Palo Alto Research Center

On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language

Paul Rovner

On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language

Paul Rovner

CSL-84-7 July 1985 [P84-00049]

Abstract: Enough is known now about garbage collection, runtime types, strong-typing, static-checking and concurrency that it is possible to explore what happens when they are combined in a real programming system.

Storage management is one of a few central issues through which one can get a good view of the design of an entire system. Tensions between ease of integration and the need for protection; between generality, simplicity, flexibility, extensibility and efficiency are all manifest when assumptions and attitudes about managing storage are studied. And deep understanding follows best from the analysis of systems that people use to get real work done.

This paper is not for those who seek arguments pro or con about the need for these features in programming systems; such issues are for other papers. This one assumes these features to be good and describes how they combine and interact in Cedar, a programming language and environment designed to help programmers build moderate-sized experimental systems for moderate numbers of people to test and use.

CR Categories and Subject Descriptors: D.2.2 [Software—Tools and Techniques], D.2.6 [Software—Programming Environments], D.3.3 [Programming Languages—Language Constructs], D.4.2 [Operating Systems—Storage Management].

Additional Keywords and Phrases: programming languages, types, garbage collection, concurrency, reference counting.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. The Cedar Project

1.1 Goals and a Brief History of Storage Management in Cedar

The primary goal of the Cedar effort was to achieve major improvements in the costs of producing reliable, high-quality software. The Cedar language is derived from Mesa, which is a compiled, production-quality system implementation language in the Pascal family with extensive facilities for static-checking, error-handling, modularity, separate compilation and light-weight processes [16]. Mesa and Cedar are the result of an evolutionary research and development effort in programming languages and systems at Xerox PARC that spans more than a decade.

Work on Cedar began in 1978 with two studies [8, 10]: a preliminary one to gather and discuss issues, features, priorities and opportunities based on experience with the Lisp [19], Mesa and Smalltalk [9, 11] programming environments at Xerox PARC, and a follow-up study to look more closely at design tradeoffs between a Lisp starting point and a Mesa one. A Mesa base was then chosen and detailed design and implementation work began in mid-1979.

Experience with Lisp and Smalltalk had taught us that the use of a single virtual address space enables the rapid development of highly integrated applications that share packages and that present a standard, ubiquitous user interface. Experience with Mesa had demonstrated the additional value of a single virtual address space for concurrent applications that share packages and that can be built on each other. It was clearly important for a human user to be able to deal with several concurrently active tasks at once, shifting his attention between them as desired (such as editing, compiling, debugging, reading or sending mail, printing a file, or using clock-driven tools such as a reminder system).

Cedar had to manage shared resources well enough to support a disparate mix of ambitious applications; efficient, non-disruptive, concurrent garbage collection was seen as a crucial requirement. It was our goal to achieve a dramatic decrease in the relative cost of managing storage compared to other programming tasks.

A reference-counting scheme much like the one in Interlisp [7] was designed, built, and used for early versions of Cedar, but was abandoned later in favor of the design outlined here. The earlier design was based on the use of a hash table to hold the reference-counts of collectible objects. Though with this scheme the memory accesses required by a reference-counted assignment had good locality, the execution cost was too high; we were able to cut it by roughly 60%. In addition, the Interlisp scheme was overly complex; this limited our ability to make changes or port the system to other machines. It also had two serious problems: a fixed-size reference-count table and fixed-size reference-count table entries. Solving these problems would have required adding still more complexity. Our experience indicates that the page-fault behavior of the design outlined here is comparable to that of the Interlisp scheme. It should be noted that this conclusion is based on the apparent behavior of the system to its regular users, not on precise measurements.

A word about tactics: Storage management in Mesa is done via language features that allow a program to construct, modify and manipulate pointers. It is effectively impossible for Mesa to protect against a memory smash caused by a buggy program. It was necessary to identify these features and exclude them from a "storage-safe" subset of Cedar for which garbage collection is possible. Though most new code for Cedar was written in the safe subset, it was still possible for debugged Mesa code that used unsafe features to run in Cedar. This allowed Cedar to evolve from a working compiler and runtime system.

1.2 System Overview

The current Cedar system includes the Cedar language [13]; communication software that uses the Ethernet [15] to provide a "remote procedure call" facility [1] and a distributed file system [3]; a comprehensive suite of integrated packages for graphics [21], screen management, input/output, and text manipulation; a user-extendible executive; debugging facilities based on an abstract machine model; and facilities for managing multiple versions of large software components [17]. Cedar runs on the Dorado, a high-performance personal computer [5, 14] and on other Xerox computers. The system is currently being used by about 80 computer professionals at the Xerox Palo Alto Research Center and elsewhere within Xerox for daily work. Development continues on various experimental prototypes such as a database system [4], a calendar system, a mail system, three-dimensional graphics applications, color graphics applications, some CAD tools, an experimental system for organizing information spatially, and several distributed applications based on the remote procedure call facility. Cedar and its applications comprise over 500,000 lines of source program and 3000 files, and represent roughly 60 person-years of effort over a 6 year period. An overview of a recent version of Cedar from a user's point of view can be found in [20].

