation) to each state procedure. Upon failure to recognize the input, the failure continuation is invoked with no arguments, and computation effectively backtracks to the last choice point. The choice point `(a|b)` would look something like this:

```
(define (state1 input fail)
  (cond ((null? input) (fail))
        ((eq? (car input) 'a)
         (state2 (cdr input) (lambda ()
                              (if (eq? (car input) 'b)
                                  (state3 (cdr input) fail)
                                  (fail)))))
        ((eq? (car input) 'b)
         (state3 (cdr input) fail))
        (else
         (fail))))
```

This benchmark measures the difference between using the recursive style (which uses the program stack to keep track of choice points and hence will save some extraneous state information on the stack), and the iterative continuation-passing style which uses no stack space but which creates a lot of higher-order procedures. Both are quite heavy on the procedure calls. Essentially, this benchmark explores the differences in cost between recursion and iteration, between two kinds of implicit stacks (because, after all, the nesting of failure procedures is nothing but a stack), and between two equivalent but very different programming styles.

Summary of Benchmarks

Table 2 summarizes the benchmarks and the style issues each benchmark is used to measure. The `memparams` and `stackparams` benchmarks are not included in the table as they are useful only for finding system parameters used in the analysis of the style issues.

Table 2: Summary of benchmarks and style issues.

Legend: A–GC performance; B–Implicit vs. Explicit Stacks;
C–Recursion vs. Iteration; D–Assignment vs. Rebinding;
E–Imperative vs. Functional; F–Creating Data Structures;
G–Implicit Data Structures.

	A	B	C	D	E	F	G
Boyer	X						
Samefringe	X						X
Reverse	X		X		X	X	
Append	X		X		X	X	
Map	X		X		X	X	
Sort					X		
Quicksort		X	X	X			
Fibonacci		X	X				
Vector-add					X	X	
Sieve				X	X		X
NFA		X	X				X

CHAPTER VI

MEASUREMENTS AND ANALYSIS

In this Chapter, we present the measurements from running the benchmark programs which were presented in Chapter V, along with an analysis of the results. The results are presented and analyzed by benchmark rather than by style issue, and some conclusions are drawn along the way. A common analysis is presented in the last section.

The resolution of the timer on our system is 10–15 ms, so to get reliable results, we have run most benchmarks many times and divided the total time by the number of iterations. For those benchmarks which were run only once, any difference less than 20 ms should be treated as suspect; all numbers presented here (except averages) are rounded upward to the nearest 10 ms.

Each benchmark is presented with a description of its input and the number of iterations the test was run, followed by the results and an analysis of the results.

Basic Benchmarks

The basic benchmarks `memparams` and `stackparams` measure the costs of basic operations in the system. The basic operations are `cons`, `set-car!` and `set-cdr!`, `vector-set!`, `vector-ref`, and stack frame creation. While we can calculate the rough costs of these operations by looking at which instructions are executed (and indeed these estimates were presented in Chapter V), the costs estimated are a priori not likely to be very helpful. Estimated costs cannot properly account for hardware cache behavior, page faults, or operating system interruptions, all of which contribute to the real cost of an operation.

The Cost of a `Cons`

In many functional-style programs the `cons` operation is among the most common and often-used ones. The cost of performing a `cons` is therefore important in the overall performance of a particular system.

We define the cost of a `cons` to be a fraction of the elapsed time it takes to allocate a large number of pairs which immediately become garbage; i.e. copying costs for the pair are not included in the cost. It is clear that this cost is not constant, but rather depends on several parameters of the test system: the amount of physical memory, the ephemeral heap size, and (for some collector types) the amount of live data. When the number of pairs allocated is sufficiently large, garbage collection costs play a part in the calculated costs; this is deliberate.

The `cons` benchmark allocated 1,000,000 pairs with fixnums as their contents; all pairs became garbage immediately after being allocated.

The cost of a `cons` for the test system is summarized in table 3, and the relationship between the size of the ephemeral area and the cost of a `cons` is shown graphically in figure 9.[1]

It turns our that cost of a `cons` is roughly proportional to the number of (major) page faults incurred by the running of the benchmark. In the case of the generation-scavenging collector, no major page faults occurred during the run, and the times for all memory sizes are all low. The mostly-generational collector

[1]Please note the different scales on the y-axes in this and subsequent figures; since the stop-and-copy collector has only a large ephemeral area, its y-axis scale is larger than for the two other collectors.

Table 3: Summary of `cons` costs, in nanoseconds per operation.

Collector	Min	Max	Average	Median
Generation scavenging	1070	1640	1352	1440
Mostly-generational	2710	9720	4893	3900
Stop-and-copy	22320	62100	44974	49730

experienced some page faults due to garbage collection and the cost of a `cons` is quite a bit higher for this system than for the generation-scavenging collector (and the variations in time are large). Finally, the stop-and-copy collector experienced tremendous page faulting, which is reasonable considering that the test system had little memory and that the `cons` benchmark in the stop-and-copy system will have referenced all of the pages allocated to the heap by the start of the second collection. The cost of a `cons` in the stop-and-copy system is therefore very high.

The size of the ephemeral area affects the frequency of collections in all three systems. A small ephemeral area results in frequent collections and a large area in more seldom collections, so a large ephemeral area is effective in reducing garbage collection costs. However, for the generational collectors, a small ephemeral area appears to have better locality of reference than a large ephemeral area, reducing the number of page faults but increasing the time spent garbage collecting. The extra time in garbage collection is not noticeable in the generation-scavenging collector, but the mostly-generational collector has to scan the tenured area more times. Since garbage collection is more expensive than poor locality for the mostly-generational collector, a larger ephemeral area is beneficial for the `cons` benchmarks; the generation-scavenging collector has better performance with a smaller ephemeral area, however (see figure 9), with locality of reference being the major contributing factor to performance.

In order to find the real cost of a `cons`, we subtract the overhead of the benchmark loop from the running time. The overhead was measured to be about 400 ns per `cons` operation for all test configuration; the numbers in table 3 are asjusted for the loop overhead, as are other `memparams` costs in subsequent tables.

Table 3 shows that the cost of a `cons` on our test hardware, without even accounting for the cost of copying the created cell, depends very much on the kind of garbage collector. The performance advantage of the generation-scavenging collector is partly that it has a reference pattern which does not cause page faults, but also that it does not have to scan a large number of tenured objects to find references into the ephemeral heap during an ephemeral collection.

From tables 9 and 1 we see that the actual cost of a `cons` under the generation-scavenging collector is slightly better than the derived worst-case cost (which was calculated to be 50 cycles, or 1500 ns). The actual cost of a `cons` under the two other collectors is much worse than the derived worst-case cost, however; we conclude that for the mostly-generational and stop-and-copy systems, the cost of a `cons` is dominated by other effects, in this case page faults due to allocation and collection.

The Cost of `Set-Car!` and `Set-Cdr!`

The `set-car!` and `set-cdr!` operations execute nearly identical instruction sequences, the only difference being the offset in the last store instruction. We will therefore measure only the cost of `set-cdr!` and expect the cost of `set-car!` to be identical.

The `set-cdr!` test executes 1,000,000 stores into the `cdr` of a pair. This pair is either cached or uncached and is located either in the tenured or the ephemeral

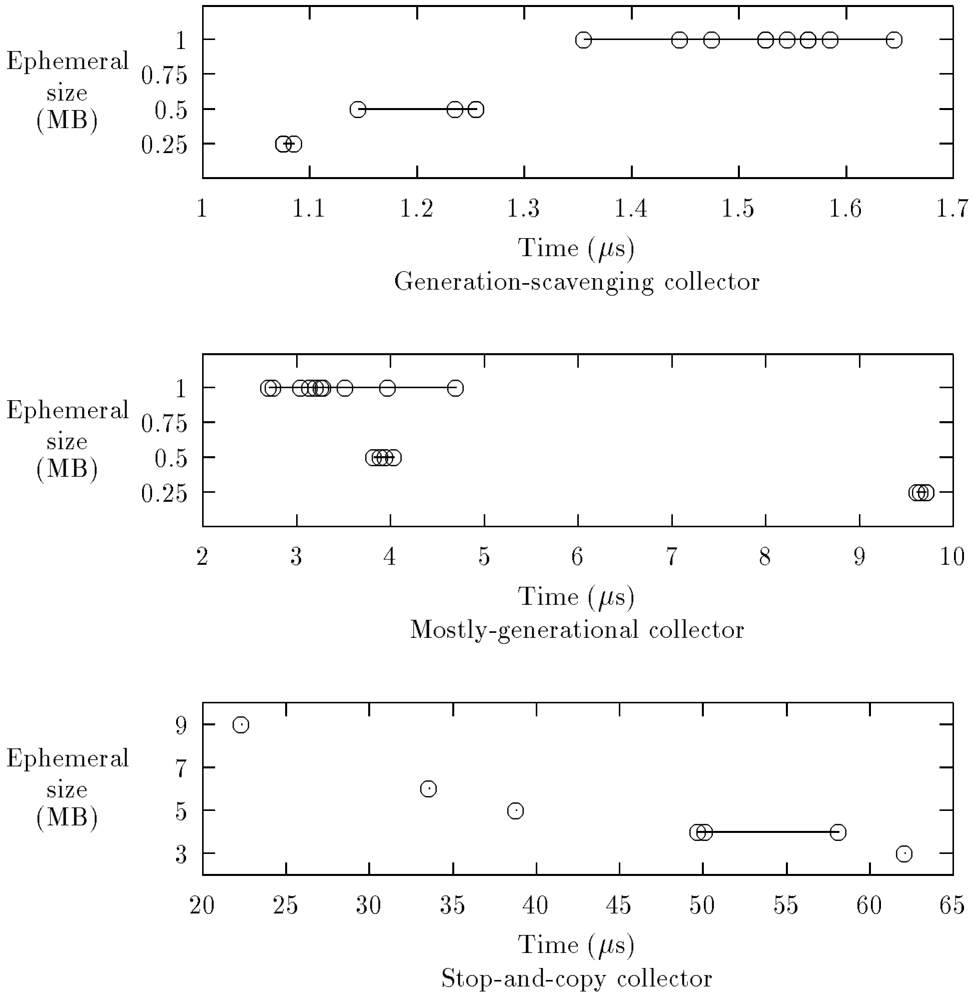

Figure 9: Cost of a `cons` as related to the ephemeral area size.

area. How we ensure the location and cache status of a pair was explained in Chapter V.

Figure 10 summarizes the cost of the `set-cdr!` operation when the pair being operated on is located in the ephemeral area. As we can see (and as expected), the cost of storing into an uncached pair is higher than that of storing into a cached pair; the difference is about 100 ns, or 3 cycles. The difference corresponds well with our assumptions about the hardware as outlined in Chapter IV.

The cost of a `set-cdr!` under the stop-and-copy and mostly-generational collectors varies from about 210 ns to 390 ns. The cost of a `set-cdr!` under the generation-scavenging collector varies from 690 ns to 830 ns; from these tests it would appear that a store is from 2 to 3 times more expensive under the generation-scavenging collector than under the other two collectors. Again, this is for data located in the ephemeral area. The higher cost under the generation-scavenging collector is accounted for by the instructions which are executed to check whether a transaction must be recorded; this check is performed by a subroutine and is therefore somewhat costly.

The measured times correspond well to the expected times as derived in ChapterV; the measured ratios of the costs of the `set-cdr!` operation under the generation-scavenging to the cost under the mostly-generational and stop-and-copy collectors also correspond well to the expected ratios.

Figure 10 plots the cost of an operation against how much ephemeral memory is allocated. As was the case for `cons`, there appears to be a correlation, at least for cached data, between the amount of ephemeral area and the cost, in that the cost goes up with the amount of ephemeral area allocated. The cost increase is very slight, however, and may be of no useful consequence. The inverse relationship seems to hold for uncached data; again, the increase is very slight.

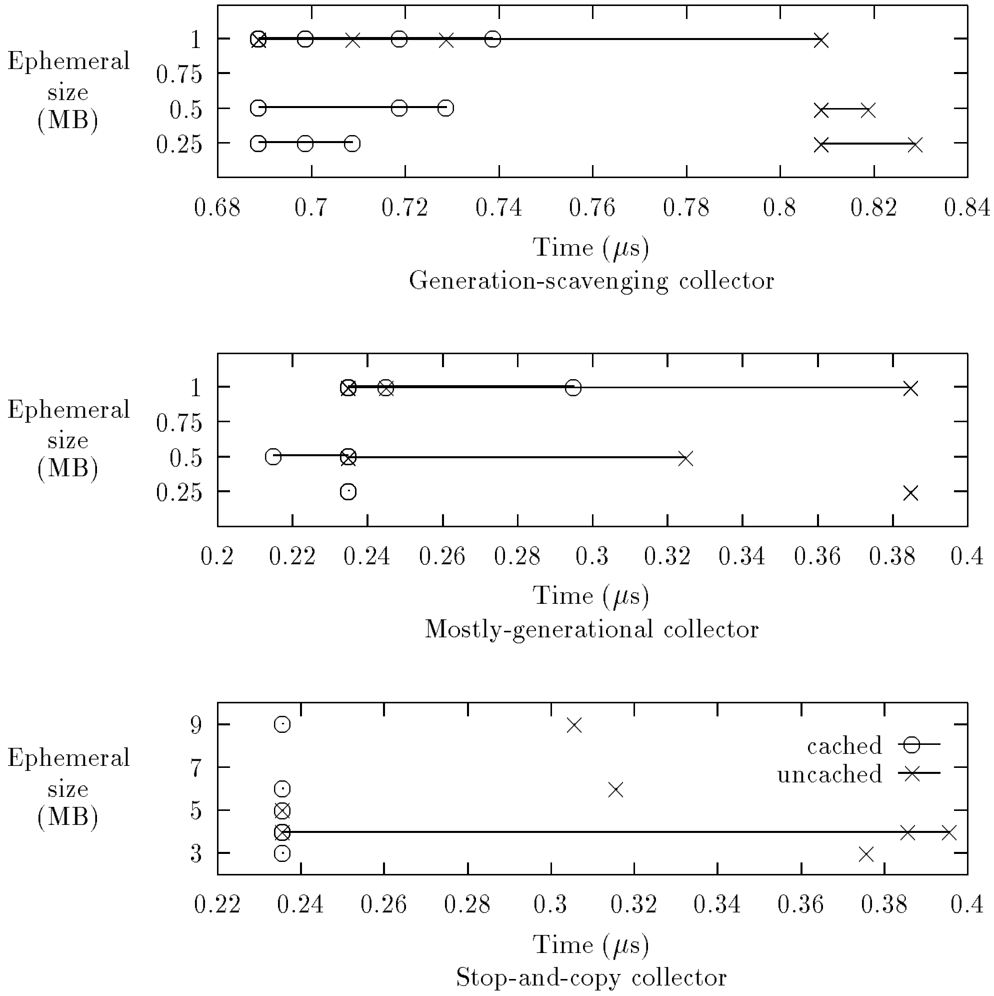

Figure 10: Cost of a `set-cdr!` in the ephemeral area.

Figure 11 summarizes the cost of the `set-cdr!` operation when the pair is located in the tenured area; no times are shown for the stop-and-copy collector because it does not have a tenured area. Again there is the expected division between cached and uncached data; however, the difference between a cached and an uncached reference appears to be much greater for the generation-scavenging collector than what is reasonable given the machine model. A closer examination of the test results reveals that the uncached `set-cdr!` benchmarks experienced a number of page faults. The cause of the faults is not known.

The cost of a `set-cdr!` under the mostly-generational collector does not change when the data moves into the ephemeral area; this is as we would expect. Under this collector, the cost of a `set-cdr!` is still in the range from 210 to 390 ns. The cost of a `set-cdr!` under the generation-scavenging collector, however, is very much higher when the data is located in the tenured area than when it is located in the ephemeral area. If the data is located in the tenured area, the cost of a `set-cdr!` varies from about 1650 ns to 6900 ns (in one pathological case); most of the time, the cost lies in the range from 1650 ns to 3500 ns.

In other words, a store into an object in the tenured area can be from less than twice as expensive to over five times as expensive as storing into an object in the ephemeral area. Also, a `set-cdr!` under the generation-scavenging collector can be as much as 16 times as expensive as a `set-cdr!` under the two other collectors, on the average five to nine times as expensive.

The observed times for a `set-cdr!` in the tenured area correspond well with the expected costs which were derived in Chapter V, although the observed worst-case times (3500 ns) are worse than the expected worst-case times (2400 ns). The poor observed behavior is most likely due to garbage collection overhead and paging effects: for each assignment into a tenured object, a transaction is recorded, eventually causing one or more garbage collections as the transaction area fills up, and also causing page faults as the transaction area is accessed.

Table 4 summarizes the costs of `set-cdr!`; again, these numbers are adjusted

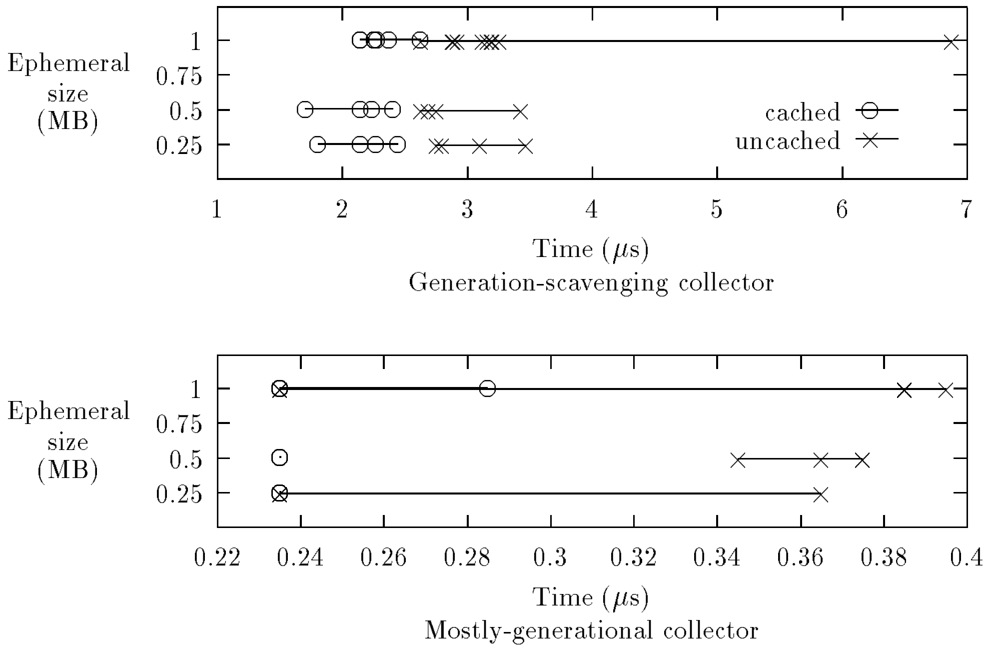

Figure 11: Cost of a `set-cdr!` in the tenured area.

for the loop overhead.

Table 4: Summary of the cost of a `set-cdr!`, in nanoseconds.

Collector	Ephemeral data				Tenured data			
	Min	Max	Avg	Median	Min	Max	Avg	Median
Cached Data								
Generation-scavenging	509	639	528	509	1619	2479	2245	509
Mostly-generational	55	205	76	55	55	195	111	55
Stop-and-copy	56	196	77	56	—	—	—	—
Uncached Data								
Generation-scavenging	509	629	525	509	2419	7769	4261	509
Mostly-generational	55	185	74	55	55	245	116	55
Stop-and-copy	206	236	215	206	—	—	—	—

The Ratio of `Cons` and `Set-Cdr!` Costs

The ratios of the costs of `cons` and `set-cdr!` are presented in table 5. The cost of a `cons` is on the average 65 times as expensive as a `set-cdr!` under the mostly-generational collector, and under the stop-and-copy collector, the cost ratio is 308. Under these collectors, therefore, it is unreasonable to allocate new memory when old memory can be updated in-place instead. For the generation-scavenging collector, however, a `cons` is over twice as expensive as a `set-cdr!` in the ephemeral area, but has less than half the cost of a `set-cdr!` in the tenured area. Under this collector it is therefore desirable to allocate new memory rather than changing a tenured pair, and even if an pair is in the ephemeral area, updating it is not tremendously much faster than creating a new object.