2. Language Extensions For Garbage Collection

The most profound extensions of Mesa for Cedar are ones to enable automatic storage de-allocation and the manipulation of objects whose types are not known until runtime. This section contains a brief overview of these aspects of the Cedar language. For more complete language descriptions of Mesa and Cedar, the reader is directed to [16] and [13].

A word about the notation used below will help. In the examples, keywords are shown in SMALL CAPS. ":" should be pronounced "has type", "←" as "gets", and "[" as "of" or "applied to."

2.1 Language Forms Added to Mesa to Support Collectible Storage

2.1.1 REF: a type constructor for declaring "pointer to collectible object" types.

The declaration:

```
        i: REF INT;
```

defines i to be a variable of type REF INT. The value of i is a pointer to a collectible object that holds INT values. Pointers to collectible objects are called "refs". For any Cedar type T, one can define a type REF T. The initial value of i is NIL.

2.1.2 *REF ANY: a type for "pointer to collectible object of a type not known until run-time."*

The declaration:

```
        r: REF ANY;
```

defines r to be a variable of type REF ANY. The value of r can be any ref. It is initially NIL. REF ANY provides a way for the programmer to delay complete specification of the type of an operand until execution-time without giving up either type-safety or execution efficiency.

2.1.3 *NIL: a built-in value that conforms to any REF type.*

NIL is the default initial value of ref variables.
For example, the declarations:

```
        r: REF ANY;
        s: REF INT ← NIL;
```

defines r and s to be ref variables with initial value NIL.

2.1.4 *NEW: an expression for creating new collectible objects at runtime.*

For example,

```
        i ← NEW[INT ← 6];
```

is a statement that allocates a new collectible object to hold INT values, initializes this value to 6, and binds the variable i to a ref to the new object.

2.1.5 *REF literals*

Cedar programmers need efficient, flexible and convenient ways of dealing with character strings. We have found three sets of features to be useful: those associated with immutable strings ("ROPEs"), mutable strings ("REF TEXTs") and unique strings ("ATOMs"). The use of ROPEs is most common: sharing is easy and concatenation and substring operations are efficient. REF TEXTs are more efficient and convenient than ROPEs when constructing a string one character at a time. ATOMs are more efficient and convenient than the others when many string equality comparisons are needed.

Mesa was extended to include ROPE, TEXT and ATOM as built-in types and to recognize quoted strings of characters as literals of type REF TEXT or ROPE (depending on the surrounding context) and $<identifier> as an ATOM literal.

2.2 Language Forms Added to Mesa to Support REF ANY

Like Pascal, Mesa is a strongly-typed language that is defined to make compile-time type checking easy. Mesa requires that the type of each expression and variable be specified completely at compile time though it does provide an escape mechanism to disable type checking altogether. Strong-typing helps facilitate both the detection of programming errors at compile time and the production of efficient code. On the other hand, this restriction often makes programming difficult. It is not possible in Mesa to define a package of functions for manipulating arrays without specifying the type of the array element. One must define a different array package for each kind of array.

In principle, of course, one should be able to define a function only in terms of those properties of its arguments which are necessary for correct behavior. This was a Cedar goal that has not been fully achieved, although REF ANY and the related language forms listed below were added to Mesa as a small step in this direction. The functions listed below had to be built-in because they are polymorphic.

Any ref can be passed as a REF ANY value, assigned to a REF ANY variable, etc. The three language forms shown below are used at execution time to verify that the actual type of the referenced object conforms to the assumptions of the program.

2.2.1 ISTYPE, NARROW

ISTYPE is a BOOLEAN-valued function that takes a REF ANY value (r) and a type expression for a ref type (REF T) and returns TRUE if the object referenced by r has type T, FALSE otherwise.

NARROW is a function that takes a REF ANY value (r) and a type expression for a ref type (REF T). NARROW returns r as a REF T value if ISTYPE[r, REF T]. Otherwise, NARROW raises an error.

For example,

```
r: REF ANY ← NEW[INT ← 6];
        declare a new variable named r of type REF INT and set its initial value to be a ref to an
        INT with initial value = 6
b1: BOOL ← ISTYPE[r, REF INT]; -- sets b1 TRUE
b2: BOOL ← ISTYPE[r, REF REAL];-- sets b2 FALSE
s: REF INT ← NARROW[r]; -- sets s = r. The second argument (REF INT) is implicit.
t: REF REAL ← NARROW[r]; -- raises an error
```

2.2.2 WITH <REF ANY> SELECT FROM ...

This is a language form similar to the one in Mesa for variant record discrimination. The example illustrates selection on the type of r from among REF INT, REF REAL or any other following type. Keywords are shown in all caps; refINT and refREAL are variables with scopes delimited by <statement 1> and <statement 2>, respectively.

```
WITH r SELECT FROM
        refINT: REF INT => <statement 1>; --refINT is defined (= r) within statement 1
        refREAL: REF REAL => <statement 2>; --refREAL is defined (= r) within statement 2
        ENDCASE => <statement 3>;
```

Logically, this example could be expressed using ISTYPE and NARROW as follows:

```
IF ISTYPE[r, REF INT]
      THEN {refINT:  REF INT = NARROW[r]; <statement 1>}
      ELSE IF ISTYPE[r, REF REAL]
      THEN {refREAL:  REF REAL = NARROW[r]; <statement 2>}
      ELSE <statement 3>;
```

2.3 Language Forms Added to Mesa to Support Lists

Extensive experience with Lisp over many years has demonstrated the power both of list structure as a representation technique and of high-level language primitives for manipulating lists. Lists help programmers think.

The best way to add lists to Mesa from a language design standpoint would have been to solve the problems of extending the Mesa type system to accommodate type parameters and subclassing. Polymorphic operations like ISTYPE and ones for lists could then be defined using the language, rather than having to be built-in as primitives. But it's hard to solve all problems simultaneously; an overhaul of the Mesa type system was left as a problem for the future. Though the language extensions described below are inadequate to support generic list-processing, they are worthwhile in the meantime.

Cedar provides list-processing facilities similar to those in Lisp, but different in that the type of each Cedar list variable or expression is known to the compiler. The Mesa language was extended to include the forms described below.

2.3.1 LIST OF <type> and LIST[<element1>, <element2>, ...]

LIST OF is a type constructor. LIST[...] builds a new list. Some examples are:

```
listOfAtom: TYPE = LIST OF ATOM;  -- defines a new type named listOfAtom

loa: listOfAtom; -- declare a variable named loa to be a list of ATOM's
loint: LIST OF INT; -- declare a variable named loint to be a list of integers

loa ← LIST[$atom1, $atom2, $atom3]; -- assign a three element list of atoms to loa
loint ← LIST[1, 2, 3]; -- assign a three element list of integers to loint
```

2.3.2 list.first, list.rest

A value of a type LIST OF T is a REF to a record that has two fields: one named "first" having type T and one named "rest" having type LIST OF T (think CAR and CDR). Some examples are:

```
a: ATOM = loa.first; -- the value of a = $atom1
b: ATOM = loa.rest.rest.first; -- the value of a = $atom3
i: INT = loint.rest.first; -- the value of i = 2
```

2.3.3 CONS[<element>, <list>]

CONS[<element>, <list>] builds a new list (l) with l.first = <element> and l.rest = <list>. An example

loa ← CONS[$atom4, loa];

assign a four element list [$atom4, $atom1, $atom2, $atom3] of atoms to loa

2.4 Implementation

The compiler translates each NEW expression into an ALLOCATE instruction followed by a sequence of instructions to initialize the new object. The ALLOCATE instruction takes two parameters:

size: the number of storage cells needed for the object, and

typeTag: a one-word value to be associated with the object.

The typeTag parameter is an index to a vector of "type descriptors" that is maintained by the Cedar runtime software. Each type descriptor includes three pieces of information:

1. A "ref-containing map" that locates REFs within objects having this typeTag. This is used by the garbage collector.

2. A value that characterizes all types equivalent to this one (a "type string"; see Section 6 for more detail). This is used to support the REF ANY feature.

3. Information sufficient to locate a symbolic description of the type in a compiler-produced symbol table. This is used to support the Cedar interpreter and debugger.

TypeTags are stored in object headers. TypeTags have the property that they are equal if and only if the types they represent are equivalent (see Section 6 for more detail). This enables the ISTYPE function to be implemented as a one-word equality test. Accordingly, NARROW and WITH ... SELECT are fast and simple.

Newly allocated objects are cleared to all zeros (the representation of NIL) to avoid problems that would arise if the collector unwittingly examines an uninitialized ref-containing object. This could happen if a client process is interrupted (or destroyed) while in the middle of initializing a new object and a collection then occurs. In an attempt to avoid the cost of clearing all new objects we considered synchronizing the collector with type-specific initialization actions, but found the costs, both in performance and complexity, to outweigh the advantages.

An ability to specify and develop the parts of a program separately is crucial to the successful management of a large software project. Mesa, hence Cedar, enables separate compilation of the parts of a program.