Table 5: The cost ratios of `cons` to `set-cdr!`.

Collector and data placement	Min	Max	Average
Generation-scavenging (ephemeral)	2.1	2.6	2.6
Generation-scavenging (tenured)	0.7	0.2	0.4
Mostly-generational	49.3	47.4	65.6
Stop-and-copy	400.7	263.5	308.7

The Cost of `Vector-Ref` and `Vector-Set!`

The vector benchmarks were run 50 times on vectors of length 20,000 (for the uncached test) and 200 times on vectors of length 5,000 (for the cached test), each test totaling 1,000,000 vector operations.

As one would expect, the cost of a `vector-ref` is affected only by whether the element being referenced is already cached or not; the times are virtually identical for all three collectors. Table 6 summarizes the cost of a `vector-ref`.

The times for `vector-ref` compare well to the expected costs, being on the average less than twice as expensive as the best-case estimate, but not nearly as bad as the worst-case estimate.

Table 6: Summary of the cost of `vector-ref`, in nanoseconds.

	Min	Max	Average	Median
Cached	816	856	827	826
Uncached	866	886	867	866

The cost of the `vector-set!` operation depends on the collector type being used. Figure 12 shows that when the vector being operated on is located in the ephemeral area, the costs under the mostly-generational and stop-and-copy collectors are mostly identical, with a cost of 950–1100 ns if the vector element has been cached.[2] Under the genration-scavenging collector, the cost is 1350–1450 ns.

Interestingly, a `vector-set!` is only 33% more expensive under this collector than under the other collectors (for data in the ephemeral area), compared to the 200-300% difference for `set-cdr!`. The reason is fairly clear: while the intergenerational check dominates the `set-cdr!` operation, a `vector-set!` performs much more work which does not change between collectors: the vector type must be verified with an extra tag check, the tag of the index operand must be checked, and a range check must be performed.

The large variations in the observed times for the generation-scavenging collector when the data is in the tenured area are due to as many as 20 garbage collections triggered by the filling up of the transaction area; the same phenomenon was observed for the `set-cdr!` benchmark.

When the vector being operated on is stored in the tenured area (the results for which are shown in figure 13), the times for the mostly-generational collector stay the same, but the times for the generation-scavenging collector increase noticeably, from 1400 ns on the average to about 2800 ns.

The cost of a `vector-set!` doubles under the generation-scavenging collector when the datum being operated on moves into the tenured area. Also, a `vector-set!` under the generation-scavenging collector is as much as three times as expensive as the same operation under the stop-and-copy collector.

The times for `vector-set!` fall in the range of the expected costs, typically on the best-case side of the spectrum.

---

[2] Uncached numbers are not presented for the `vector-set!` benchmark.

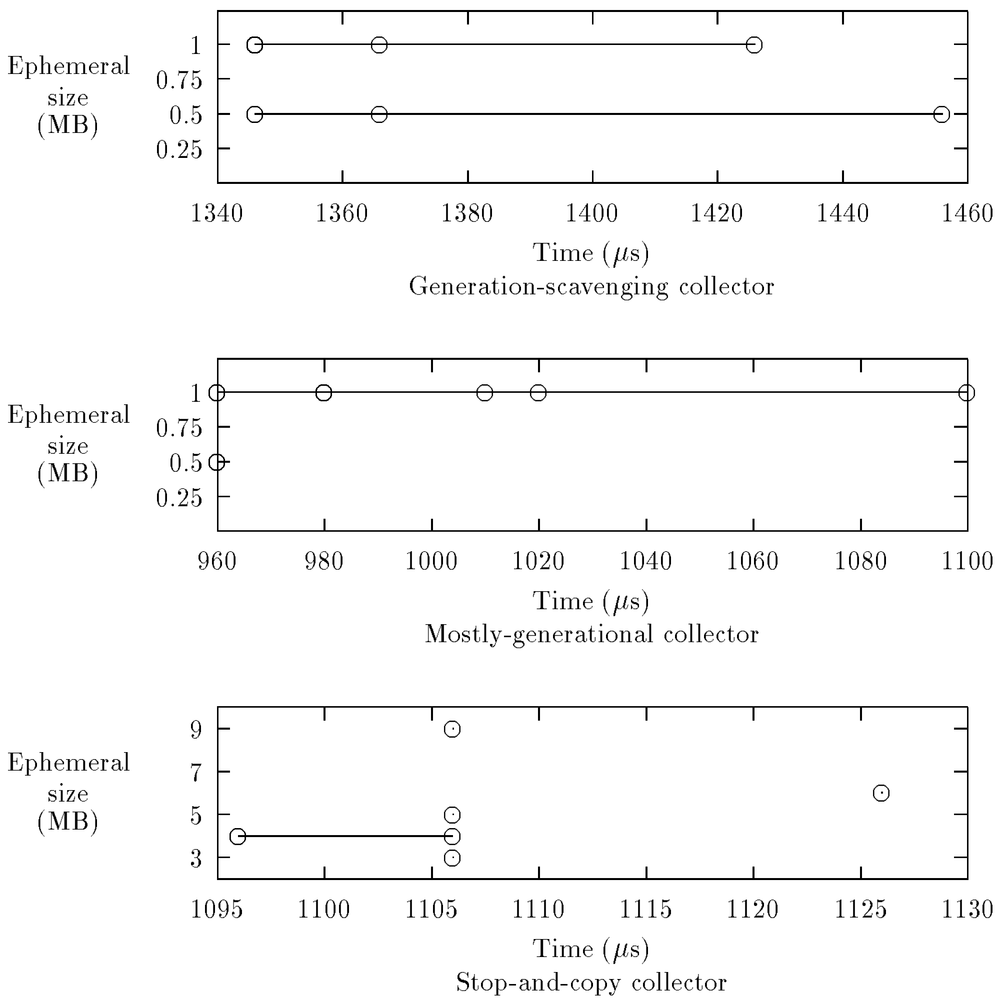

Figure 12: Summary of the cost of a Cached `vector-set!` in the ephemeral area.

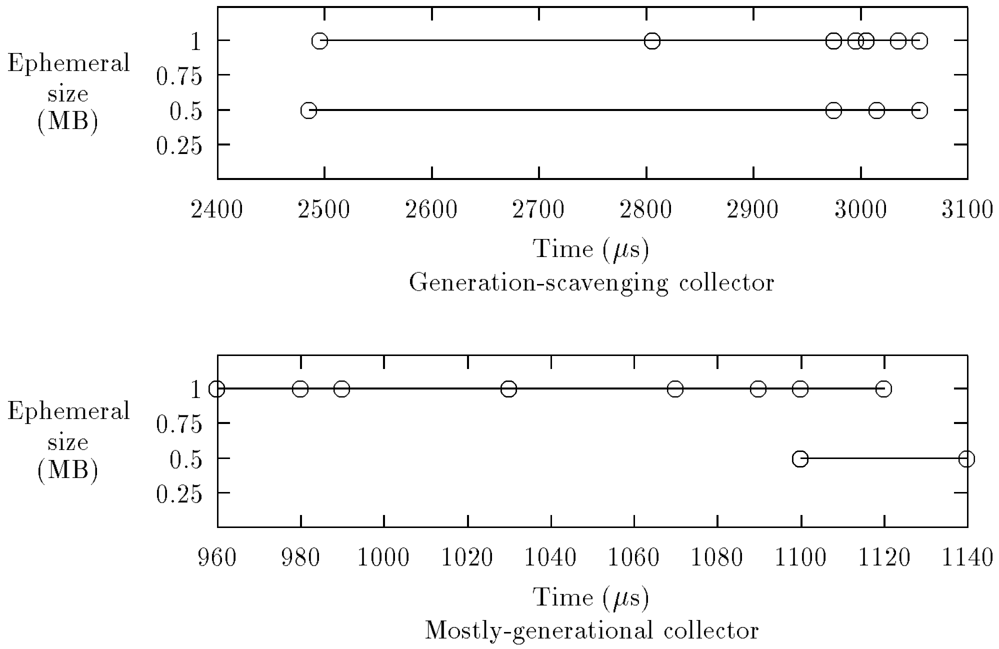

Figure 13: Summary of the Cost of a Cached `vector-set!` in the tenured area.

The Cost of Creating a Stack Frame

Costs associated with the creation, flushing, and restoring of stack frames are interesting because given these costs, we can ideally predict how programs written in various ways will behave.

The `stackparams` benchmark measures some basic costs which let us in turn calculate the stack frame costs. The costs we calculate are for one particular stack frame size only (7 words); the costs are summarized in table VI. The costs for the stack frame flush include the amortized costs of garbage collections incurred by the allocation of heap space for flushed frames.

The benchmarks were run to a depth of 10,000 frames 100 times, and to a depth of 100 frames 10,000 times.

caption[Summary of Stack Frame Costs] Summary of stack frame costs, in μs per frame.

	Create	Flush	Restore
Generation-scavenging	1.5	9.8	3.2
Mostly-generational	1.5	18	3.2
Stop-and-copy	1.5	159	31

From table VI we see that the stack frame creation time is independent of the collector type, which is as expected. The time observed for a creation is about midway between the expected best-case and worst-case times as calculated in Chapter V (for five saved registers). Moreover, the creation of a frame of this size (which would be typical for the inner loop of a recursive procedure like `map-1`) is about the same as the cost of a `cons` in the generation-scavenging collector, but less than the cost of a `cons` in either of the other collectors. Since procedure calls, returns, and the popping of frames which are still on the stack are inexpensive operations, a recursive algorithm which does not cause a stack flush should be about the same cost as an iterative algorithm which creates a list which is finally reversed; that is, `map-1` should be as fast as `map-2` on lists that are not long enough to cause stack flushes.

Once the stack is flushed, the cost per frame goes up a great deal. Table VI shows that the total cost of creating a frame which is flushed is 14.5 μs in the generation-scavenging collector, almost an order of magnitude worse than the non-flushed cost. For the other collectors, the cost goes up even more when the stack is flushed: a factor of 15 for the mostly-generational collector, and a factor of almost 130 for the stop-and-copy collector. The time for restoring a stack frame is particularly high under the latter collector; the poor performance is due to extremely poor page locality in this collector which is again due to a large number (254) of expensive collections. The two other collectors had many more collections (1354 each), but locality was very good under both collectors.

It is clear that with large inputs, recursive algorithms can be much more expensive than iterative ones; in particular, simple algorithms (where the stack overhead will be costly relative to "useful" work) will likely suffer. However, if the cost of stack flushing and restoring can be reduced, recursive algorithms may be practical even for large inputs.[3]

Garbage Collection Benchmarks

The garbage collection benchmarks enable us to find the copying speeds of each of our garbage collectors. Also, for the `boyer` benchmark, we can observe how the collectors behave under more varied workloads than represented by many of

[3]We have designed a more efficient implementation of the incremental stack/heap strategy, but have not made it part of the test system yet. The new strategy avoids copying the frames into the heap and should also have much-improved cache locality. It remains to be seen how much of a difference this makes.

the other benchmarks.

Copying speed is by no means the only measurement of garbage collector performance; indeed, in a system with little memory, the elapsed time for some collectors will have been spent mostly handling page faults. The copying speed tells us how the collector will behave in a system with enough memory to avoid page faults, however.

The `Boyer` Benchmark

The `boyer` benchmark was run once for each memory configuration; the results are shown graphically in figure 14. Most notable is that the mostly-generational collector was consistently fastest for this benchmark. This is unexpected because `boyer` is often regarded as a garbage collector benchmark, and we'd expect the generation-scavenging collector to do best. The performance loss in the generation-scavenging collector can be ascribed to several factors:

1. In the course of execution, `boyer` performs 126,000 assignments to closed-over variables; in the generational collectors, these variables are located in the tenured area. In Larceny, an assignment to such a variable is essentially a `set-car!`, so in the worst case, an assignment takes 3.5 μs. The assignments therefore account for as much as 0.5 seconds of execution time.

2. While garbage collections under the generation-scavenging collector are consistently less expensive than the collections under the mostly-generational collector, there are more collections under the generation-scavenging collector, resulting in increased garbage collection overhead.

   The larger number of collections is probably due to the transaction area filling up because of the assignments; while the transaction area should have room for all transactions during the run of `boyer` under all but a couple of the runs, the tenured area may contain much garbage from earlier benchmark runs, and this garbage will make the transaction area fill up quickly, triggering a collection.

3. The generation-scavenging collector causes a number of page faults. Again, the page faults are probably due to bad locality in the transaction area.

In summary, assignments hurt the `boyer` benchmark in three different ways under the generation-scavenging collector.

While there is no clear correspondence between the amount of memory available and the time it takes to run `boyer`, some bad times are caused by a low tenuring limit coupled with a small tenured area, causing full collections to happen in the generation-scavenging collector; the transaction area is a contributing factor as well, as how full the transaction area is will affect the kind of collection performed.

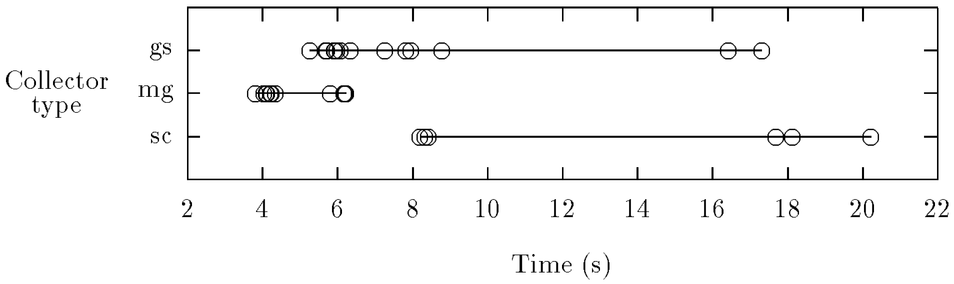

Figure 14: The running time of the `boyer` programs.

More interesting than the running times themselves are the numbers we can derive from them. Table 7 presents some garbage collection performance measurements for the `boyer` benchmark. The rates are in MB copied per CPU second; page faults are not taken into account. We notice that the stop-and-copy collector appears to have approximately three times the copying performance of the generation-scavenging collector, which is again twice as fast as the mostly-generational collector. The superiority of stop-and-copy comes from its tight and simple inner loop, and the inferiority of the mostly-generational collector comes from its having to scan a large number of words for each word it copies.

Table 7: Garbage collector performance for `boyer`.

Collector	MB/s copied
Generation-scavenging	1.6
Mostly-generational	0.8
Stop-and-copy	4.1

The `Samefringe` Benchmarks

`Samefringe-1` is used as a garbage collection benchmark. Its input is two very different trees, each with 1001 leaf nodes: one tree is a degenerate left-heavy tree, the other is a degenerate right-heavy tree. The test program flattens each tree to a list of its labels using the same traversal order, and then compares the lists. The benchmark was run five times.

The flattening allocates a great deal of memory: nearly 20 MB. This furious allocation naturally causes many garbage collections (as many as 220 with some memory configurations), and since a garbage collection flushes the stack cache, much memory is allocated to the flushed stack frames. Due to the nature of the input, the call graph is deep. Memory allocated to flushed frames varies from 250 KB to 2 MB, and in one case where the stack cache is particularly shallow, 42 MB.

`Samefringe-2` and `samefringe-3` do not allocate much memory but are used to compare the cost of various "good" traversal strategies. The input to these programs was two full binary trees of height 12 with no shared structure; the trees differed in the rightmost leaf, and the traversal order was always left child first. Each benchmark was run five times.

No tenuring or full collections occurred in any of the test runs.

The results for the fringe benchmarks are summarized in table 8. The generation-scavenging system had superior running times in all cases, and with the exception of the run with the shallow stack cache, all times for the generation-scavenging system were better than any time for the two other systems. Key to the performance advantage of the generation-scavenging system is that very little time was spent in garbage collection. Curiously, the mostly-generational system experienced very few page faults yet was slower than the generation-scavenging system, which experienced an unexpectedly high number of page faults. The large number of faults in the generation-scavenging collector relative to the mostly-generational collector can be explained by the fact that the test data resides in the tenured area: the mostly-generational collector will have a working set consisting of the ephemeral area as well as the tenured area (because it is scanned on each collection); the working set of the generation-scavenging collector is much smaller and sparse, and the paging effects seen may be due to operating system page allocation policies based on frequency and locality of reference.

The stop-and-copy system was from five to 50 times slower than the mostly-generational system, spending much time in garbage collection and experiencing as many as 4,000 major and 50,000 minor page faults.

Table 8: The results of the samefringe benchmarks.

Collector	Min time	Max time	Avg time	Median time
Generation-scavenging	12000	26750	14171	13340
Mostly-generational	14940	96470	32635	21750
Stop-and-copy	74820	465640	178446	143730

Results for `samefringe-1`

Collector	Min time	Max time	Avg time	Median time
Generation-scavenging	1230	2900	1896	1750
Mostly-generational	950	1700	1091	950
Stop-and-copy	1000	1270	1149	1200

Results for `samefringe-2`

Collector	Min time	Max time	Avg time	Median time
Generation-scavenging	960	1310	1145	1150
Mostly-generational	920	3060	1381	1070
Stop-and-copy	990	4030	2350	1610

Results for `samefringe-3`

Calculating pure garbage collector performance, we find that the results from the

`samefringe-1` benchmark correspond reasonably well with the results from the `boyer` benchmark; table 9 summarizes the garbage collector performance numbers. While the performance for the mostly-generational collector is only 25% of what it was for that same collector for the `boyer` benchmark, the lifetime of objects in the `samefringe-1` benchmark is so short and the number of collections so large that this difference in performance is reasonable.

Table 9: Garbage collector performance for `samefringe-1`.

Collector	MB/s copied
Generation-scavenging	2.0
Mostly-generational	0.2
Stop-and-copy	4.1

The `samefringe-2` benchmark, which uses an explicit stack to keep track of choice points, runs faster under the mostly-generational and stop-and-copy collectors than under the generation-scavenging collector. The main reason for this seems to be that the generation-scavenging collector page faults a fair bit (for no obvious reason).

The results for `samefringe-3` are the opposite of those for `samefringe-2`: the generation-scavenging system has superior performance. The mostly-generational system suffers from too much garbage collection overhead when the ephemeral area gets small, and the stop-and-copy system page faults much. The generation-scavenging system performs as many collections as the mostly-generational system, but spends no measurable time doing so.

The results are doubly surprising because `samefringe-3` uses generators, which use side effects to maintain local state; we would almost have expected the results for `samefringe-2` and `samefringe-3` to be the opposite of what they are. On closer inspection it turns out that `samefringe-3` allocates twice as much memory as `samefringe-2` for a stack internal to the generator; the results are therefore not as strange as they would seem.

In our measurements, the cost ratio of the explicit stack approach and the generator approach vary from 0.5 to 1.7 (for the averages). Overall, `samefringe-3` runs faster than `samefringe-2` under the generation-scavenging collector, but slower under the two other collectors. This is good news for the generators, but need to be substantiated further. If the generator is moved into the tenured area by a garbage collection, it may be too costly due to the increased cost of assignments.