The Cedar runtime loader makes the linkages from symbolic descriptions in an object module to corresponding internal descriptors in the runtime system. For each program module, REF literals and typeTag parameters are acquired by the loader during its linking phase and stored in a "links" region of the program module's data area. The location of this region is known to the compiled code. The compiler constructs a special segment of the object module to hold unrelocated type descriptors and the text of REF literals. For each type descriptor, the loader performs a lookup in a runtime data structure that associates type descriptors with typeTags. Type descriptors not found are entered and assigned new typeTags. For each TEXT or ROPE literal, a new object is created. A hash-coded table is used to associate ATOM literals with a corresponding unique ATOM descriptor.

3. Allocation

Any allocation scheme having proper synchronization between the collector and multiple concurrent client programs would work for Cedar. The major design issue is performance, of course. Allocation should be fast and require small storage overhead, although the cost of an allocation can be roughly proportional to the size of the request. Reference-counted assignment should be fast. The work required by the reference-counting collector to reclaim an object should be small. Operations that determine the size and type of an object should be fast. It should be possible to implement a stop-the-world, trace-and-sweep collector that runs in a reasonable amount of time and requires only a modest amount of storage space.

Every collectible object in Cedar has a header. One word of the header carries the object's typeTag; this is 0 if the object is on a free list. Another word carries a coded size field, a reference-count field and four tag bits named "maybeOnStack", "rcOverflowed", "onZCT" and "finalizationEnabled." The headers of small objects have only these two words; the headers of larger objects have an additional two words.

For small objects, the Cedar allocator uses a vector of free lists, each a chain of blocks of uniform size. Block sizes are quantized to decrease fragmentation, but the quantization grain is fine enough to keep expected excess storage in a block to under 10% of the requested size. Indices into this vector (called "block size indices") are stored in object headers to identify both the size of the object and the free list from which it came and to which it should be returned by the collector. This enables use of a small size field in object headers. Table lookup is used to convert between raw size, quantized size and block size index. When a free list is found to be empty, a sequence of pages is acquired from the operating system's VM allocator. This has sufficient length to be carved up into an integral number of blocks of the desired size. Such blocks are never subdivided or merged with others.

For medium-sized objects, a modified first-fit algorithm using a doubly-linked free list of variable-sized blocks is used. For each large object, Cedar utilizes a sequence of pages that is acquired directly from the operating system's VM allocator. Both medium-sized and large objects require extended headers to store their lengths; a special block size index is used to identify such objects. Recent experience with Cedar indicates that most programs seldom, if ever, require a medium-sized allocator, and that the Cedar algorithm would cause excessive VM fragmentation if it were used more heavily. The medium-sized allocator would benefit from further development.

The Cedar allocator maintains a vector of bits, one for each page in VM, called the "allocation map." The bit for a page is set TRUE if the page is assigned to the allocator. This is used by the trace-and-sweep collector both to determine whether a bit pattern found on the active stack could be a ref that should be traced, and to enumerate all objects during its sweep phase.

Critical allocation and reclamation operations are microcoded to improve performance. These either run to completion (i.e., are atomic with respect to all other operations) or they trap to equivalent software procedures to handle unusual situations (for example when a free list is found to be empty). If a trap does occur, the microcode has made no changes.

Concurrency is managed in software through the use of Mesa monitors and condition variables. The method outlined below enables both low-cost execution via microcode in the simple, normal case and flexible, synchronized load-sharing between microcode and software in unusual, more complex cases. Similar techniques can be used in a concurrent execution environment to do incremental development or testing of an ensemble of microcode and software routines that require synchronization and mutual exclusion.

The first instruction executed in each ENTRY procedure of the allocator's monitor (after the monitor lock is acquired) is a call on a microcoded routine that sets a flag, call it "F." The last instruction executed before releasing the monitor lock is a call on a microcoded routine that resets F. Microcode operations that must be atomic with respect to the monitor will test F on entry; if it is set, the microcode traps to software that waits until the monitor is unlocked and then re-tries the operation. Thus, while a process is inside the allocator's monitor (i.e., in one of its software ENTRY procedures), competing microcode operations will trap. In the normal case, a trap will not occur, but this test can be made by the microcode without having to access the memory to read the state of the monitor lock.

A different trap to software is caused when a microcode operation encounters a situation that requires software assistance. Typically, the software trap handler will take some action to solve the problem (such as adding new blocks to an empty free list) and then re-try the operation. Solving the problem may require use of an ENTRY procedure, but this causes no difficulty.

4. Garbage Collection

The distinguishing characteristic of good garbage collection is that under normal circumstances you never notice it. It certainly should not interrupt the user or cause a noticeable suspension of system activity or a degradation of system performance. It should be extremely reliable. And the cost of garbage collection (storage overhead, execution time, working set requirements) should be comparable to the cost of manual storage management.

Cedar provides both a concurrent reference-counting collector that runs in the background when needed and a pre-emptive, conventional "trace-and-sweep" collector that can be invoked explicitly by the user to reclaim circular data structures which cannot be reclaimed by the reference-counting collector.

Both collectors treat procedure-call activation records (called frames) "conservatively"; that is, they assume that every ref-sized bit pattern found in a frame might be a ref, hence its referent might be accessible. An object referenced from a frame will not be reclaimed. For the reference-counting collector, this trick eliminates the need to count references in frames, a substantial optimization. For the trace-and-sweep collector, this eliminates the need to locate refs in frames, which would require much more complex and expensive synchronization between the trace-and-sweep collector and compiled code, not to mention a major overhaul of the compiler and parts of the microcode.

The conservative collection scheme can cause unnecessary retention. Extensive experiments were done under a variety of execution circumstances to determine an upper bound on the amount of such retention. These indicated that excessive retention is sufficiently unusual to be insignificant. Uncertainty is more of a problem (see Section 5).

4.1 Reference Counting

4.1.1 Design

The simplest reference-counting GC scheme would be to count every reference to each object. Whenever an object's reference count fell to zero it would be reclaimed (and the count of each object referenced directly from it would be decreased by one). This is the scheme used in Smalltalk. As we considered whether to do this for Cedar, we made the following observations:

1. A reference-counted assignment is significantly more expensive than an ordinary assignment.

2. Far more assignments are made to cells of procedure frames (i.e., local variables) than to cells of allocated objects.

3. The cost of clearing new procedure frames is high.

4. Collection is required sufficiently seldom that a modest fixed cost to get it started is acceptable.

5. It would be difficult to change the Cedar compiler and microcode to enable the collector to determine the location of refs in procedure frames at acceptable cost. Such a change would surely increase the cost of procedure calls and the complexity of both the compiler and the runtime system.

Based on these observations and the ideas in [7], the following design decisions were made:

1. Refs residing in procedure frames are not counted. Refs residing in allocated objects and in the data areas of program modules are counted.

2. Garbage collection occurs at intervals, based on the amount of storage allocated since the previous collection. The size of this "allocation interval" (measured in words of storage) can be changed under program control. The collector is also invoked automatically when its tables begin to fill or when the amount of available virtual memory drops below a pre-defined threshold.

3. At the beginning of each collection, a snapshot is made of all procedure frames. Objects referenced from the snapshot are not reclaimed during that collection.

4. The snapshot of procedure frames is treated "conservatively"; that is, if a bit pattern that might be a ref is found in the snapshot, such an object is assumed to be referenced from a frame

and will therefore not be reclaimed.

Concurrency is used to minimize disruption of client programs by the collector. Client processes are able to continue allocating new objects while the collector is running, except for the short time (a few tens of milliseconds) required at the beginning of the collection to snapshot procedure frames. To keep the allocator from getting too far ahead of the collector, a strategy for balancing allocation activity with collection activity is employed. An allocating process is suspended until the current collection finishes if that process attempts to allocate more than double the "allocation interval" while the collector is running. This simple strategy works well in practice.

4.1.2 Implementation

The reference count for each object is stored in a field of its header. The reference count of each newly allocated object is initialized to zero. There is a mechanism for dealing with reference counts that grow too large for the field.

The compiler treats ref assignments to local variables of procedures as normal assignments (no reference counting). For ref assignments to cells of allocated objects or to variables in program data areas, the compiler emits a special "reference-counted assignment" opcode.

When a new object is created or a reference count makes the transition from 1 to 0, a pointer to the object is placed in the next slot of a "zero-count table." A flag in the object header is used to avoid making duplicate entries in the table.

The zero-count table consists of a sequence of fixed-size blocks of storage that are linked, the end of one connecting to the beginning of the next. The last block in this chain ends with a NIL link. The zero-count table is referenced via two variables: a read pointer and a write pointer. The collector examines and modifies the read pointer and examines the write pointer; the allocator and reference-count machinery examine and modify the write pointer. There is at most one active collector process, but there may be many concurrent processes allocating objects and changing reference counts.

Here is a basic description of the reference-counting collector: The collector begins by making a snapshot (i.e., a copy with interrupts disabled) of all procedure frames. It then scans the copied frames for bit patterns that might be refs; for each that it finds, it makes an entry in a hash table called the "FoundOnStack table (FOSTable)." This table is used by the collector to help determine whether an object being considered for reclamation might be referenced from an active procedure frame. To decrease bogus retention, a new hash function is used for each collection.