We also coded `samefringe-1` in C; this is the `samefringe-C` benchmark. The motivation for this experiment is to measure the differences in performance between a garbage-collected implementation and an implementation which manages memory explicitly. Using the standard C memory management procedures `malloc()` and `free()`, we measured a single run of `samefringe-C` with the same inputs as for `samefringe-1`. This single run takes about 50 seconds, whereas a single run of `samefringe-1` takes 2.8 seconds, that is, the C implementation is a factor of 18 slower than the Scheme implementation.[4] The C version of the benchmark used loops where it could, but even if the compiler had compiled tail recursion properly, procedures which are tail-recursive in Scheme could not always be written in a tail-recursive form in C since temporary storage

[4]Measurements obtained on the Sparc ELC with the bundled Sun C compiler at optimization level 4 and the standard Sun library.

for e.g. list cells sometimes have to be freed after a recursive call.

In order to even the odds somewhat, we changed the C implementation to manage its own free lists; the resulting program runs a little faster on the average than the Scheme program (2.4 seconds). When our compiler gets better procedure call performance, we expect this difference to go away.

While it is the case that few C programs will use memory the way Scheme programs do, a memory-intensive programming style can often make programs easier to write and understand. The point we are making is that explicit memory management is not necessarily faster than garbage collection, and it can in fact be much slower. As we saw, the use of programmer-managed free lists sped the C program up by a factor of 20, but such free lists are increasingly bothersome when the programs get larger and use several sizes of memory blocks. In addition, free lists can cause fragmentation of memory and subsequent poor paging behavior. Finally, Zorn [28] has observed that for larger programs, programmer-managed storage has few performance advantages over storage managed by e.g. `malloc`.

The `Reverse`, `Append`, and `Map` Benchmarks

The `reverse`, `append`, and `map` benchmarks are sufficiently closely related that we will treat them together. Recall that `map-1` uses `reverse` and that `map-3` uses `reverse!`, and furthermore that for both these `map` procedures, the reversal procedure is likely to contribute significantly to the execution time (because the procedure being mapped is just the identity procedure).

Variations on `Reverse`

The reverse benchmarks were run 10,000 times on lists of length 100, 1,000 times on lists of length 1,000, and 100 times of lists of length 10,000. All new data became garbage right away. For `reverse!` the output of one pass was the input to the next; for the shorter lists, this will probably contribute to very good cache locality for `reverse!`, and even for longer lists, `reverse!` will have an advantage locality-wise as the parts of a list which were referenced last on one pass are referenced first on the next pass. The results are therefore skewed somewhat in the favor of the imperative implementation.

Figures 15 and 16 show graphically the execution time for `reverse` and `reverse!` for ephemeral data and tenured data, respectively. The results are also summarized in tables 10 and 11.

As could be expected, `reverse!` is vastly superior to `reverse` under all configurations with the mostly-generational and stop-and-copy collectors, using from 700 to 1,000 ms in almost all cases. With these collectors, the time for `reverse` varies from the benign 3,000 ms to the outrageous 760,000 ms (13 minutes). Interestingly, the worst times are for the mostly-generational collector. Looking behind the scenes, we discover that the slowest run was somewhat anomalous (twice as slow as the next slowest one), and also that the reason for the slowness is the almost 10,000 major and 140,000 minor page faults incurred during the run.

Under the generation-scavenging collector the tradeoffs are more interesting. When the data being operated on is in the ephemeral area, `reverse!` is still appreciably faster than `reverse`, especially when the ephemeral area is small. Once the data being operated on moves into the tenured area, however, the running times of `reverse` do not change, but the running time of `reverse!` gets worse by a factor of 3, on the average (although more dramatic slowdowns are also observed). From Chapter V we have that the expected worst case of `set-cdr!` is

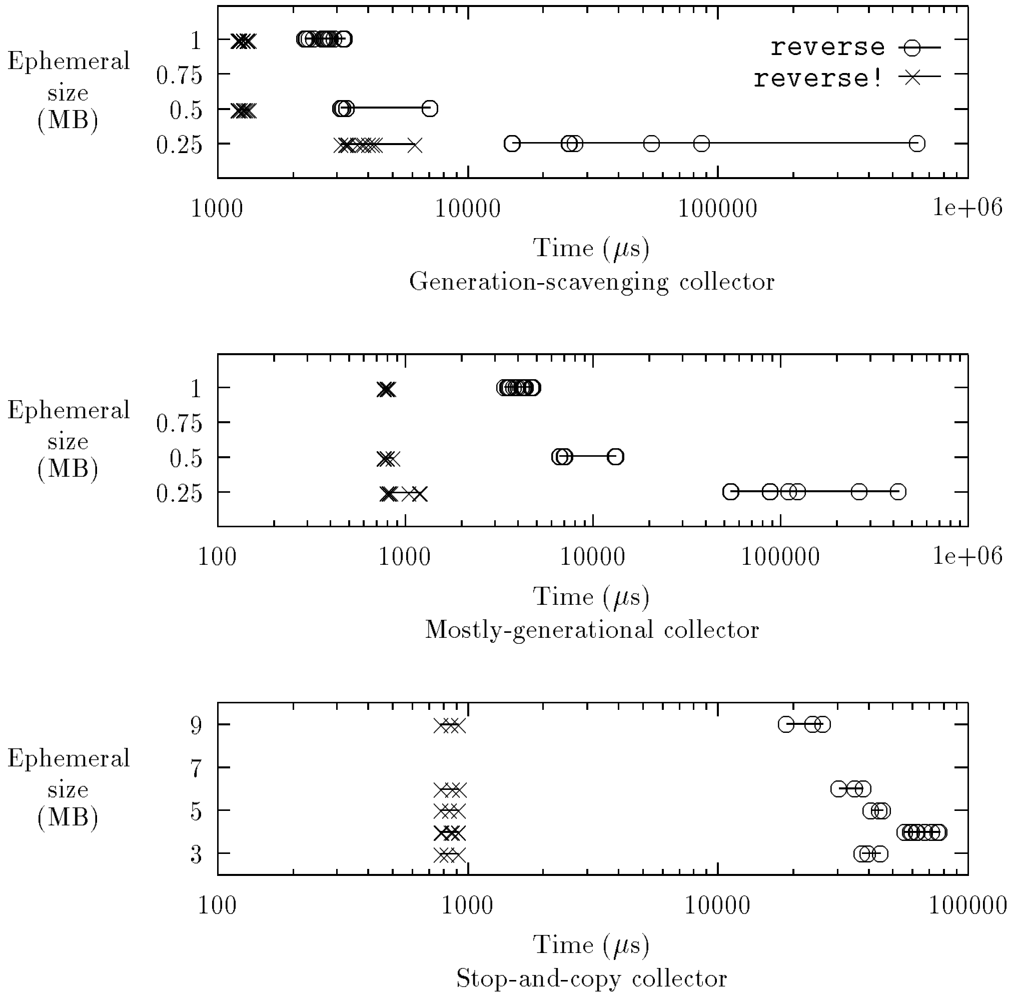

Figure 15: Cost of `reverse` vs. `reverse!` in the ephemeral area.

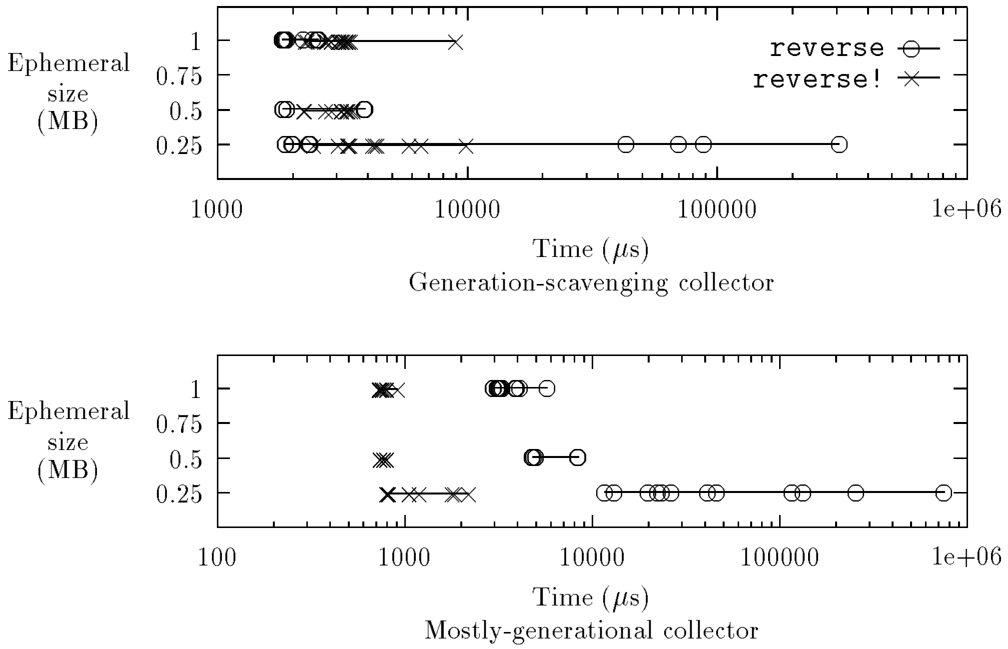

Figure 16: Cost of `reverse` vs. `reverse!` in the tenured area.

worse than the worst case of `cons`; the costs of `reverse!` relative to `reverse` in the tenured area correspond relatively well to the cost of `set-cdr!` relative to `cons`.

The unexpectedly bad worst-case times for `reverse` with a small ephemeral area are due to subtleties of the tenuring policies in the test system. During the run, the ephemeral area fills up frequently, and a tenuring collection is called for. However, the worst-case run was done with a 3MB tenured heap, and when a tenuring collection was initiated, there was less free space in the tenured heap than might be needed by the tenured objects. Therefore, a full collection was performed instead, resulting in disastrous page faulting behavior. The corresponding run with a 4 MB tenured area used only 1/7 of the time. A more stable test system will undoubtedly be able to correct for this kind of behavior by enlarging the tenured heap when it gets close to full.

Table 10: Summary of running times for the `reverse` benchmark in ms.

	Ephemeral data				Tenured data			
Input size	Lo	Hi	Avg	Median	Lo	Hi	Avg	Median
Generation-scavenging								
100	2650	15190	5596	2940	1820	2020	1856	1840
1000	2240	25700	7703	2440	1840	2370	1982	1900
10000	2640	629840	47676	3240	2210	309130	30689	2560
Mostly-generational								
100	3920	54830	16067	4760	2970	41380	8335	4820
1000	3420	89160	23377	3920	3000	46440	8917	3300
10000	3810	428900	57188	4870	3340	759050	74333	4130
Stop-and-copy								
100	18840	62980	44170	41010	N/A	N/A	N/A	N/A
1000	23950	67240	48190	44940	N/A	N/A	N/A	N/A
10000	26350	77250	53403	45930	N/A	N/A	N/A	N/A

Table 12 shows the garbage collector performance in bytes copied per CPU second for the `reverse` and `reverse!` benchmarks. The performance compares

Table 11: Summary of running times for the `reverse!` benchmark in ms.

	Ephemeral data				Tenured data			
Input size	Lo	Hi	Avg	Median	Lo	Hi	Avg	Median
	Generation-scavenging							
100	1210	6170	1931	1310	3150	9930	4312	3330
1000	1210	3350	1681	1230	2210	3400	2926	3060
10000	1270	4280	1891	1270	2240	4370	3236	3280
	Mostly-generational							
100	780	840	792	780	740	1790	829	770
1000	770	820	779	770	730	1850	813	740
10000	800	1210	890	810	800	2200	936	800
	Stop-and-copy							
100	830	870	859	860	N/A	N/A	N/A	N/A
1000	780	790	781	780	N/A	N/A	N/A	N/A
10000	920	930	921	920	N/A	N/A	N/A	N/A

relatively well with the performance measured in the `boyer` and `samefringe` benchmarks; the performance is slightly higher for the generation-scavenging collector than it was measured to be under the garbage collection benchmarks.

Table 12: Garbage collector performance for `reverse` and `reverse!`.

Collector	MB/s copied (`reverse`)	MB/s copied (`reverse!`)
Generation-scavenging	2.2	0.8
Mostly-generational	0.5	0.2
Stop-and-copy	4.0	4.1

Variations on `Append`

There are two versions of `append`: a recursive one and an iterative one which uses `set-cdr!` to build up the resulting list as it goes along. The inputs to the `append` procedures were of three lengths: 100, 1,000, and 10,000 elements; these tests were repeated 10,000, 1,000, and 100 times, respectively. All input data were initially located in the ephemeral area. Figure 17 summarizes the running times for the append benchmarks; tables 14 and 15 detail the results based on the size of the input.

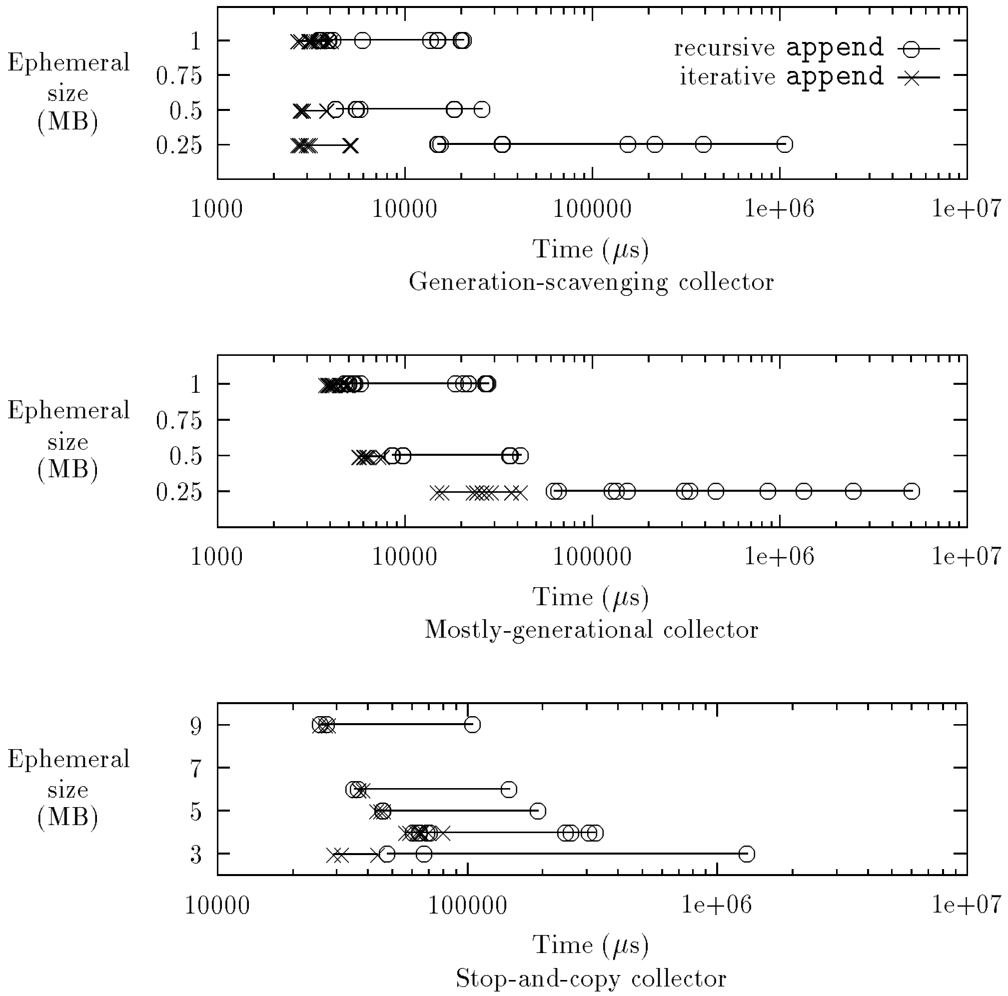

Figure 17: Cost of recursive `append` vs. iterative `append`.

In general, we see that the stop-and-copy collector is an order of magnitude slower (in the best case) than the two other collectors. Considering that these benchmarks are heavy on allocation and hence cause a large number of garbage collections (from 10 to 50, depending on the allocated heap space), coupled with the fact that a collection in the stop-and-copy system is expensive, the slowness of the stop-and-copy system is not surprising.

Also notable is that in the generation-scavenging system, the times for the iterative program are mostly independent of the size of the ephemeral area, whereas in the two other configurations, the heap size affects the running time for the iterative program. The reason is probably that under the less-sophisticated collectors, the cost of allocation and reclamation dominates, and that cost goes up when the size of the ephemeral area decreases.

From figure 17 it is quite clear that the iterative program is significantly faster than the recursive one. Although the two programs are nearly tied in some cases, the iterative program is so much faster on the average and also so much less prone to pathological behavior that it is impossible to justify the use of the recursive program.

Looking closer at the results from the append tests (tables 14 and 15), we see that the worst times for the recursive programs come from the runs where the input lists are very long. In these runs, a very large number of stack frames are flushed to the heap (and later have to be restored). From earlier results we know that the running time goes up by as much as 1000% under the generation-scavenging collector when the stack cache starts overflowing. The results from `append` substantiates this; for large inputs, the recursive implementation is an order of magnitude slower than the iterative one, and it is reasonable to blame the slowdown on stack cache overflows. Garbage collection time also goes up with increased stack overflows; for the larger inputs, as much as 40% of the running time is spent garbage collecting.

We can also calculate garbage collector performance for the append benchmark; the results are shown in table 13. The stop-and-copy collector shows better performance than we've seen earlier, but the other collectors are behaving as on the earlier benchmarks.

Table 13: Garbage collector performance for `append`.

Collector	MB/s copied (recursive)	MB/s copied (iterative)
Generation-scavenging	2.3	1.9
Mostly-generational	0.5	0.1
Stop-and-copy	4.7	5.2

Table 14: Cost of recursive `append` by input size, ms running time.

Input size	Min time	Max time	Average time	Median time
Generation-scavenging				
100	3480	15660	6543	4300
1000	3960	33480	11423	5550
10000	15000	1072170	117373	20220
Mostly-generational				
100	4820	2484040	158583	5480
1000	5340	5125060	322711	9860
10000	20780	1362020	191088	27890
Stop-and-copy				
100	26010	64950	49430	48010
1000	27600	261300	82851	67470
10000	105960	1324020	379454	248580

Variations on `Map`

Each `map` procedure was run on three combinations of lists: 10,000 lists of length 100, 1,000 lists of length 1,000, and 100 lists of length 10,000. All data was initially located in the ephemeral area. The procedure being mapped was always

Table 15: Cost of iterative `append` by input size, ms running time.

Input size	Min time	Max time	Average time	Median time
Generation-scavenging				
100	2700	3180	2933	2900
1000	2780	3280	3026	3140
10000	3380	5200	4068	3870
Mostly-generational				
100	3810	27290	8707	4540
1000	3900	29180	8986	4530
10000	4280	41630	12076	5060
Stop-and-copy				
100	25700	65860	46164	43510
1000	27900	65460	49163	45170
10000	27250	79890	50764	46260

the identity function. Figure 18 shows the running times for all the programs relative to the collector type. Table 16 summarizes the costs by input size and collector type.

Under all collectors, the imperative implementations are faster by as much as two orders of magnitude for some inputs; table 16 shows that the worst results are obtained with large inputs and small heaps. Also, under all collectors, the recursive `map-1` is the most expensive program. Finally, there is little difference between `map-3` and `map-4` under any collector.

Under the generation-scavenging collector, the imperative programs `map-3` and `map-4` are fairly stable in their timings, relative to the ranges of the functional implementations. The `map-2` program runs for the most part as fast as the imperative implementations; it has poor behavior only for the large inputs with severely constrained ephemeral areas. This relationship is shown in figure 19. With a small (0.25 MB) ephemeral area, the garbage collection and page faulting

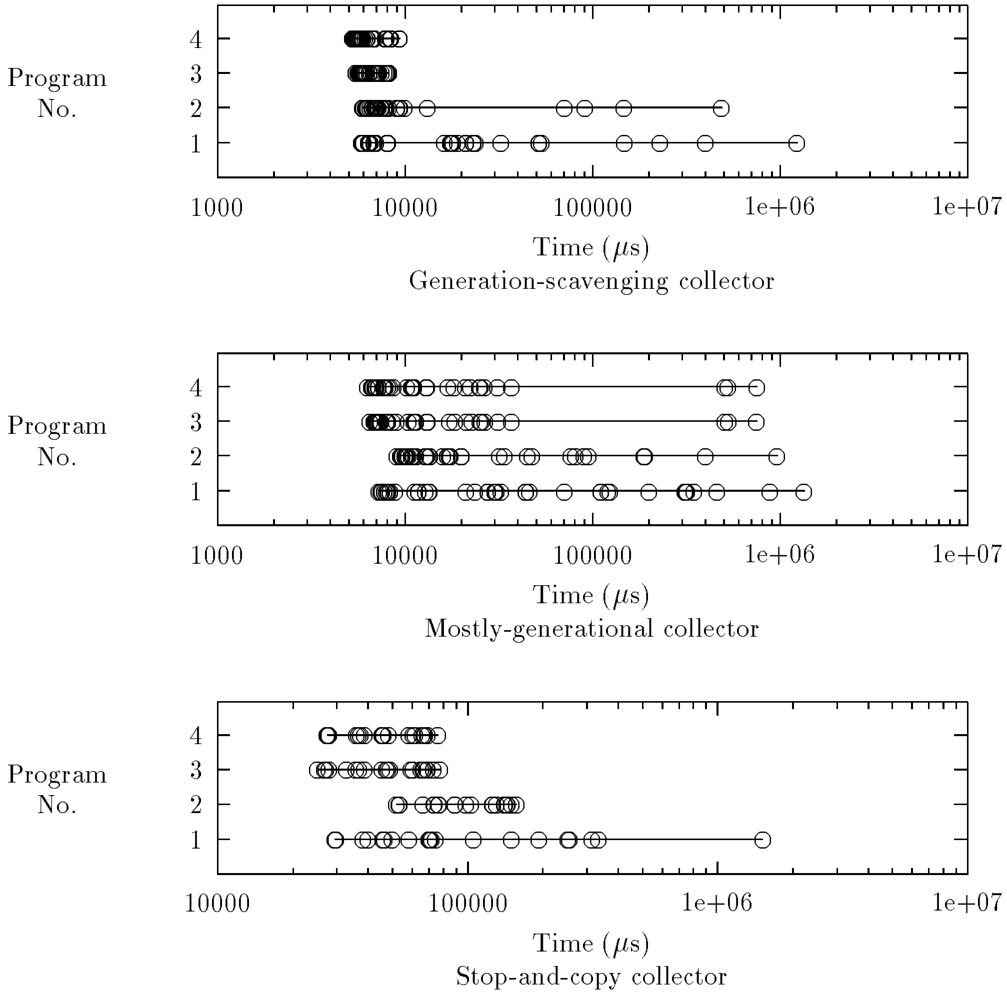

Figure 18: The cost of various `map` implementations relative to the collector type.

overhead is tremendous, with as many as 50 full garbage collections and 5000 major page faults in the worst case. If we except the four worst times (which all occurred with the small ephemeral area), `map-2` is as efficient as `map-3` and `map-4`. The recursive program `map-1` performs poorly in general, and is viable only for small and medium-sized inputs and a sufficiently large heap size and a high tenuring limit.

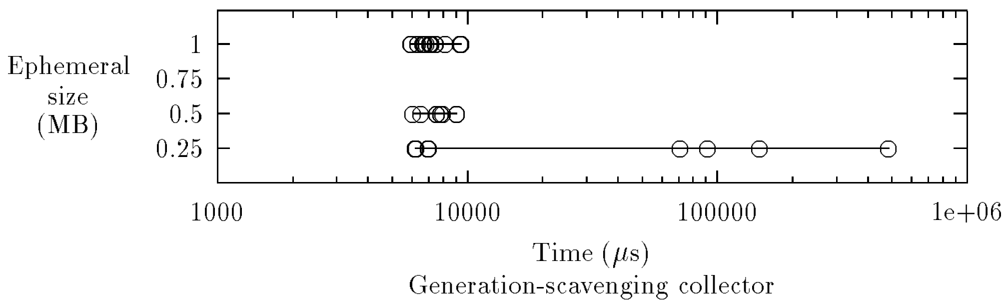

Figure 19: The cost of `map-2` relative to the size of the ephemeral area.

Interestingly, all `map` implementations perform equally poorly under the mostly-generational collector. The best-case times are similar to the best-case times under the generation-scavenging collector; however, the times get worse very quickly. Also under the mostly-generational collector, the imperative implementations have the edge, with most running times at the low end of the range and only occasional abnormal results in the high end of the range. The worst results for `map-3` and `map-4` are independent of input size but strongly dependent on the size of the ephemeral area: a small ephemeral area (0.25 MB) causes the poor performance. Also, all the poor times for the imperative programs were observed with the largest (8 MB) tenured area. The combination of a small ephemeral area and a large tenured area can be deadly in the mostly-generational collector. The reason is that the tenured area can fill up (during previous runs, for example) with tenured garbage, all of which has to be scanned on each collection. At the same time, collections are frequent because of the small ephemeral area.

The performance of `map` under the stop-and-copy collector is an order of magnitude worse than under the generation-scavenging collector, but the cost patterns are similar: the imperative programs are best, and `map-1` is worst. The worst-case time of `map-1` is the same under all three collectors, however, and in fact, the worst-case time of `map-2` is much better under the stop-and-copy collector than under the generation-scavenging collector.

In conclusion, it would appear that given a reasonably-sized ephemeral area and a run-time system which can detect near-full heaps, `map-2` is about as efficient as its imperative variations under the generation-scavenging collector.

The `Fib` Benchmarks

The `fib` benchmark was run with an argument of 30 for all three variations under each memory configuration. Figure 20 summarizes the results from the `fib` benchmark.

For all three collector types, the straightforward implementation `fib-1` (using the program stack) was the fastest. For the generation-scavenging collector, `fib-2` (using a list for the stack) came in second place, with `fib-3` (using a vector for the stack) last, despite the fact that `fib-2` caused on the average 34 garbage collections and `fib-3` caused none. The differences in execution time between `fib-1` and `fib-2` were small, strengthening the earlier analysis that the cost of a `cons` is roughly equivalent to the cost of creating a small stack frame under the generation-scavenging collector. Only `fib-2` incurred any garbage collections, from

Table 16: Summary of the cost of `map`, in ms per run.

Generation-scavenging collector				
Program and input size	Min time	Max time	Average time	Median time
`map-1` / 100	5910	19230	8869	6050
`map-1` / 1000	6440	53570	17602	8070
`map-1` / 10000	17660	1234280	129469	23340
`map-2` / 100	5970	7600	6654	6730
`map-2` / 1000	6360	7870	7129	7140
`map-2` / 10000	7200	488490	51216	9430
`map-3` / 100	5500	7320	6117	5880
`map-3` / 1000	5770	7550	6340	6150
`map-3` / 10000	6480	8300	7426	7080
`map-4` / 100	5300	8390	6029	5670
`map-4` / 1000	5520	8490	6266	5860
`map-4` / 10000	6140	9420	7357	6770

Mostly-generational collector				
Program and input size	Min time	Max time	Average time	Median time
`map-1` / 100	7330	124720	30105	8040
`map-1` / 1000	8130	349720	73894	13010
`map-1` / 10000	23700	1348510	194100	31100
`map-2` / 100	9100	90390	22583	10480
`map-2` / 1000	9730	95310	23672	10900
`map-2` / 10000	11110	971210	108678	13060
`map-3` / 100	6560	511740	38226	7390
`map-3` / 1000	6910	535280	39900	7490
`map-3` / 10000	7390	755670	54483	8150
`map-4` / 100	6350	510970	37977	7240
`map-4` / 1000	6620	528290	39256	7250
`map-4` / 10000	7040	759140	54446	8130

Stop-and-copy collector				
Program and input size	Min time	Max time	Average time	Median time
`map-1` / 100	29660	71320	51590	46980
`map-1` / 1000	29870	257990	85077	71150
`map-1` / 10000	106560	1525700	411500	252820
`map-2` / 100	52110	140640	98737	89370
`map-2` / 1000	53190	144030	103950	99120
`map-2` / 10000	53780	157540	107729	103890
`map-3` / 100	26890	67100	48026	45720
`map-3` / 1000	25190	69390	47879	47620
`map-3` / 10000	28030	77680	54389	49190
`map-4` / 100	27680	67360	50337	48650
`map-4` / 1000	27560	67970	50297	46090
`map-4` / 10000	27990	76230	50926	46100

20 to 60 depending on the size of the ephemeral area.

For the mostly-generational and stop-and-copy collectors, `fib-3` was faster than `fib-2`; the number of garbage collections incurred by `fib-2` made this program run up to three times slower than `fib-3` under the mostly-generational collector and up to 20 times slower than `fib-3` under the stop-and-copy collector. For both collectors, the gap between `fib-1` and `fib-3` was on the order of 1500 ms.

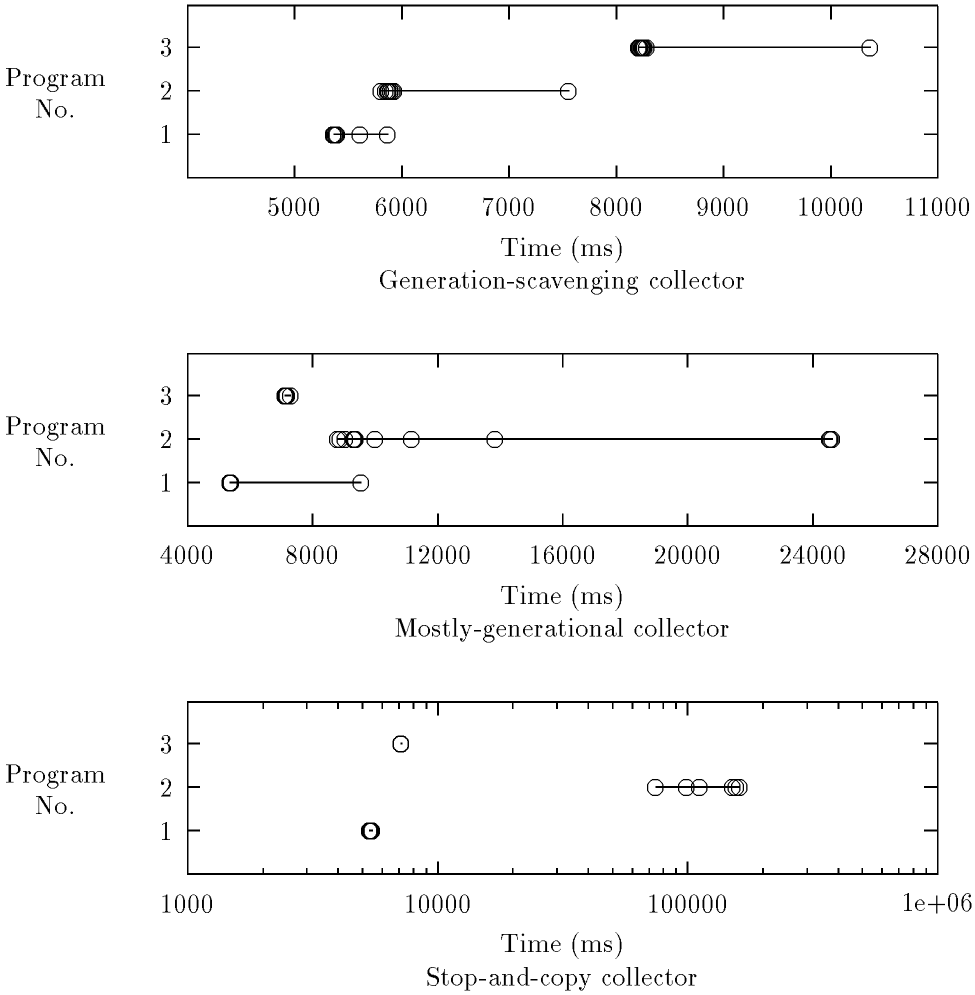

Figure 20: Results from running `fib`.

In conclusion, using the program stack still seems to be preferable to using a programmer-managed stack, and this result is independent of the garbage collector type.

The `Quicksort` Benchmarks

Recall that the quicksort benchmarks were written to explore the differences in various implementation strategies of an essentially recursive algorithm, by using either the program stack or various forms of explicit stacks. Figure 21 shows the running times of the quicksort programs for all the collectors. Each run performs an in-place sort of a vector of length 30,000 containing random non-negative integers less than 4,000.

As one would expect (because these programs use `vector-set!` heavily), the costs are usually the highest for the generation-scavenging collector, where side effects are more expensive. In addition, the costs are similar for the mostly-generational and the stop-and-copy collectors, between 4,050 and 4,450 ms for a run.

The notable exceptions to the uniformity of the running costs are `quick-2` and `quick-5` running under the stop-and-copy collector. These two benchmarks are much more expensive than the others because they allocate quite a bit of memory for the explicit stack, eventually causing an expensive garbage collection. A garbage collection happened only in the test configuration where the heap was nearly full (3 MB allocated to the ephemeral area), yet other test runs were expensive as well; in fact, the configurations which caused the collection did not have the worst running times. Rather, poor running times were observed in configurations with plenty of memory and was caused by poor paging behavior, which in turn is probably due to there being enough memory available to cause page faults when memory is allocated (because pages in the ephemeral area are brought in from disk). In other words, the program may run faster with less memory allocated to it.

For all three collectors, however, the `quick-1` program is the fastest. This program uses the program stack for its recursion, whereas all the other programs use explicit stacks of various kinds. The most expensive stack type was the one used by `quick-2` and `quick-5`: a list onto which the program conses data. However, for the explicit-stack programs, `quick-2` and `quick-5` also had the best best-case running times. The two other kinds of stacks (vector and doubly-linked list) were consistent in their running times, with the vector strategy being the slower one. In the generation-scavenging collector, all explicit-stack strategies had similar worst-case times.

In conclusion, using the implicit stack is again faster than using any sort of explicit stack, and when using an explicit stack, the doubly-linked list, although complicated in implementation, gives good performance.

Furthermore, since `quick-2` consistently beat `quick-5`, it appears that in this (admittedly stack-intensive) case, rebinding is less expensive than assignment.

The `Sort` Benchmarks

The merge sort benchmarks `sort-1` and `sort-2` were run on two sets of inputs, both random lists of positive integers less than 10,000. For the first test, the list had length 30,000 and the sort was run once. For the second test, the list had length 1,000, and the sort was run 30 times. All data was initially located in the ephemeral area. The results are shown graphically in figures 22 and 23.

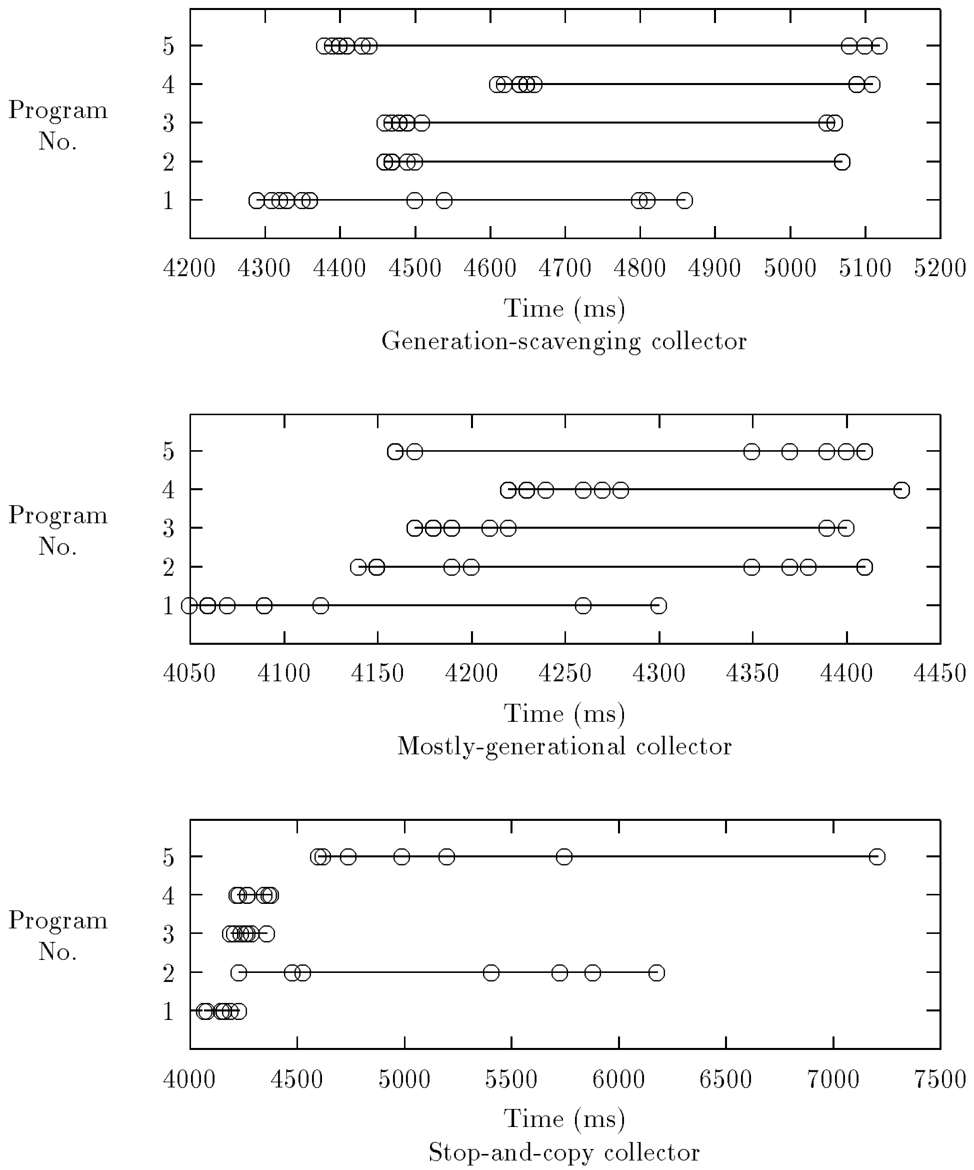

Figure 21: The running times for the `quicksort` programs.

For the generation-scavenging collector, the figures show us that for small inputs, the side-effect free `sort-1` is only 50% slower than the imperative `sort-2`, and for large inputs, the same relationship holds for a sufficiently large ephemeral area (1 MB); for small ephemeral areas, the extra allocation performed by `sort-1` degrades performance noticeably.

For the mostly-generational system, `sort-2` is about three times as fast as `sort-1` on both large and small inputs, but the running times for large inputs are very poor when the ephemeral area gets small. The mostly-generational system has the best running times for `sort-2` on large inputs.

The stop-and-copy system has poor performance on both sets of inputs, although the times are bearable given enough memory. The times for `sort-1` are as much as an order of magnitude worse than those for `sort-2`.

Finally, the times for `sort-2` are similar for both the generation-scavenging and mostly-generational systems in spite of all the assignments performed by this benchmark. The reason why the generation-scavenging system can keep up is probably that the data is all in the ephemeral area. For `sort-1`, however, the extra allocation penalizes the mostly-generational system much more than it does the generation-scavenging system.

In summary, the functional-style `sort-1` appears practical for small and medium-sized inputs, and given a sufficiently large ephemeral area, it is practical for relatively large inputs as well.