Each FOSTable entry contains the logical OR of all refs that hash to that table address. An object will not be reclaimed if its ref hashes to an entry that includes the ref. A logical AND (of the ref and the entry) is used to check for inclusion. This scheme is simpler than using chaining or a secondary hash, but increases the "conservatism" of the collector. Experience based on the use of the system indicates that increased retention is at worst a minor problem.

After building the FOSTable, the collector scans the zero-count table, incrementing the read pointer as it goes, until the write pointer is reached. It looks at the object referenced by each entry.

There are four cases: if the object's reference-count is greater than zero, the scan continues. If the object might be referenced from an active procedure frame, the entry is copied to the end of the zero-count table and the scan continues. If the object is enabled for finalization, it is put on its type's finalization queue if there is space for it; if not, the entry is copied to the end of the zero-count table (see Section 5 for more detail). If the object is not suitable for finalization, it is reclaimed. When the collector finishes scanning a block of the zero-count table, it moves the block to the end of the chain.

4.2 Reclamation

The reclaimer avoids recursion and uses no extra storage. Processing of an object for reclamation proceeds as follows: The ref-containing map associated with the object's type is used to locate refs within the object. For each such ref, the reclaimer decreases the count of the referenced object by one (we call this object a "discovered" object). If the result is zero, then this discovered object must be considered for reclamation as well.

A backward-pointing list of stacks is used to remember discovered objects for subsequent reclamation, in this manner: The reclaimer uses storage in the object being reclaimed to remember pointers to discovered objects. These pointers are stored in a block at the top of the object. This block, together with a pointer to its last entry, is used as a "current stack of pending actions" by the reclaimer. The reclaimer iterates by popping a pointer from its current stack and processing the discovered object for reclamation. This may require construction of another stack in the discovered object. If so, a pointer to the last entry in the current stack is stored as the first item of the new block in the discovered object. Such a pointer is marked to distinguish it from a pointer to a discovered object. When the current stack is exhausted, if the first item is marked as a pointer to the last entry in another stack, the object containing the current stack is put on the free list, the other stack becomes the current one and the reclaimer continues. If the first item is not so marked, the object containing the current stack is put on the free list and the reclaimer returns.

4.3 The Trace-And-Sweep Collector

The main purpose of this collector is to reclaim inaccessible circular structures missed by the reference-counting collector. We chose to use a conventional stop-the-world garbage collector [6] to solve this problem rather than one of the techniques proposed in the literature [2, 18] because its implementation is simple and we were unsure about the severity of the problem.

In retrospect, inaccessible circular structures have been more of a problem in Cedar than we expected. Some programs produce much circular garbage, and it is often difficult to know that a given program will do so or to "fix" it not to do so. The trace-and-sweep collector is used regularly by a few people. Though it is robust and reliable, its use is sufficiently disruptive to be more than an annoyance (it takes between thirty seconds and several minutes to run). An incremental, background version would be useful.

The existence of a trace-and-sweep collector has had several beneficial side effects. It was easy to adapt it to help study patterns of storage utilization. Experiments included placing an upper bound on the amount of storage retained by the conservative scan, discovering where refs to specified objects were stored, and finding objects that were referenced solely within inaccessible circular structures. Integration of the trace-and-sweep collector also served to uncover problems in the structure of related software; the current system is better organized and more modular as a result.

The trace-and-sweep collector begins by acquiring the monitor locks required to guarantee that refs in counted storage will not move while tracing and sweeping are underway. Deadlock avoidance is the name of the game. The software must be structured to accommodate this: locks must be acquired in the right order, and facilities needed by the collector must be available.

The algorithm is straightforward and conventional; rather than detailing it here I list a few of its more salient features below.

Storage for a stack is required by the trace phase to remember objects that remain to be traced. This storage comes from the operating system's VM allocator. The onZCT flag in the object header is used as a mark bit. The collector reconstructs free lists, the zero-count table and all reference-counts, and leaves free lists sorted by VM address. It does not compactify, merge, or relocate objects. It can return runs of free pages to the VM allocator rather than putting them back on free lists.

The trace-and-sweep collector treats active frames conservatively, like the reference-counting collector. But it does not use the FOSTable because the collector must know with certainty whether a potential ref found in a frame is valid (the trace-and-sweep collector is therefore less conservative than the reference-counting one). Certainty is required because the collector must trace from such objects. Ref validity is determined by scanning the smallest region of VM that encloses the object and has been assigned to the allocator; the page map is used for this purpose.

5. Finalization

It is often useful to specify actions to take when an object of a particular type is no longer accessible to any client of the package that created it. Such "package finalization" actions might include (for example) removal of the object from a cache that the package maintains.

In a world without garbage collection, special-purpose finalization actions are performed by the same application-dependent code required to explicitly free the object; deciding when to finalize is not an issue. In a world with garbage collection, explicit de-allocation is not required but the need for finalization remains. Application-dependent arrangements for deciding when to finalize are usually inconvenient, expensive, error-prone and they duplicate in one way or another functions that the collector provides; they defeat its purpose. Like automatic storage management, automatic finalization is more than a convenience.