The `Vector-add` Benchmark

All runs of the vector addition benchmarks add two vectors 1,000 times, producing 1,000 result vectors which immediately become garbage. All vectors are

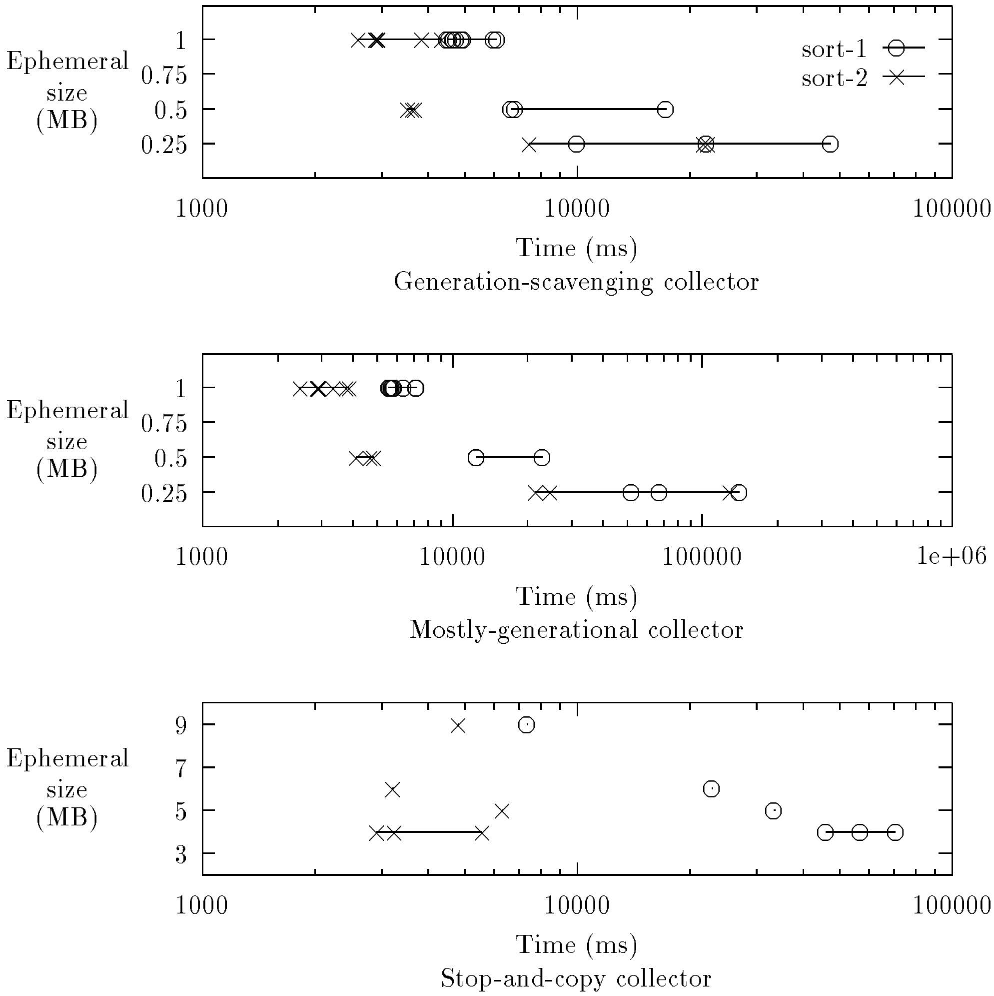

Figure 22: The cost of the sort benchmarks for input size 30,000.

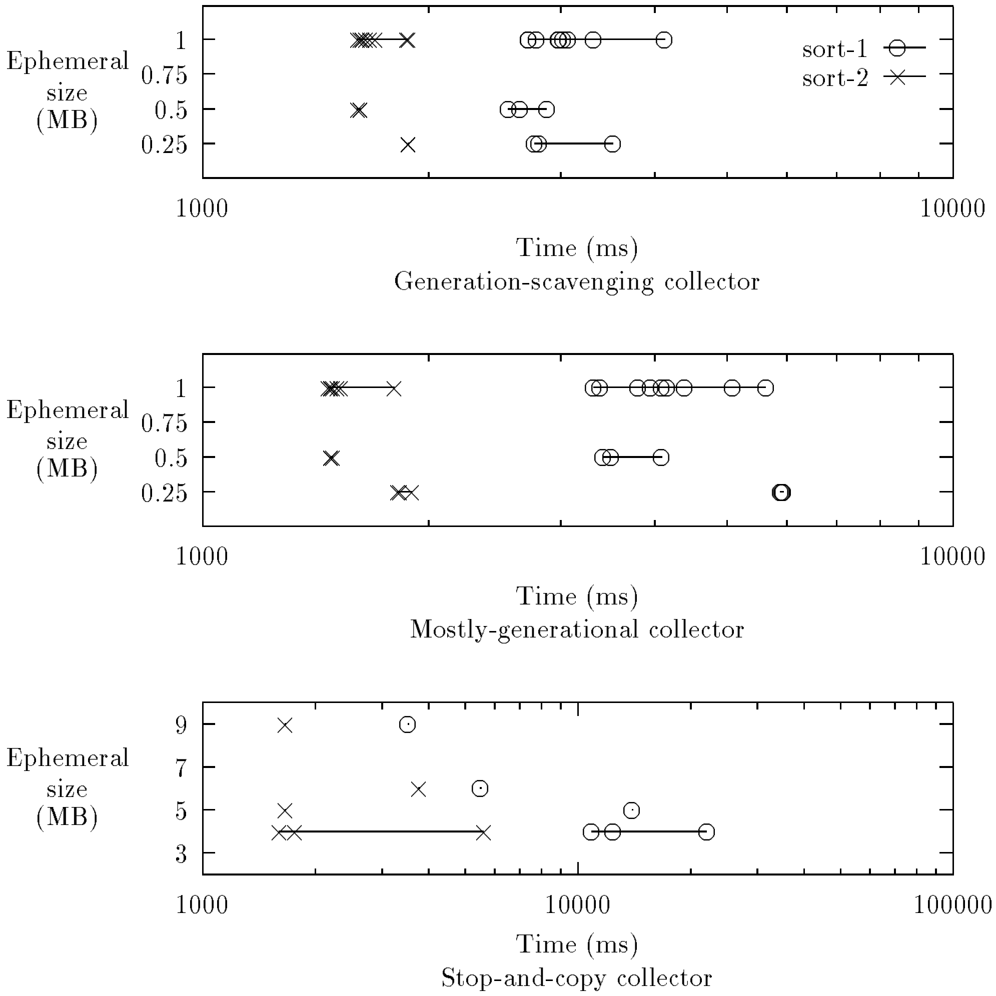

Figure 23: The cost of the sort benchmarks for input size 1,000.

of length 100; we would expect cache locality to be excellent (although it may be irrelevant).

Figure 24 summarizes the results of the vector addition benchmarks. The general result is that with the generation-scavenging collector, the functional implementation `vector-add-2` is from 2% to 20% slower than the imperative implementation. For the mostly-generational collector, the extra cost for the functional implementation is from 20% to 15,000%, with most runs in the low part of that range. The stop-and-copy system runs slowly for both implementations, with the functional implementation being from 3% to 1,500% slower than the imperative one.

The exceptionally large ranges of running times seen for the mostly-generational and stop-and-copy systems with small ephemeral areas are due both a large number of page faults and increased garbage collection costs; these causes are not always related, however.

The benchmarks show that given a generation-scavenging collector (and for the measured input sizes), it is viable to allocate results in a list and then create a vector from the list. Also bear in mind that the imperative vector addition procedure operates on a vector in the ephemeral area; if the result vector is copied into the tenured area, costs for the imperative version will increase.

The `Sieve` Benchmarks

The `sieve` benchmarks belong to two logical groups. `Sieve-1`, `sieve-2`, and `sieve-4` all have time complexity $O(n^2)$ for finding the first n primes, while `sieve-3` and `sieve-5` always run in $O(n^{3/2})$ time.

As a consequence of the computational complexity of the two groups,

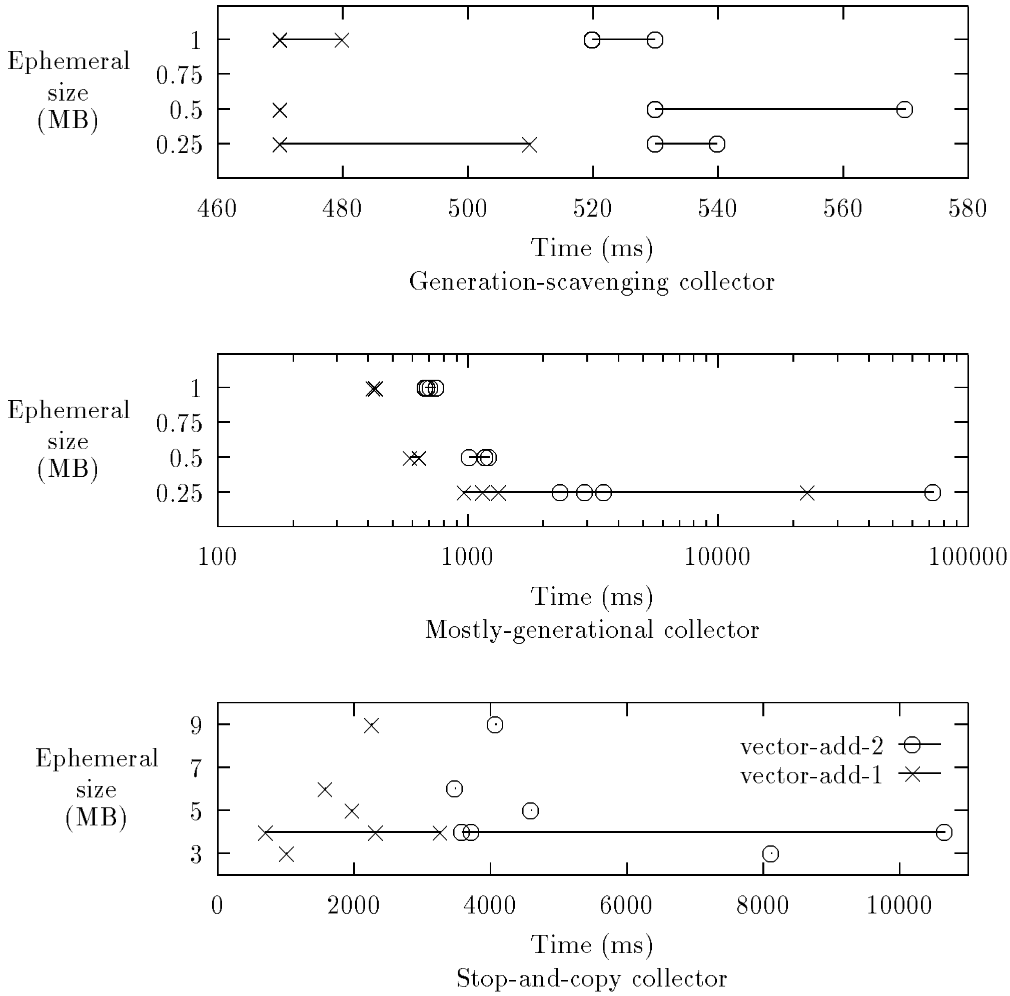

Figure 24: The cost of the `vector-add` benchmark.

programs in the former group are run to find the first 800 primes, and programs in the latter group are run to find the first 3000 primes. They all do this task once.

Table 17 summarizes the running times for the first group and table 18 summarizes the running times for the second group.

Recall that the purpose of `sieve-1` and `sieve-2` was to measure the difference between rebinding and assignment. As such, they are inconclusive: no significant differences are measurable. Possibly, any differences are lost in the noise of the rather inefficient algorithm.

`Sieve-4` allocates about 40 KB of memory for its nested generators, which is not enough to trigger a collection in any of the systems. However, since the generator implementation is not tail-recursive, the test run with the short stack cache performs a lot of stack flushes, triggering one garbage collection in the stop-and-copy system and two collections in the generational systems. The stop-and-copy system suffers the most from this, with the run in question being an order of magnitude slower than the others; the generational systems slow down by a factor of two.

Table 17: Summary of the cost of the slow sieves, times in ms per run.

Collector and program	Min	Max	Average	Median
Generation-scavenging, sieve-1	7800	8090	7956	7980
Generation-scavenging, sieve-2	7800	7910	7859	7870
Generation-scavenging, sieve-4	900	2040	1087	940
Mostly-generational, sieve-1	7790	8010	7840	7830
Mostly-generational, sieve-2	7790	7950	7822	7820
Mostly-generational, sieve-4	880	2320	1049	990
Stop-and-copy, sieve-1	7810	7970	7900	7910
Stop-and-copy, sieve-2	7800	8000	7889	7880
Stop-and-copy, sieve-4	960	14330	3016	1110

For `sieve-3`, the times are uniformly 910-920 ms for all configurations; this should not be a surprise, as the only memory allocated is the vector to hold the primes as they are found. What is interesting is that the times for the generation-scavenging system are the same as for the other systems, even in the presence of side effects. But, as we saw earlier, `vector-set!` in the ephemeral area is not much more expensive for the generation-scavenging systems than for the others, and if the operations is executed infrequently (which is the case in this benchmark), the impact should be low.

The streams program `sieve-5` varies in cost with the collector type. It allocates about 2 MB of memory and causes several collections. The collections have most impact in the stop-and-copy system, but are noticeable in the mostly-generational system as well. The ratio of the cost of the streams approach to the cost of the imperative approach is about 4 for the generation-scavenging system, 4 to 5 for the mostly generational-system, and from 6 to 25 for the stop-and-copy system.

Whether a factor of four is an acceptable cost for the streams approach over the vector approach is perhaps debatable; however, we see that the streams approach increases in viability with the sophistication of the garbage collector.

Table 18: Summary of the cost of the fast sieves, times in ms per run.

Collector and program	Min	Max	Average	Median
Generation-scavenging, imperative	920	1740	981	920
Generation-scavenging, stream	3690	4800	3905	3830
Mostly-generational, imperative	910	920	917	920
Mostly-generational, stream	4090	5260	4461	4200
Stop-and-copy, imperative	910	1180	974	930
Stop-and-copy, stream	6360	25950	14061	17010

The `Nfa` Benchmarks

The `nfa` benchmarks were both run 1000 times, matching straightforward implementations of the pattern `((a|c)*(bcd))|(a*(bc))` against a string containing 133 "a"s followed by "bc". Table 19 summarizes the results.

The recursive implementation, `recursive-nfa`, allocated about 1 MB of memory; this corresponds well to the 135,000 cons cells created by the call to `string->list` when the input string (135 characters long) is converted to a list before every call to one of the two programs (1000 calls). The continuation-passing implementation, `cps-nfa`, allocated almost 8 MB of memory; the 7 MB allocated beyond the 1 MB for the input is accounted for by the large number of closures created.

The generation-scavenging system had the best running times, with the mostly-generational system second, and the stop-and-copy system a distant third. In all cases, the `recursive-nfa` program was faster than the `cps-nfa`, which is hardly surprising. However, for the generation-scavenging system, the difference between the two benchmarks was only about 150 ms, or 10%. The difference between the two programs was bigger for the other two systems: a factor of 8 or 9 for the stop-and-copy system, and a factor of 2 or 3 for the mostly-generational system. These systems suffered from much garbage collection overhead and in the case of the stop-and-copy system, a large number of page faults.

In conclusion, a continuation-passing style can be practical with a good garbage collector, but for this test program, using the program stack was still faster.

Table 19: Cost of the NFA implementations.

Collector and program	Min	Max	Avg	Median
Generation-scavenging (recursive)	1490	1520	1499	1500
Generation-scavenging (cps)	1670	1690	1674	1670
Mostly-generational (recursive)	1630	2000	1745	1660
Mostly-generational (cps)	2670	6030	3789	3280
Stop-and-copy (recursive)	3820	7800	4729	4410
Stop-and-copy (cps)	20600	61680	41950	45960

Summary of Results

In this section we summarize our findings by style issue and make some general observations which have been lost in the detail of the benchmark results.

Cost of System Primitives

The correspondences observed between expected and actual costs indicate that for the generation-scavenging collector, the expected cost for an operation is actually indicative of the actual cost of the operation. This fact enables us to predict the running times of our programs with some accuracy. For the mostly-generational and stop-and-copy collectors, however, the relationships between expected and actual costs are meaningful only for operations which do not allocate memory, like `set-cdr!`.

We also observed that a `set-car!` or `set-cdr!` under the generation-scavenging collector is as much as 16 times as expensive as the same operation under the stop-and-copy and mostly-generational collectors (5 to 9 times as expensive on the average). Furthermore, under the generation-scavenging collector, a `set-cdr!` in the tenured area is 2 to 3 times as expensive as the same operation in the ephemeral area, and in fact, a `cons` is only half the cost of a `set-cdr!` in the tenured area. A `cons` under the mostly-generational and stop-and-copy collectors is so expensive that it should not be considered a viable alternative to using a side effect if the choice is possible.

The cost of a `vector-set!` is as much as three times as expensive under the generation-scavenging collector as under the other two collectors; also for the generation-scavenging collector, a `vector-set!` in the tenured area is about twice as expensive as one in the ephemeral area.

If the stack cache does not overflow, the cost of creating, using, and disposing of a small stack frame is comparable under all collectors to the cost of a `cons` under the generation-scavenging collector; this has an impact on the use of explicit stacks, as explained below. The cost of using a stack frame goes up a great deal once the stack cache is flushed, however: 10 times for the generation-scavenging collector, 15 times for the mostly-generational collector, and 130 times for the stop-and-copy collector. The cost of garbage collecting the flushed frames is a major factor for the latter collector. Deep recursion is therefore possible under the better collectors, but not feasible under the poorer ones.

Since a stack flush causes the running time to go up, one might question whether the stack cache should be flushed before a garbage collection (the alternative is to modify the garbage collector to consider the stack among the roots for collection; such a modification complicates the collector). The flushing-before-collecting issue needs to be investigated further.

Garbage Collector Performance

It is relatively clear from the results that a generation-scavenging collector makes a functional style viable. For the other collectors, imperative programs are invariably faster than functional ones and storage allocation is expensive. In particular, the mostly-generational collector will behave poorly if much data is tenured prematurely, as scanning costs will increase. A large tenured area coupled with frequent collections and/or a poor tenuring policy will bring a mostly-generational system to its knees. The stop-and-copy system has excellent copying performance, but requires that there be enough physical memory available to hold most of its pages, or the page fault costs will soon outweigh the performance gains from the simple algorithm. Even with enough physical memory to hold the entire heap, however, the generation-scavenging collector will likely outperform the stop-and-copy collector when a large amount of data is live.

The remainder of the analysis will concentrate on the results for the generation-scavenging system.

Garbage Collector Implementation Issues

One observation which was not made previously is that it seems to be wise to go to some lengths to check whether the value stored into the tenured area really requires a transaction to be recorded. The current test system always records a transaction on a store into the tenured area, regardless of the value stored. However, if memory is tight and many values are stored which do not really require having transactions recorded, our current simplistic transaction recording policy can be detrimental to performance. Storing many transactions causes page faulting as the transaction area needs to be paged in from and out to disk, and scanning the area during a collection causes faults as well.

If memory is plentiful, the issues of the transaction recording policies are not as clear. However, stores which miss the cache (as most stores to the transaction area will) are fairly expensive on the Sparcstations. Unless a large number of assignments in the tenured area need to have transactions recorded, doing an extra check is not going to hurt performance: the check can be done in fewer cycles than the store.

Automatic expansion of the tenured heap based on the frequency of (full) garbage collections can be very important in avoiding pathological garbage collection behavior like that observed for the `reverse` benchmark.

The impact of our simple tenuring policy on the performance of the garbage collector is not clear from our benchmarks and will need further investigation. Premature tenuring is a problem for garbage collection performance because, as the tenured area is getting full (of garbage), side effects to tenured objects will cause the transaction area to fill up more quickly, causing more frequent collections, none of which will remove the tenured garbage.

Imperative Implementation vs. Functional Implementation

In those cases where the programmer has a choice of either an imperative or a functional implementation, the results are ambiguous in how wise it is to choose a functional implementation. In the case of `map`, the functional but iterative `map-2` was as fast as the imperative procedures (with the exception of some pathological cases which can be avoided by making the run-time system smarter). Similarly, the functional `sort-1` procedure performed within 50% of the imperative `sort-2` on small and medium-sized inputs, and therefore seems a viable alternative. Finally, the functional `vector-add-2` procedure was nearly as fast as its imperative counterpart.

However, in the case of `append`, the imperative implementation was significantly better than the functional implementation; the differences were mostly due to overhead associated with stack cache overflows. The functional implementation worked well for small inputs, but very poorly for large. Also, the streams approach of `sieve-5` was four times slower than the traditional `sieve-3`; while this ratio might change if the computations are more substantial than those required for the Sieve benchmark, there is too much overhead to make streams as practical as traditional data structures for "light" computations.

Whenever the programmer can choose between `reverse` and `reverse!`, the choice should be made based on where the data is likely to be located: if the data is old, it may be in the tenured area, and `reverse` will be better. If the data is new, `reverse!` will likely fit the bill. However, `reverse!` is only 60% faster than `reverse` on ephemeral data, and it may be that `reverse` could be used everywhere without undue performance penalty.

Imperative Iteration vs. Functional Recursion

As outlined in the previous section, `append-1` is too expensive for medium and large inputs due to the stack overflow costs. For small inputs, however, `append-1` is still a reasonable choice. Given that the semantics of `append-1` and `append-2` are equivalent in all cases, the implementor of the run-time system should choose `append-2` over `append-1`.

In the case of `map`, the recursive `map-1` suffers from the same problems as does `append-1`, and we must conclude that the imperative implementations are better, as is the functional (and iterative) `map2`.

The `quicksort` benchmark favors a recursive implementation, however. `Quick-1` is faster than any of the other implementations, in particular the imperative `quick-3` and `quick-4`. The recursion is fairly shallow, contributing to the feasibility of a recursive implementation. Any recursion which is not deep enough to overflow the stack cache should have an edge on a similar imperative implementation using an explicit stack.

Explicit Stacks vs. Implicit Stacks

The results of the benchmarks measuring the costs of implicit vs. explicit stacks (`quicksort`, `fib`, and `nfa`) are all in agreement: an implicit stack is always faster than an explicit programmer-managed stack. An explicit stack is not usually much slower than an implicit stack, but the point is that there is not the widely-touted advantage of explicit stacks.

In fact, since the cost of creating a stack frame in any of the test systems is comparable to the cost of doing a `cons` in the generation-scavenging system, and since a `cons` is very expensive in the two other systems, we would expect that a programmer-managed stack which just uses `cons` for the push operation will be very much more expensive under the mostly-generational and stop-and-copy collectors than the implicit stack. This expectation is confirmed by both `fib` and `quicksort`: these simple stacks are very expensive under both the stop-and-copy and mostly-generational collectors.

If we compare the costs of various implementations of explicit stacks, it turns out that the doubly-linked list strategy used by `quick-3` is the better one overall, barely beating the vector strategy used by `quick-4`.

The `nfa` benchmark shows that given our compiler (which was not specially tailored for dealing with continuation-passing style programs) a continuation-passing style is a viable alternative to “standard” recursion. However, for the mostly-generational and stop-and-copy collectors, the extra memory management overhead makes the continuation-passing style too expensive for consideration.

Contradicting these conclusions is the high cost of deep recursions. If a recursion is deep enough to cause a stack cache flush, an explicit-stack strategy may outperform the implicit-stack strategy.

Creating New Data

The `reverse` benchmark shows that `reverse` is faster than `reverse!` on tenured data, and that the relationship for ephemeral data is the opposite of that for tenured data. One should therefore use `reverse!` on newly created data, but if the list being operated on may have been tenured, `reverse` should definitely be used.

The `map` benchmarks show that a functional, iterative style which requires reversing an accumulated list at the end of the calculation can be as fast as an imperative procedure which reverses the list in-place or which creates the correct links as the list is built up. The `append` and `map` benchmarks show that a recursive style is not really viable, however.

Finally, the `vector-add` benchmark shows that the differences in execution speed between creating a result vector at the outset and assigning results into its slots, and creating a list of results and creating a vectpr from that list using the `list->vector` primitive, are small enough that the latter approach can be used anywhere the former can.

Rebinding vs. Assignment

As noted earlier, the `sieve` benchmark was inconclusive in establishing any differences between assignment and rebinding; if anything, we must conclude that rebinding is as inexpensive as assignment.

However, the `quicksort` benchmarks `quick-2` and `quick-5` show us that although the differences are small, `quick-2`, which uses rebinding, is as fast as or faster than `quick-5`, which uses assignment. It therefore seems clear that assignment has no inherent efficiency advantages over rebinding in our simple test programs.

Explicit Data Structures vs. Implicit Data Structures

The results of the `samefringe` benchmark show that generators, despite the overhead of having to call the generator procedure for every element in the data structure, are practical abstraction mechanisms. Most likely, the procedure calling overhead is lost in the cost of actually finding the next element of the data structure hidden by the generator. For the `samefringe` benchmarks, the generator approach and the explicit-stack approach had similar running times.

The `sieve-4` benchmark also implements a generator approach to finding prime numbers. The generator is much faster than the two procedures `sieve-1` and `sieve-2` which keep previously found primes in a list; however, the comparison is not fair because the latter benchmarks are penalized by the ordering of the list of previously found primes. The generator approach implemented by `sieve-4` is slower than both the "streams" program `sieve-5` and the "vector" program `sieve-3`; however, a generator built around `sieve-3` is not unreasonable, and hence the inefficiency of `sieve-4` is due to its unfortunate choice of data representation as a nested procedural structure generating potential primes.

The continuation-passing program `cps-nfa`, which builds up an implicit stack structure by dynamically nesting procedures, is sufficiently efficient to consider a continuation-passing style reasonable efficiency-wise. Whether a continuation-passing style leads to readable programs is outside the scope of this thesis.

The Impact of `Call-With-Current-Continuation`

The results of the `map` and `sort` benchmarks show that for small and medium-sized inputs, and even for large inputs under some circumstances, functional implementations of system procedures which are vulnerable if implemented imperatively is reasonable: performance does not suffer greatly from not being able to use an imperative implementation. However, as has been noted earlier, for large inputs the costs of a functional implementation of e.g. `sort` (but not `map`) can be considerably higher than for equivalent imperative ones.

Deep recursions are affected adversely by the cost of stack cache overflows and underflows, and so it would appear that programs which do not use first class continuations are paying a price. How big a price this is for “typical” programs is unclear. It is also unknown how much time we could save by re-implementing our stack handling code in assembly language (and changing the strategy for allocation slightly, avoiding much copying overhead); an estimate suggests that a factor of four is achievable. However, garbage collection costs may dominate the cost of stack cache overflows.

Finally, it is worth noting that procedures like `map` and `sort` are used hardly at all in the entire run-time library, and only occasionally in the compiler, and inputs to these procedures are usually small.

In conclusion, it seems to the author that the impact on performance due to first class continuations is small. When one considers the benefits of the incremental stack/heap strategy for programs which do use the continuations (and many sophisticated programs do), there is little evidence that the availability of first-class continuations causes major performance loss in the language.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Conclusions

We have investigated a number of issues relating to how different programming styles impact the performance of Scheme programs. Our main objective was to challenge the classical viewpoints that side effects are necessarily cheaper than allocation and that iteration is faster than recursion. We have also investigated the practicality of implicit data structures like generators, and of lazy data structures like streams. We have analyzed the impact of the constraints that the availability of first-class continuations put on the implementation of the runtime system. Finally, we have investigated the impact of three different garbage collection strategies on each of the above mentioned issues from a performance point of view.

Our findings indicate that in general, of the garbage collectors we investigated, only a generation-scavenging collector was able to provide sufficient virtual memory locality and copying performance to support a functional programming style. The other two collectors, a mostly-generational collector and a classical stop-and-copy collector, have either poor copying performance or poor page locality, or both, ultimately making an allocation-hungry programming style irresponsible from a performance point of view. Our findings are relevant primarily in systems where there is a lack of physical memory, but based on copying speed alone, a generation-scavenging collector will outperform the others also in large systems where there is enough physical memory.

Measurements in our system show that a side effect in a generation-scavenging system can be as much as 16 times as expensive as a side effect in a non-generational system.

The following conclusions apply to systems with a generational collector.

We have found that in general, recursion is as efficient as or more efficient than a programmer-managed stack. However, very deeply recursive procedures do not seem to be practical in our current implementation of stack management; for large inputs, an iterative procedure which (say) finally reverses its accumulated result is much more efficient than the corresponding recursive procedure. Furthermore, an iterative functional procedure which reverses its result often will have performance comparable to an iterative imperative procedure which avoids the final reversal or which performs an in-place reversal. In fact, an in-place reversal is almost always more expensive than an allocating reversal if the list being reversed is allocated in the tenured area.

From our experiments we also conclude that when creating a new vector, it is just as efficient to create a list of the elements of the vector and then apply the primitive procedure `list->vector` to the list as it is to create the vector beforehand and assign values to the individual slots of the vector.

Our investigations into implicit data structures show that generators are viable devices for abstracting away the meaning of a data structure, allowing us to write e.g. general procedures for comparing the contents of data structures, secure in the knowledge that there is no inherent performance penalty in doing so.

Lazy data structures like streams have relatively significant overhead costs compared to more traditional data structures like vectors and lists; streams may be practical only when calculating each element of the stream has significant computational cost.

Updating loop variables by parameter passing/rebinding has been found to be no more expensive than updating the loop variables by assignment, and rebinding can sometimes be faster than assignment.

Finally, we have determined that the presence of first-class continuations in Scheme has little effect on the performance of the run-time system and libraries, and that in most user code, the effects will be small as well. However, due to the stack cache which supports first-class continuations, certain penalties are paid for deep recursions, and due to the complicated semantics resulting from mixing side effects with first-class continuations, the cost of certain system procedures like `sort` is higher because these procedures have to be implemented functionally even when an imperative implementation would have been more efficient.

Future Work

The most obvious shortcoming of this work is the lack of larger programs among the benchmarks. All benchmarks were written to measure aspects of performance in an isolated and controlled way, and there were only occasional impacts on the measurements from earlier benchmark programs. The next stage of this research must consider using larger programs for its test suite.

There are other style issues to investigate which we have not even mentioned in the preceding pages. How, for example, should one implement a message-passing object-oriented system for best efficiency? Can such a system be implemented sufficiently well in user code, or are the costs too great? Given that procedure calls are relatively cheap, can or should message passing be implemented as procedure calls?

Further investigations into the costs of streams is definitely necessary; streams are very intriguing data structures, and finding out how costly they are (and how to tune the implementation to support them better) is an interesting topic. Expanding on this idea, it would be interesting to see how the `delay` and `force` procedures, used to implement streams, could find wider applicability, and what their costs are.

Some Scheme systems (although not Larceny, at this point) are able to generate "unsafe" code, i.e. code which performs no type-checking at run-time. The speedups from running unsafe code can in theory be considerable, but we are aware of no published measurements.

We need to experiment with more garbage collectors; for example, a card-marking collector has the advantage over our current generation-scavenging collector that the bitmap it uses for the mark bits, unlike our transaction area, has very good locality, and in addition, the bitmap is not prone to filling up and causing a collection. It may be that assignment-intensive programs (which we scorn) perform better under this collector. There is little evidence that a card-marking collector will beat our current collector on functional programs, but the issue needs investigating.

Another garbage collection issue which needs to be looked closer at is the tenuring policies used by our current collector and how these policies impact performance.

Finally, we need to run our test programs on different hardware (in particular, machines with more physical memory), and we should also run the programs under other Scheme systems to see if our results carry over as well as we would like to think they do.

APPENDIX

BENCHMARK PROGRAMS

The appendix contains listings for all the benchmark programs presented in Chapter V. All scaffolding code has been left out. The reader is referred to Chapter V for a description of the purpose of each program.

```
(define (constest iterations outp info)

  (define (consloop dummy iterations)
    (if (zero? iterations)
        #t
        (consloop (cons 1 1) (- iterations 1))))

  (collect 'tenuring)
  (timeit 'cons info (lambda () (consloop #f iterations)) outp))
```

Figure 25: The `constest` benchmark.

```
(define (vector-add-1 v1 v2)
  (let* ((len (vector-length v1))
         (v3  (make-vector len 0)))
    (let loop ((i 0))
      (if (< i len)
          (begin
            (vector-set! v3 i (+ (vector-ref v1 i) (vector-ref v2 i)))
            (loop (+ i 1)))
          v3))))

(define (vector-add-2 v1 v2)
  (let loop ((i (- (vector-length v1) 1)) (result '()))
    (if (>= i 0)
        (loop (- i 1)
          (cons (+ (vector-ref v1 i) (vector-ref v2 i)) result))
        (list->vector result))))
```

Figure 26: The `vector-add` benchmarks.

```
(define (setcdrtest where iterations ref outp info)

  (define (allocpair)

    (define (loop n)
      (if (zero? n)
          (cons '() '())
          (let ((p (cons '() '())))
            (car p)
            (cdr p)
            (loop (- n 1)))))

    (loop 10000))

  (define (setcdrloop pair otherpair iterations)
    (if (zero? iterations)
        pair
        (begin (car otherpair)
               (set-cdr! pair #f)
               (setcdrloop pair otherpair (- iterations 1)))))

  (let ((pair (allocpair)))
    (collect where)
    (timeit 'setcdr
            (append (list (if (eq? where 'ephemeral) '(edata) '(tdata))
                          (if ref '(cached) '(uncached)))
                    info)
            (lambda ()
              (if (not ref)
                  (setcdrloop pair (cons 1 1) iterations)
                  (setcdrloop pair pair iterations)))
            outp)))
```

Figure 27: The `setcdrtest` benchmark.

```
(define (vector-ref-test vsize iterations type outp info)

  (define (vrefloop2 v iterations)
    (if (negative? iterations)
        #t
         (begin (vector-ref v iterations)
                (vrefloop2 v (- iterations 1)))))

  (define (vrefloop v size iterations)
    (if (zero? iterations)
        #t
         (begin (vrefloop2 v (- size 1))
                (vrefloop v size (- iterations 1)))))

  (let ((v (make-vector vsize 0)))
    (collect type)
    (timeit 'vector-ref
            (append (list (if (eq? type 'ephemeral) '(edata) '(tdata)))
                    info)
            (lambda ()
              (vrefloop v vsize iterations))
            outp)))
```

Figure 28: The `vector-ref-test` benchmark.

```
(define (reverse l)

  (define (r l h)
    (if (null? l)
        h
        (r (cdr l) (cons (car l) h))))

  (r l '()))


(define (reverse! l)

  (define (loop prev curr)
    (if (null? curr)
        prev
        (let ((next (cdr curr)))
          (set-cdr! curr prev)
          (loop curr next))))

  (loop '() l))
```

Figure 29: The `reverse` benchmarks.

```
(define (vector-set-test vsize iterations type ref outp info)

  (define (vsetloop2 v iterations)
    (if (negative? iterations)
        #t
         (begin (vector-set! v iterations #f)
                (vsetloop2 v (- iterations 1)))))

  (define (vsetloop3 v iterations)
    (if (negative? iterations)
        #t
         (begin (vector-ref v iterations)
                (vector-set! v iterations #f)
                (vsetloop3 v (- iterations 1)))))

  (define (vsetloop v size iterations loop)
    (if (zero? iterations)
        #t
         (begin (loop v (- size 1))
                (vsetloop v size (- iterations 1) loop))))

  (let ((v (make-vector vsize 0)))
    (collect type)
    (timeit 'vector-set!
            (append (list (if (eq? type 'ephemeral) '(edata) '(tdata)))
                    info)
            (lambda ()
              (vsetloop v vsize iterations (if ref vsetloop3 vsetloop2)))
            outp)))
```

Figure 30: The `vector-set-test` benchmark.

```
(define (append-1 l1 l2)

  (define (append l1 l2)
    (if (null? l1)
        l2
        (cons (car l1) (append (cdr l1) l2))))

  (append l1 l2))


(define (append-2 l1 l2)

  (define (loop l1 prev head)
    (if (null? l1)
        (begin (set-cdr! prev l2)
               head)
        (let ((q (cons (car l1) #f)))
          (if prev
              (begin (set-cdr! prev q)
                     (loop (cdr l1) q head))
              (loop (cdr l1) q q)))))

  (if (not (null? l1))
      (loop l1 #f #f)
      l2))
```

Figure 31: The `append` benchmarks.

```
(define (stacktest depth iterations info outp)

  (define (stackloop2 depth)
    (if (zero? depth)
        0
        (+ 1 (stackloop2 (- depth 1)))))

  (define (stackloop depth iterations)
    (if (zero? iterations)
        #t
        (begin (stackloop2 depth)
               (stackloop depth (- iterations 1)))))

  (collect 'tenuring)
  (timeit 'stack
          (cons '(slow-exit) info)
          (lambda () (stackloop depth iterations))
          outp))
```

Figure 32: The `stacktest` benchmark.

```
(define (stacktest2 depth iterations info outp)

  (define (stackloop2 depth k)
    (if (zero? depth)
        (k 0)
        (+ 1 (stackloop2 (- depth 1) k))))

  (define (stackloop depth iterations)
    (if (zero? iterations)
        #t
        (begin
          (call-with-current-continuation
           (lambda (k)
             (stackloop2 depth k)))
          (stackloop depth (- iterations 1)))))

  (collect 'tenuring)
  (timeit 'stack
          (cons '(quick-exit) info)
          (lambda () (stackloop depth iterations))
          outp))
```

Figure 33: The `stacktest2` benchmark.

```
(define (map-1 proc list)
  (if (null? list)
      '()
      (cons (proc (car list)) (map-1 proc (cdr list)))))


(define (map-2 proc list)

  (define (map-helper proc list1 list2)
    (if (null? list1)
        (fun-reverse list2)
        (map-helper proc (cdr list1) (cons (proc (car list1)) list2))))

  (map-helper proc list '()))


(define (map-3 proc list)

  (define (map-helper proc list1 list2)
    (if (null? list1)
        (imp-reverse list2)
        (map-helper proc (cdr list1) (cons (proc (car list1)) list2))))

  (map-helper proc list '()))


(define (map-4 proc l)

  (define (map-helper proc l last-pair)
    (if (not (null? l))
        (let ((new (cons (proc (car l)) '())))
          (set-cdr! last-pair new)
          (map-helper proc (cdr l) new))))

  (if (null? l)
      '()
      (let ((first (cons (proc (car l)) '())))
        (map-helper proc (cdr l) first)
        first)))
```

Figure 34: The `map` benchmarks.

```
(define (quick-1 v less?)

  (define (helper left right)
    (if (< left right)
        (let ((median (partition v left right less?)))
          (if (< (- median left) (- right median))
              (begin (helper left (- median 1))
                     (helper (+ median 1) right))
              (begin (helper (+ median 1) right)
                     (helper left (- median 1)))))
        v))

  (helper 0 (- (vector-length v) 1)))


(define (partition v left right less?)
  (let ((mid (vector-ref v right)))

    (define (uploop i)
      (let ((i (+ i 1)))
        (if (and (< i right) (less? (vector-ref v i) mid))
            (uploop i)
            i)))

    (define (downloop j)
      (let ((j (- j 1)))
        (if (and (> j left) (less? mid (vector-ref v j)))
            (downloop j)
            j)))

    (define (ploop i j)
      (let* ((i (uploop i))
             (j (downloop j)))
        (let ((tmp (vector-ref v i)))
          (vector-set! v i (vector-ref v j))
          (vector-set! v j tmp)
          (if (< i j)
              (ploop i j)
              (begin (vector-set! v j (vector-ref v i))
                     (vector-set! v i (vector-ref v right))
                     (vector-set! v right tmp)
                     i)))))

    (ploop (- left 1) right)))
```