Cedar provides an automatic finalization mechanism that is efficient, type-safe and storage-safe, but uncertain. Though one can imagine more general facilities for finalization, we have found that the features described below are a useful and powerful set.

In Cedar, finalization is associated with types. If the objects of a particular type are to be finalized, the program identifies that type and specifies a "finalization queue" and a number ("npr") of "package references."

After each new object of the type is created and its package references are established (i.e., it has been entered in the package's data structure, if any), the program makes an explicit call to "enable" the object for finalization. This causes the "finalizationEnabled" flag to be set in the object's header and its reference count to be decremented by npr. When on a subsequent garbage collection the object is found to have a reference-count of zero, finalizationEnabled set, and that it is not referenced from an active procedure frame, then it is placed on the type's finalization queue. The object's finalizationEnabled flag is reset at this time and its reference-count is incremented by npr. Of course, any object found by the collector to have no references and not to be enabled for finalization will simply be reclaimed, e.g., sometime after the client finalization action removes the object from its package data structure.

A queue rather than a callback procedure is used for finalization. This decouples the collector from the dangers of unreliable client code. For one thing, the collector process is restricted in various ways (e.g., it can not use the allocator); for another, it would not be good to disable the collector. Decoupling via a queue allows client finalization code to use any Cedar feature. Finalization queues are of fixed length because the collector cannot extend them; if a queue is found by the collector to be full, the object that would be queued is simply put back on the zero-count table. The next collection will try again to queue it.

The use of a conservative garbage collection strategy affects the reliability of the finalization mechanism: objects that are not referenced by client code may not get finalized. For this reason, finalization should be viewed as a performance enhancement. The correctness of client programs must not depend on finalization.

Typically, client finalization code operates from an infinite loop in a dedicated process that waits until a ref is put on the queue by the collector, performs the finalization action on the specified object, then goes back to await the next. Such a process is inactive most of the time.

Looking a little more carefully at usage, there appear to be three kinds of applications for finalization: ones with no package data structure, ones with a package data structure that is accessed only during object creation or object finalization, and ones with a package data structure that is used as a cache from which copies of refs to existing objects are handed out to clients. Finalization for the first two applications is straightforward, but the required synchronization is somewhat subtle for caching applications. For these, finalization actions must be synchronized with access to the cache. The following example illustrates what is believed to be a correct framework for such applications:

```
table: HashTable = <etc>; -- *the package data structure*

myQueue: FinalizationQueue = <create a new finalization queue>;

EstablishFinalization[type: <type of Object>, npr: <whatever>, fq: myQueue];
FORK[Finalizer[]];

Finalizer: PROC = { -- the finalization process
    DO InternalFinalizer[Pull[myQueue]] ENDLOOP;
    };

InternalFinalizer: ENTRY PROC[handle: REF Object] = {
    IF IsFinalizationEnabled[handle] THEN RETURN; -- it was reissued
    Remove[handle]; -- remove it from the hash table, i.e., nullify the package refs
    };

  Find a cache entry for key; this is called by other (clients of this package) processes
Lookup: ENTRY PROC[key: Key]
    RETURNS [handle: Handle] = {

    FOR list: LIST OF Handle ← table[Hash[key]], list.rest UNTIL list = NIL
    DO IF Equal[list.first.key, key] THEN --found it--{handle ← list.first; EXIT};
    REPEAT
      FINISHED =>
        handle ← CreateAndInsert[key];
          Make a new one and enter it in the hash table
    ENDLOOP;

    now enable the object for insertion on myQueue when the collector finds that no client
    references to it exist. This also marks the object as (re)issued.
EnableFinalization[handle]; --no-op if IsFinalizationEnabled[handle]
  };
```

6. The Runtime Type System

Our experience with programming Cedar and its applications indicates that it is natural to use the REF ANY feature extensively, but only if the runtime type-equivalence predicate is fast.

To this end, we developed a finite-state-machine model for the Cedar type system (recursively defined types are common) and designed an algorithm based on finite-state machine minimization techniques to derive a canonical finite-state machine for a given Cedar type. We also developed an algorithm for converting the representation of such a "canonical" finite-state machine into a variable-length string of characters with the property that any two equivalent types would yield the same "type string." The Cedar type equivalence algorithm was thereby reduced to comparison of two variable-length strings for equality. Algol68 employs a similar method [12]. A further reduction in the cost of the algorithm can be achieved by computing a (fixed-length) check-sum from the type string and using numerical equality for type equivalence. A runtime data structure can then be used to map check-sums to the smaller typeTags. With appropriate choice of check-sum length and algorithm, this scheme can be adjusted to have an arbitrarily small likelihood of failure due to collisions.