Figure 35: The `quick-1` benchmark and partition procedure.

```
(define (quick-2 v less?)

  (define (helper left right lstk rstk)
    (cond ((< left right)
           (let ((median (partition v left right less?)))
             (if (< (- median left) (- right median))
                 (helper left
                         (- median 1)
                         (cons (+ median 1) lstk)
                         (cons right rstk))
                 (helper (+ median 1)
                         right
                         (cons left lstk)
                         (cons (- median 1) rstk)))))
          ((not (null? lstk))
           (helper (car lstk) (car rstk) (cdr lstk) (cdr rstk)))
          (else
           v)))

  (helper 0 (- (vector-length v) 1) '() '()))
```

Figure 36: The quick-2 benchmark.

```
(define (quick-3 v less?)

  (define (helper left right stkp)
    (cond ((< left right)
           (let ((median (partition v left right less?)))
             (if (< (- median left) (- right median))
                 (helper
                   left
                   (- median 1)
                   (begin
                     (if (null? (car stkp))
                         (set-car! stkp (list '()
                                              stkp
                                              (+ median 1)
                                              right))
                         (begin
                           (set-car! (cddr (car stkp)) (+ median 1))
                           (set-car! (cdddr (car stkp)) right)))
                      (car stkp)))
                 (helper
                   (+ median 1)
                   right
                   (begin
                     (if (null? (car stkp))
                         (set-car! stkp (list '()
                                              stkp
                                              left
                                              (- median 1)))
                         (begin
                           (set-car! (cddr (car stkp)) left)
                           (set-car! (cdddr (car stkp)) (- median 1))))
                     (car stkp))))))
          ((not (null? (cadr stkp)))
           (helper (caddr stkp) (cadddr stkp) (cadr stkp)))
          (else
           v)))

  (helper 0 (- (vector-length v) 1) (list '() '() 0 0)))
```

Figure 37: The `quick-3` benchmark.

```
(define (quick-4 v less?)

  (define lstk (make-vector 50 0))
  (define rstk (make-vector 50 0))

  (define (helper left right stkp)
    (cond ((< left right)
           (let ((median (partition v left right less?)))
             (if (< (- median left) (- right median))
                 (helper left
                         (- median 1)
                         (begin (vector-set! lstk stkp (+ median 1))
                                (vector-set! rstk stkp right)
                                (+ stkp 1)))
                 (helper (+ median 1)
                         right
                         (begin (vector-set! lstk stkp left)
                                (vector-set! rstk stkp (- median 1))
                                (+ stkp 1))))))
          ((not (zero? stkp))
           (helper (vector-ref lstk (- stkp 1))
                   (vector-ref rstk (- stkp 1))
                   (- stkp 1)))
          (else
           v)))

  (helper 0 (- (vector-length v) 1) 0))
```

Figure 38: The `quick-4` benchmark.

```
(define (quick-5 v less?)

  (define lstk '())
  (define rstk '())

  (define (helper left right)
    (cond ((< left right)
           (let ((median (partition v left right less?)))
             (if (< (- median left) (- right median))
                 (begin (set! lstk (cons (+ median 1) lstk))
                        (set! rstk (cons right rstk))
                        (helper left (- median 1)))
                 (begin (set! lstk (cons left lstk))
                        (set! rstk (cons (- median 1) rstk))
                        (helper (+ median 1) right)))))
          ((not (null? lstk))
           (let ((left (car lstk))
                 (right (car rstk)))
             (set! lstk (cdr lstk))
             (set! rstk (cdr rstk))
             (helper left right)))
          (else
           v)))

  (helper 0 (- (vector-length v) 1)))
```

Figure 39: The `quick-5` benchmark.

```
(define (sieve-1 n)

  (define primes '())
  (define count 0)
  (define (loop trial)
    (cond ((= count n)
           primes)
          ((prime? trial primes)
           (set! primes (cons trial primes))
           (set! count (+ count 1))
           (loop (+ trial 1)))
          (else
           (loop (+ trial 1)))))

  (reverse (loop 2)))


(define (sieve-2 n)

  (define (loop trial primes count)
    (cond ((= count n)
           primes)
          ((prime? trial primes)
           (loop (+ trial 1) (cons trial primes) (+ count 1)))
          (else
           (loop (+ trial 1) primes count))))

  (reverse (loop 2 '() 0)))


(define (prime? trial oldprimes)
  (or (null? oldprimes)
      (and (not (zero? (remainder trial (car oldprimes))))
           (prime? trial (cdr oldprimes)))))
```

Figure 40: The `sieve-1` and `sieve-2` benchmarks.

```
(define (sieve-3 n)
  (let ((table  (make-vector n))    ; table of computed primes
        (stable (make-vector n))    ; table of squares of primes
        (next   0))                 ; next free element of table

    (define (make-prime-list table n r)
      (if (negative? n)
          r
          (make-prime-list table (- n 1) (cons (vector-ref table n) r))))

    (define (loopi trial)
      (if (= next n)
          (make-prime-list table (- n 1) '())
          (loopj trial 0)))

    (define (loopj trial idx)
      (cond ((or (= idx next) (> (vector-ref stable idx) trial))
             (vector-set! table next trial)
             (vector-set! stable next (* trial trial))
             (set! next (+ next 1))
             (loopi (+ trial 1)))
            ((zero? (remainder trial (vector-ref table idx)))
             (loopi (+ trial 1)))
            (else
             (loopj trial (+ idx 1)))))

    (loopi 2)))
```

Figure 41: The `sieve-3` benchmark.

```
(define (sieve-4 n)

  (define (new-generator next-prime prime)
    (let ((psquare (* prime prime)))

      (define (loop t)
        (if (or (< t psquare) (not (zero? (remainder t prime))))
            t
            (loop (next-prime))))

      (lambda () (loop (next-prime)))))

  (define (prime-loop next-prime primes count)
    (if (< count n)
        (let ((prime (next-prime)))
          (prime-loop (new-generator next-prime prime)
                      (cons prime primes)
                      (+ count 1)))
        primes))

  (reverse (prime-loop (integers 2) '() 0)))


;; Integer generator.

(define (integers n)
  (lambda ()
    (let ((tmp n))
      (set! n (+ n 1))
      tmp)))
```

Figure 42: The `sieve-4` benchmark.

```
(define (sieve-5 n)

  (define (prime-loop stream list count)
    (if (= count n)
        list
        (prime-loop (tail stream) (cons (head stream) list) (+ count 1))))

  (reverse (prime-loop (prime-stream) '() 0)))


;; Make a stream of prime numbers.

(define (prime-stream)

  (define (square x) (* x x))

  (define (prime-filter primes)
    (let ((primes-squared (map-stream square primes)))
      (lambda (trial)

        (define (loop primes primes-squared)
          (or (> (head primes-squared) trial)
              (and (not (zero? (remainder trial (head primes))))
                   (loop (tail primes) (tail primes-squared)))))

        (loop primes primes-squared))))

  (let ((is (integer-stream 2)))
    (cons (head is)
          (delay (filter-stream (prime-filter (prime-stream))
                                (tail is))))))
```

Figure 43: The `sieve-5` benchmark.

```
;; Generic stream procedures

(define (head stream) (car stream))
(define (tail stream) (force (cdr stream)))

(define (filter-stream proc stream)
  (if (proc (head stream))
      (cons (head stream) (delay (filter-stream proc (tail stream))))
      (filter-stream proc (tail stream))))

(define (map-stream proc stream)
  (cons (proc (head stream)) (delay (map-stream proc (tail stream)))))

(define (integer-stream starting-with)
  (cons starting-with (delay (integer-stream (+ starting-with 1)))))
```

Figure 43, continued.

```
;; Standard fib.

(define (fib-1 n)

  (define (fib n)
    (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))

  (fib n))


;; Uses list for stack.

(define (fib-2 n)

  (define (loop stk res)
    (if (null? stk)
        res
        (let ((x (car stk)))
          (if (< x 2)
              (loop (cdr stk) (+ res x))
              (loop (cons (- x 1) (cons (- x 2) (cdr stk))) res)))))

  (loop (list n) 0))


;; Uses vector for stack.
;; The stack never gets deeper than the initial value indicates.

(define (fib-3 n)

  (define (loop stk stkp res)
    (if (negative? stkp)
        res
        (let ((x (vector-ref stk stkp)))
          (if (< x 2)
              (loop stk (- stkp 1) (+ res x))
              (begin (vector-set! stk (+ stkp 1) (- x 1))
                     (vector-set! stk stkp (- x 2))
                     (loop stk (+ stkp 1) res))))))

  (loop (make-vector n n) 0 0))
```

Figure 44: The `fib` benchmarks.

```
(define (recursive-nfa input)

  (define (state0 input)
    (or (state1 input) (state3 input) #f))

  (define (state1 input)
    (and (not (null? input))
         (or (and (char=? (car input) #\a)
                  (state1 (cdr input)))
             (and (char=? (car input) #\c)
                  (state1 input))
             (state2 input))))

  (define (state2 input)
    (and (not (null? input))
         (char=? (car input) #\b)
         (not (null? (cdr input)))
         (char=? (cadr input) #\c)
         (not (null? (cddr input)))
         (char=? (caddr input) #\d)
         'state2))

  (define (state3 input)
    (and (not (null? input))
         (or (and (char=? (car input) #\a)
                  (state3 (cdr input)))
             (state4 input))))

  (define (state4 input)
    (and (not (null? input))
         (char=? (car input) #\b)
         (not (null? (cdr input)))
         (char=? (cadr input) #\c)
         'state4))

  (or (state0 (string->list input))
      'fail))
```

Figure 45: The `recursive-nfa` benchmark.

```
(define (cps-nfa input)
  (define (state0 input fail)
    (state1 input (lambda () (state3 input fail))))
  (define (state1 input fail)
    (cond ((null? input) (fail))
          ((char=? (car input) #\a)
           (state1 (cdr input) (lambda ()
                                 (cond ((char=? (car input) #\c)
                                        (state1 (cdr input) fail))
                                       (else
                                        (state2 input fail))))))
          ((char=? (car input) #\c)
           (state1 (cdr input) fail))
          (else
           (state2 input fail))))
  (define (state2 input fail)
    (if (and (not (null? input))
             (char=? (car input) #\b)
             (not (null? (cdr input)))
             (char=? (cadr input) #\c)
             (not (null? (cddr input)))
             (char=? (caddr input) #\d))
        'state2
        (fail)))
  (define (state3 input fail)
    (cond ((null? input) (fail))
          ((char=? (car input) #\a)
           (state3 (cdr input) (lambda ()
                                 (state4 input fail))))
          (else
           (state4 input fail))))
  (define (state4 input fail)
    (if (and (not (null? input))
             (char=? (car input) #\b)
             (not (null? (cdr input)))
             (char=? (cadr input) #\c))
        'state4
        (fail)))

  (state0 (string->list input) (lambda () 'fail)))
```

Figure 46: The `cps-nfa` benchmark.

```
; auxiliary procedures

(define (leaf n) (list 'leaf n))
(define (branch left right) (list 'branch left right))
(define (leaf? x) (eq? (car x) 'leaf))
(define (branch? x) (eq? (car x) 'branch))
(define (info x) (cadr x))
(define (info! x y) (set-car! (cdr x) y))
(define (left x) (cadr x))
(define (right x) (caddr x))

; samefringe by flattening the trees

(define (samefringe1 t1 t2)
  (define (flatten t)
    (cond ((leaf? t) (list (info t)))
          (else (append (flatten (left t)) (flatten (right t))))))
  (equal? (flatten t1) (flatten t2)))

; samefringe by stacking the traversal state

(define (samefringe2 t1 t2)
  (define (sf2 t1 t2 s1 s2)
    (cond ((branch? t1) (sf2 (left t1) t2 (cons (right t1) s1) s2))
          ((branch? t2) (sf2 t1 (left t2) s1 (cons (right t2) s2)))
          ((and (null? s1) (null? s2)) (= (info t1) (info t2)))
          ((and (not (null? s1)) (not (null? s2)))
           (if (= (info t1) (info t2))
               (sf2 (car s1) (car s2) (cdr s1) (cdr s2))
               #f))
          (else #f)))
  (sf2 t1 t2 '() '()))
```

Figure 47: The `samefringe-1` and `samefringe-2` benchmarks.

```
; samefringe by generators

(define (samefringe3 t1 t2)
  (define (generator t)
    (let ((stack (list t)))
      (define (g)
        (if (null? stack)
            #f
            (let ((hd (car stack))
                  (tl (cdr stack)))
              (if (leaf? hd)
                  (begin
                    (set! stack tl)
                    (info hd))
                  (begin
                    (set! stack (cons (left hd) (cons (right hd) tl)))
                    (g))))))
      g))
  (define (compare g1 g2)
    (let ((x1 (g1))
          (x2 (g2)))
      (cond ((not x1) (not x2))
            ((not x2) #f)
            ((= x1 x2) (compare g1 g2))
            (else #f))))
  (compare (generator t1) (generator t2)))
```

Figure 48: The `samefringe-3` benchmark.

```
;; Merge-sort a sequence according to a predicate.
;;
;; This one alternately merges sorted and reverse-sorted lists to avoid
;; consing all the time and still run in constant stack space.

(define (sort-1 sequence less?)

  (define (m l1 l2 less? h)
    (cond ((and (null? l1) (null? l2))
           h)
          ((null? l1)
           (m l1 (cdr l2) less? (cons (car l2) h)))
          ((null? l2)
           (m (cdr l1) l2 less? (cons (car l1) h)))
          ((less? (car l2) (car l1))
           (m l1 (cdr l2) less? (cons (car l2) h)))
          (else
           (m (cdr l1) l2 less? (cons (car l1) h)))))

  (define (loop2 a b less? notless? fixup?)
    (cond ((null? a)
           (loop1 b notless? less? fixup?))
          ((null? (cdr a))
           (loop1 (cons (reverse (car a)) b) notless? less? fixup?))
          (else
           (loop2 (cddr a)
                  (cons (m (car a) (cadr a) less? '()) b)
                  less?
                  notless?
                  fixup?))))

  (define (loop1 sublists less? notless? fixup?)
    (if (null? (cdr sublists))
        (if fixup?
            (reverse (car sublists))
            (car sublists))
        (loop2 sublists '() less? notless? (not fixup?))))

  (define (notless? a b) (not (less? a b)))

  (if (null? sequence)
      '()
      (loop1 (rsplit sequence less?) notless? less? #t)))
```

Figure 49: The `sort-1` benchmark.

```
;; Sort procedure which copies the input list and then sorts the
;; new list imperatively. Due to Richard O'Keefe; algorithm
;; attributed to D.H.D. Warren

(define sort-2
  (let ()

    (define (sort! seq less?)

      (define (step n)
        (cond ((> n 2)
               (let* ((j (quotient n 2))
                      (a (step j))
                      (k (- n j))
                      (b (step k)))
                 (merge! a b less?)))
              ((= n 2)
               (let ((x (car seq))
                     (y (cadr seq))
                     (p seq))
                 (set! seq (cddr seq))
                 (if (less? y x)
                     (begin
                       (set-car! p y)
                       (set-car! (cdr p) x)))
                 (set-cdr! (cdr p) '())
                 p))
              ((= n 1)
               (let ((p seq))
                 (set! seq (cdr seq))
                 (set-cdr! p '())
                 p))
              (else
               '())))

      (step (length seq)))

    (define (sort-2 seq less?)
      (sort! (append seq '()) less?))

    sort-2))
```

Figure 50: The `sort-2` benchmark.

```
(define (rsplit sequence less?)

  (define (loop sequence lists)

    (define (loop2 sequence i l)
      (cond ((null? sequence)
             (loop '() (cons (cons i l) lists)))
            ((less? (car sequence) i)
             (loop sequence (cons (cons i l) lists)))
            (else
             (loop2 (cdr sequence) (car sequence) (cons i l)))))

    (if (null? sequence)
        lists
        (loop2 (cdr sequence) (car sequence) '())))

  (loop sequence '()))
```

Figure 51: The `rsplit` procedure for `sort-1`.

```
(define (merge! a b less?)

  (define (loop r a b)
    (if (less? (car b) (car a))
        (begin (set-cdr! r b)
               (if (null? (cdr b))
                   (set-cdr! b a)
                   (loop b a (cdr b)) ))
        ;; (car a) <= (car b)
        (begin (set-cdr! r a)
               (if (null? (cdr a))
                   (set-cdr! a b)
                   (loop a (cdr a) b)) )) )

  (cond ((null? a) b)
        ((null? b) a)
        ((less? (car b) (car a))
         (if (null? (cdr b))
             (set-cdr! b a)
             (loop b a (cdr b)))
         b)
        (else                          ; (car a) <= (car b)
         (if (null? (cdr a))
             (set-cdr! a b)
             (loop a (cdr a) b))
         a)))
```

Figure 52: The `merge!` procedure for `sort-2`.

```
#include <stdio.h>
#include <sys/time.h>

typedef struct node {
  short type;
  union {
    int info;
    struct {
      struct node *left;
      struct node *right;
    } branch;
  } u;
} node;

#define LEAF 0
#define BRANCH 1

#define LEAFP( n ) ((n)->type == LEAF)
#define BRANCHP( n ) ((n)->type == BRANCH)
#define LEFT( n ) ((n)->u.branch.left)
#define RIGHT( n ) ((n)->u.branch.right)
#define INFO( n ) ((n)->u.info)

typedef struct intlist {
  int data;
  struct intlist *next;
} intlist;

#ifdef FAST
intlist *freelist = 0;
#endif

extern intlist *flatten();
extern intlist *append();
extern intlist *make_intlist();
extern void free_intlist();
extern node *leaf();
extern node *branch();
extern node *leftwing();
extern node *rightwing();
extern node *rightwing_helper();
```

Figure 53: The `Samefringe-C` benchmark, first part.

```
int main()
{
  node *t1, *t2;
  struct timeval tp1, tp2;
  struct timezone tzp;
  long secs, usecs;
  int i;

  t1 = leftwing( 1000 );
  t2 = rightwing( 1000 );
  printf( "Done creating data.\n" );
  gettimeofday( &tp1, &tzp );

  samefringe1( t1, t2 );

#ifdef FAST
  { intlist *p, *q;

    p = freelist;
    while (p != 0) {
      q = p;
      p = p->next;
      free( q );
    }
    freelist = 0;
  }
#endif
  gettimeofday( &tp2, &tzp );
  secs = 0;
  usecs = (tp2.tv_usec - tp1.tv_usec);
  if (usecs < 0) {
    secs++;
    usecs += 1000000;
  }
  secs += (tp2.tv_sec - tp1.tv_sec);
  printf( "Elapsed time: %lu seconds, %lu microseconds\n",
          secs, usecs );
  return 0;
}
```

Figure 53, continued.

```
/* samefringe by flattening the trees */

int samefringe1( t1, t2 )
node *t1, *t2;
{
  intlist *f1, *f2;
  int r;

  f1 = flatten( t1 );
  f2 = flatten( t2 );
  r = equalp( f1, f2 );
  free_intlist( f1 );
  free_intlist( f2 );
  return r;
}

intlist *flatten( t )
node *t;
{
  intlist *f1, *f2;

  if (LEAFP( t ))
    return make_intlist( INFO( t ) );
  else {
    f1 = flatten( LEFT( t ) );
    f2 = flatten( RIGHT( t ) );
    f2 = append( f1, f2 );
    free_intlist( f1 );
    return f2;
  }
}
```

Figure 53, continued.

```
intlist *make_intlist( n )
int n;
{
  intlist *p;

#ifdef FAST
  if (freelist != 0) {
    p = freelist;
    freelist = freelist->next;
  }
  else
#endif
    p = (intlist *)malloc( sizeof( intlist ) );

  if (p == 0) {
    perror( "malloc" );
    exit( 1 );
  }
  p->next = 0;
  p->data = n;
  return p;
}

void free_intlist( l )
intlist *l;
{
  intlist *n;

  while (l != 0) {
    n = l->next;
#ifdef FAST
    l->next = freelist;
    freelist = l;
#else
    free( l );
#endif
    l = n;
  }
}
```

Figure 53, continued.

```
intlist *append( l1, l2 )
intlist *l1, *l2;
{
  intlist *last, *first, *p;

  if (l1 == 0)
    return l2;
  first = last = make_intlist( l1->data );
  l1 = l1->next;
  while (l1 != 0) {
    p = make_intlist( l1->data );
    last->next = p;
    last = p;
    l1 = l1->next;
  }
  last->next = l2;
  return first;
}

node *leftwing( n )
int n;
{
  if (n == 0)
    return leaf( 0 );
  else
    return branch( leftwing( n - 1 ), leaf( n ) );
}

node *rightwing(  n )
int n;
{
  return rightwing_helper( n, 0 );
}

node *rightwing_helper( n, i )
int n, i;
{
  if (n == 0)
    return leaf( i );
  else
    return branch( leaf( i ), rightwing_helper( n-1, i+1 ) );
}
```

Figure 53, continued.

```
node *leaf( n )
int n;
{
  node *p = (node *) malloc( sizeof( node ) );

  if (p == 0) {
    perror( "malloc" );
    exit( 1 );
  }
  p->type = LEAF;
  INFO( p ) = n;
  return p;
}

node *branch( left, right )
node *left, *right;
{
  node *p = (node *) malloc( sizeof( node ) );

  if (p == 0) {
    perror( "malloc" );
    exit( 1 );
  }
  p->type = BRANCH;
  LEFT( p ) = left;
  RIGHT( p ) = right;
  return p;
}

int equalp( l1, l2 )
intlist *l1, *l2;
{
  while (l1 != 0 && l2 != 0 && l1->data == l2->data) {
    l1 = l1->next;
    l2 = l2->next;
  }
  return l1 == 0 && l2 == 0;
}
```

Figure 53, continued.

```
(define put putprop)

(define (get x y)
  (let ((p (getprop x y)))
    (if p p '())))

(define boyer #f)

(let ()

  (define unify-subst 0)
  (define temp-temp 0)

  (define (add-lemma term)
    (cond ((and (pair? term)
                (eq? (car term)
                     (quote equal))
                (pair? (cadr term)))
           (put (car (cadr term))
                (quote lemmas)
                (cons term (get (car (cadr term)) (quote lemmas)))))
          (else (error "ADD-LEMMA did not like term:  " term))))

  (define (add-lemma-lst lst)
    (cond ((null? lst)
           #t)
          (else (add-lemma (car lst))
                (add-lemma-lst (cdr lst)))))

  ; Hacked this to make work w/ broken expander in Larceny.

  (define (apply-subst alist term)
    (cond ((not (pair? term))
           (set! temp-temp (assq term alist))
           (cond (temp-temp
                  (cdr temp-temp))
                 (else term)))
          (else (cons (car term)
                      (apply-subst-lst alist (cdr term))))))
```

Figure 54: The Boyer benchmark.

```
(define (apply-subst-lst alist lst)
  (cond ((null? lst)
         '())
        (else (cons (apply-subst alist (car lst))
                    (apply-subst-lst alist (cdr lst))))))

(define (falsep x lst)
  (or (equal? x (quote (f)))
      (member x lst)))

(define (one-way-unify term1 term2)
  (begin (set! unify-subst '())
         (one-way-unify1 term1 term2)))

(define (one-way-unify1 term1 term2)
  (cond ((not (pair? term2))
         (set! temp-temp (assq term2 unify-subst))
         (cond ((begin temp-temp)
                (equal? term1 (cdr temp-temp)))
               (else (set! unify-subst (cons (cons term2 term1)
                                             unify-subst))
                     #t)))
        ((not (pair? term1))
         #f)
        ((eq? (car term1)
              (car term2))
         (one-way-unify1-lst (cdr term1)
                             (cdr term2)))
        (else #f)))

(define (one-way-unify1-lst lst1 lst2)
  (cond ((null? lst1)
         #t)
        ((one-way-unify1 (car lst1)
                         (car lst2))
         (one-way-unify1-lst (cdr lst1)
                             (cdr lst2)))
        (else #f)))
```

Figure 54, continued.

```
(define (rewrite term)
  (cond ((not (pair? term))
         term)
        (else (rewrite-with-lemmas (cons (car term)
                                         (rewrite-args (cdr term)))
                                   (get (car term)
                                        (quote lemmas))))))

(define (rewrite-args lst)
  (cond ((null? lst)
         '())
        (else (cons (rewrite (car lst))
                    (rewrite-args (cdr lst))))))

(define (rewrite-with-lemmas term lst)
  (cond ((null? lst)
         term)
        ((one-way-unify term (cadr (car lst)))
         (rewrite (apply-subst unify-subst (caddr (car lst)))))
        (else (rewrite-with-lemmas term (cdr lst)))))

(define (setup)
  (add-lemma-lst
   (quote ((equal (compile form)
                  (reverse (codegen (optimize form)
                                    (nil))))
           (equal (eqp x y)
                  (equal (fix x)
                         (fix y)))
           (equal (greaterp x y)
                  (lessp y x))
           (equal (lesseqp x y)
                  (not (lessp y x)))
           (equal (greatereqp x y)
                  (not (lessp x y)))
           (equal (boolean x)
                  (or (equal x (t))
                      (equal x (f))))
           (equal (iff x y)
                  (and (implies x y)
                       (implies y x)))
```

Figure 54, continued.

```
(equal (even1 x)
       (if (zerop x)
           (t)
           (odd (1- x))))
(equal (countps- l pred)
       (countps-loop l pred (zero)))
(equal (fact- i)
       (fact-loop i 1))
(equal (reverse- x)
       (reverse-loop x (nil)))
(equal (divides x y)
       (zerop (remainder y x)))
(equal (assume-true var alist)
       (cons (cons var (t))
             alist))
(equal (assume-false var alist)
       (cons (cons var (f))
             alist))
(equal (tautology-checker x)
       (tautologyp (normalize x)
                   (nil)))
(equal (falsify x)
       (falsify1 (normalize x)
                 (nil)))
(equal (prime x)
       (and (not (zerop x))
            (not (equal x (add1 (zero))))
            (prime1 x (1- x))))
(equal (and p q)
       (if p (if q (t)
                 (f))
           (f)))
(equal (or p q)
       (if p (t)
           (if q (t)
               (f))
           (f)))
(equal (not p)
       (if p (f)
           (t)))
```

Figure 54, continued.

```
(equal (implies p q)
       (if p (if q (t)
                   (f))
           (t)))
(equal (fix x)
       (if (numberp x)
           x
           (zero)))
(equal (if (if a b c)
           d e)
       (if a (if b d e)
           (if c d e)))
(equal (zerop x)
       (or (equal x (zero))
           (not (numberp x))))
(equal (plus (plus x y)
             z)
       (plus x (plus y z)))
(equal (equal (plus a b)
              (zero))
       (and (zerop a)
            (zerop b)))
(equal (difference x x)
       (zero))
(equal (equal (plus a b)
              (plus a c))
       (equal (fix b)
              (fix c)))
(equal (equal (zero)
              (difference x y))
       (not (lessp y x)))
(equal (equal x (difference x y))
       (and (numberp x)
            (or (equal x (zero))
                (zerop y))))
(equal (meaning (plus-tree (append x y))
                a)
       (plus (meaning (plus-tree x)
                      a)
             (meaning (plus-tree y)
                      a)))
```

Figure 54, continued.

```
(equal (meaning (plus-tree (plus-fringe x))
                a)
       (fix (meaning x a)))
(equal (append (append x y)
               z)
       (append x (append y z)))
(equal (reverse (append a b))
       (append (reverse b)
               (reverse a)))
(equal (times x (plus y z))
       (plus (times x y)
             (times x z)))
(equal (times (times x y)
              z)
       (times x (times y z)))
(equal (equal (times x y)
              (zero))
       (or (zerop x)
           (zerop y)))
(equal (exec (append x y)
             pds envrn)
       (exec y (exec x pds envrn)
             envrn))
(equal (mc-flatten x y)
       (append (flatten x)
               y))
(equal (member x (append a b))
       (or (member x a)
           (member x b)))
(equal (member x (reverse y))
       (member x y))
(equal (length (reverse x))
       (length x))
(equal (member a (intersect b c))
       (and (member a b)
            (member a c)))
(equal (nth (zero)
            i)
       (zero))
```

Figure 54, continued.

```
(equal (exp i (plus j k))
       (times (exp i j)
              (exp i k)))
(equal (exp i (times j k))
       (exp (exp i j)
            k))
(equal (reverse-loop x y)
       (append (reverse x)
               y))
(equal (reverse-loop x (nil))
       (reverse x))
(equal (count-list z (sort-lp x y))
       (plus (count-list z x)
             (count-list z y)))
(equal (equal (append a b)
              (append a c))
       (equal b c))
(equal (plus (remainder x y)
             (times y (quotient x y)))
       (fix x))
(equal (power-eval (big-plus1 l i base)
                   base)
       (plus (power-eval l base)
             i))
(equal (power-eval (big-plus x y i base)
                   base)
       (plus i (plus (power-eval x base)
                     (power-eval y base))))
(equal (remainder y 1)
       (zero))
(equal (lessp (remainder x y)
              y)
       (not (zerop y)))
(equal (remainder x x)
       (zero))
(equal (lessp (quotient i j)
              i)
       (and (not (zerop i))
            (or (zerop j)
                (not (equal j 1)))))
```

Figure 54, continued.

```
(equal (lessp (remainder x y)
              x)
       (and (not (zerop y))
            (not (zerop x))
            (not (lessp x y))))
(equal (power-eval (power-rep i base)
                   base)
       (fix i))
(equal (power-eval (big-plus (power-rep i base)
                             (power-rep j base)
                             (zero)
                             base)
                   base)
       (plus i j))
(equal (gcd x y)
       (gcd y x))
(equal (nth (append a b)
            i)
       (append (nth a i)
               (nth b (difference i (length a)))))
(equal (difference (plus x y)
                   x)
       (fix y))
(equal (difference (plus y x)
                   x)
       (fix y))
(equal (difference (plus x y)
                   (plus x z))
       (difference y z))
(equal (times x (difference c w))
       (difference (times c x)
                   (times w x)))
(equal (remainder (times x z)
                  z)
       (zero))
(equal (difference (plus b (plus a c))
                   a)
       (plus b c))
(equal (difference (add1 (plus y z))
                   z)
       (add1 y))
```

Figure 54, continued.

```
(equal (lessp (plus x y)
              (plus x z))
       (lessp y z))
(equal (lessp (times x z)
              (times y z))
       (and (not (zerop z))
            (lessp x y)))
(equal (lessp y (plus x y))
       (not (zerop x)))
(equal (gcd (times x z)
            (times y z))
       (times z (gcd x y)))
(equal (value (normalize x)
              a)
       (value x a))
(equal (equal (flatten x)
              (cons y (nil)))
       (and (nlistp x)
            (equal x y)))
(equal (listp (gopher x))
       (listp x))
(equal (samefringe x y)
       (equal (flatten x)
              (flatten y)))
(equal (equal (greatest-factor x y)
              (zero))
       (and (or (zerop y)
                (equal y 1))
            (equal x (zero))))
(equal (equal (greatest-factor x y)
              1)
       (equal x 1))
(equal (numberp (greatest-factor x y))
       (not (and (or (zerop y)
                     (equal y 1))
                 (not (numberp x)))))
(equal (times-list (append x y))
       (times (times-list x)
              (times-list y)))
```

Figure 54, continued.

```
(equal (prime-list (append x y))
       (and (prime-list x)
            (prime-list y)))
(equal (equal z (times w z))
       (and (numberp z)
            (or (equal z (zero))
                (equal w 1))))
(equal (greatereqpr x y)
       (not (lessp x y)))
(equal (equal x (times x y))
       (or (equal x (zero))
           (and (numberp x)
                (equal y 1))))
(equal (remainder (times y x)
                  y)
       (zero))
(equal (equal (times a b)
              1)
       (and (not (equal a (zero)))
            (not (equal b (zero)))
            (numberp a)
            (numberp b)
            (equal (1- a)
                   (zero))
            (equal (1- b)
                   (zero))))
(equal (lessp (length (delete x l))
              (length l))
       (member x l))
(equal (sort2 (delete x l))
       (delete x (sort2 l)))
(equal (dsort x)
       (sort2 x))
(equal (length (cons x1
                  (cons x2
                     (cons x3 (cons x4
                        (cons x5
                           (cons x6 x7)))))))
       (plus 6 (length x7)))
```

Figure 54, continued.

```
(equal (difference (add1 (add1 x))
                   2)
       (fix x))
(equal (quotient (plus x (plus x y))
                 2)
       (plus x (quotient y 2)))
(equal (sigma (zero)
              i)
       (quotient (times i (add1 i))
                 2))
(equal (plus x (add1 y))
       (if (numberp y)
           (add1 (plus x y))
           (add1 x)))
(equal (equal (difference x y)
              (difference z y))
       (if (lessp x y)
           (not (lessp y z))
           (if (lessp z y)
               (not (lessp y x))
               (equal (fix x)
                      (fix z)))))
(equal (meaning (plus-tree (delete x y))
                a)
       (if (member x y)
           (difference (meaning (plus-tree y)
                                a)
                       (meaning x a))
           (meaning (plus-tree y)
                    a)))
(equal (times x (add1 y))
       (if (numberp y)
           (plus x (times x y))
           (fix x)))
(equal (nth (nil)
            i)
       (if (zerop i)
           (nil)
           (zero)))
```

Figure 54, continued.

```
(equal (last (append a b))
       (if (listp b)
           (last b)
           (if (listp a)
               (cons (car (last a))
                     b)
               b)))
(equal (equal (lessp x y)
              z)
       (if (lessp x y)
           (equal t z)
           (equal f z)))
(equal (assignment x (append a b))
       (if (assignedp x a)
           (assignment x a)
           (assignment x b)))
(equal (car (gopher x))
       (if (listp x)
           (car (flatten x))
           (zero)))
(equal (flatten (cdr (gopher x)))
       (if (listp x)
           (cdr (flatten x))
           (cons (zero)
                 (nil))))
(equal (quotient (times y x)
                 y)
       (if (zerop y)
           (zero)
           (fix x)))
(equal (get j (set i val mem))
       (if (eqp j i)
           val
           (get j mem)))))))
```

Figure 54, continued.

```
(define (tautologyp x true-lst false-lst)
  (cond ((truep x true-lst)
         #t)
        ((falsep x false-lst)
         #f)
        ((not (pair? x))
         #f)
        ((eq? (car x)
              (quote if))
         (cond ((truep (cadr x)
                       true-lst)
                (tautologyp (caddr x)
                            true-lst false-lst))
               ((falsep (cadr x)
                        false-lst)
                (tautologyp (cadddr x)
                            true-lst false-lst))
               (else (and (tautologyp (caddr x)
                                      (cons (cadr x)
                                            true-lst)
                                      false-lst)
                          (tautologyp (cadddr x)
                                      true-lst
                                      (cons (cadr x)
                                            false-lst))))))
        (else #f)))

(define (tautp x)
  (tautologyp (rewrite x)
              '() '()))
```

Figure 54, continued.

```
(define (test)
  (define ans '())
  (define term '())
  (set! term
        (apply-subst
         (quote ((x f (plus (plus a b)
                            (plus c (zero))))
                 (y f (times (times a b)
                             (plus c d)))
                 (z f (reverse (append (append a b)
                                       (nil))))
                 (u equal (plus a b)
                    (difference x y))
                 (w lessp (remainder a b)
                    (member a (length b)))))
         (quote (implies (and (implies x y)
                              (and (implies y z)
                                   (and (implies z u)
                                        (implies u w))))
                         (implies x w)))))
  (set! ans (tautp term))
  ans)

(define (trans-of-implies n)
  (list (quote implies)
        (trans-of-implies1 n)
        (list (quote implies)
              0 n)))

(define (trans-of-implies1 n)
  (cond ((equal? n 1)
         (list (quote implies)
               0 1))
        (else (list (quote and)
                    (list (quote implies)
                          (- n 1)
                          n)
                    (trans-of-implies1 (- n 1))))))
```

Figure 54, continued.

```
(define (truep x lst)
  (or (equal? x (quote (t)))
      (member x lst)))

(setup)
(set! boyer test)
'boyer)
```

Figure 54, continued.

BIBLIOGRAPHY

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. Structure and Interpretation of Computer Programs. The MIT Press, Cambridge, Massachusetts, 1985

[2] John Backus. Can Programming Be Liberated from the von Neumann Style?. Communications of the ACM, 21, 8, August 1978, 613-641.

[3] H. Barendregt. The Lambda Calculus – Its Syntax and Semantics. North-Holland, Amsterdam, 1981.

[4] C. J. Cheney. A Nonrecursive List Compacting Algorithm. Communications of the ACM, 13, 11, November 1970.

[5] Fred Chow. Minimizing Register Usage at Procedure Calls. In Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation, 85–94. ACM, New York, New York, 1988.

[6] William Clinger, Anne Hartheimer, and Eric Ost. Implementation Strategies for Continuations. Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, 124-131.

[7] L. Peter Deutsch and Daniel G. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. Communications of the ACM, 19, 9, September 1976.

[8] Richard P. Gabriel. Performance and Evaluation of Lisp Systems. The MIT Press, Cambridge, Massachusetts, 1985.

[9] Adele Goldberg and David Robson. Smalltalk-80: The Language Addison Wesley, Menlo Park, California, 1989

[10] Lars Thomas Hansen, William Clinger, and Rémy Evard. Larceny Notes. University of Oregon Computer Science Technical Report, in preparation.

[11] Christopher T. Haynes, et al. IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language. IEEE, New York, New York, 1990.

[12] Gordon Irlam. The Low-level Computational Performance of Sparcstations. Personal communication.

[13] Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley Publishing Company, Reading, Massachusetts, 1973

[14] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. Communications of the ACM, 26, 6, June 1983.

[15] Lightship Software. MacScheme Reference Manual.

[16] John McCarthy. Recursive Functions of Symbolic Expressions. Communications of the ACM, 3, 4, April 1960, 185-195.

[17] John McCarthy, et al. LISP 1.5 Programmer's Manual. The MIT Press, Cambridge, Massachusetts, 1965.

[18] Robin Milner, Mads Tofte, and Robert Harper. The Definition of Standard ML. The MIT Press, 1990.

[19] Robert Allen Shaw. Empirical Analysis of a Lisp System. Ph.D. dissertation, Stanford University, 1988.

[20] P. G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers. B.S. thesis, MIT EECS Dept., 1988.

[21] Sparc International, Inc. The SPARC Architecture Manual, Version 8. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[22] David Ungar. The Design and Evaluation of a High Performance Smalltalk System. Ph.D. dissertation, University of California, Berkeley, 1986.

[23] David Ungar. Generation-Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. ACM SIGPLAN Notices, 19, 5 (May 1984), 157–167.

[24] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "Static-graph" Reorganization to Improve Locality in Garbage-Collected Systems. In Proceedings of the 1991 ACM Conference on Programming Language Design and Implementation. ACM, New York, New York, 1991.

[25] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching Considerations for Generational Garbage Collection. In Proceedings of the 1992 ACM Conference on Lisp and Functional Programming. ACM, New York, New York, 1992.

[26] Paul R. Wilson and Thomas G. Moher. A "Card-Marking" Scheme for Controlling Intergenerational References in Generation-Based Garbage Collection on Stock Hardware. ACM SIGPLAN Notices 24, 5 (May 1989), 87–91.

[27] Benjamin Zorn. Comparative Performance Evaluation of Garbage Collection Algorithms. Ph.D. dissertation, University of California, Berkeley, 1989.

[28] Benjamin Zorn. The Measured Cost of Conservative Garbage Collection. University of Colorado at Boulder Technical Report CU-CS-573-92, April 1992.