The computation of type string hash codes is done by the compiler. The Cedar runtime software keeps the hash code with each type descriptor; use of a hash code equality test enables the loader to acquire typeTags rapidly when a new program module is loaded. TypeTags are indices in a table of pointers to type descriptors. TypeTags are stored in object headers. Two (one-word) typeTags are equal if and only if the corresponding Cedar types are equivalent. Thus, the runtime type-equivalence predicate is fast.

7. Conclusions

7.1 Dealing with Concurrency

Dealing correctly and efficiently with concurrency was the biggest challenge in the design of the allocator, collector and reference-counting machinery. Deadlock had to be avoided. Maximal concurrency was needed to minimize suspension of client activity and degradation of system performance. Atomicity requirements for operations on shared resources had to be specified correctly and guaranteed by the implementation. An ability to debug concurrent programs that exhibit unreproducible behavior was necessary.

One concurrency-related problem arises when changes are made to the contents of an active procedure frame while the collector is running. Consider the following scenario: at the start of a collection, an object X has a reference count of 2; one reference resides in an accessible object Y, and the other resides in an inaccessible object Z. No references to X reside on the stack. The collection begins; a client program picks up the reference to X that is in Y and then NIL-ifies it, decreasing X's count to 1 but leaving a reference to X in a procedure variable on the stack. The collector finds Z; the reclaimer decrements the count on X when it discovers the reference within Z; X now has a count of 0 and gets reclaimed. Disaster ensues sometime later when the client program tries to make sense of its reference to what used to be X. The solution to this problem is to provide a flag named "maybeOnStack" in each object's header; this is set while the collector is active whenever a client assignment decrements the reference count. Also, a pointer to the object is placed in the zero-count-table. Any increment of the reference-count clears the flag. The collector checks the flag before deciding that an object can be reclaimed. All maybeOnStack flags in objects referenced from the zero-count-table are cleared by the collector just before it finishes.

7.2 Performance

Our experience with Cedar indicates that the design outlined here is complete and that an efficient implementation on a microcoded machine can be produced with modest effort. Time spent by programmers to solve storage management related problems has decreased from an estimated 40% in Mesa to an estimated 5% in Cedar (the remaining problems deal almost exclusively with the management of large areas of virtual memory). Automatic storage deallocation can improve programmer productivity dramatically without affecting performance adversely.

Good performance was a major design criterion for the Cedar allocator, collector and reference-counting machinery. In the current Cedar system, reference-counted assignment takes about 2 microseconds on a Dorado [5, 14]. The average "round trip" time for allocation and collection of an object, including collector overhead, is roughly 150-200 microseconds.

Acknowledgments

Many people contributed to the ideas described herein and to the many tasks required to explore them. I am especially indebted to Russ Atkinson for proposing use of a zero-count-table; to Butler Lampson for many detailed design ideas; to Peter Deutsch and Danny Bobrow for many of the early ideas and (from Peter) much of the early code; to the two EPE study groups for motivation and focus; to Ed Satterthwaite for language design and compiler work; to Willie-Sue Orr and Howard Sturgis for microcode work; to Bob Hagmann and Russ Atkinson for design improvements based on their performance studies; to Jim Morris for perspective and good cheer; but mostly to Bob Taylor, who managed superbly both to anticipate the need for Cedar and to make it happen.

Responsibility for mistakes or oversights evident herein is, of course, my own.

References

[1] Andrew D. Birrell, and Bruce Jay Nelson. "Implementing Remote Procedure Calls," CSL-83-7, October, 1983 (also in *Transactions on Computer Systems*, **2** 1, February 1984).

[2] D. G. Bobrow, "Managing Reentrant Structures Using Reference Counts," ACM Toplas **2** 3 (July 1980).

[3] Mark Brown, Karen Kolling, and Ed Taft, "The Alpine File System"Xerox Palo Alto Research Center Report CSL-84-4. October 1984.

[4] R. G. G. Cattell, "Design and Implementation of a Relationship-Entity-Datum Data Model," Xerox Palo Alto Research Center Report CSL-83-4. May 1983.

[5] Douglas W. Clark, Butler W. Lampson, Kenneth A. Pier, "The Memory System of a High-Performance Personal Computer," Xerox Palo Alto Research Center Report CSL-81-1. January, 1981 (also in *IEEE Transactions on Computers*, **C-30** 10, pp. 715-733, October 1981).

[6] Jacques Cohen, "Garbage Collection of Linked Data Structures," Computing Surveys, (September 1981).

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

[8] L. Peter Deutsch and Edward A. Taft, "Requirements for an Experimental Programming Environment," Xerox Palo Alto Research Center Report CSL-80-10. June 1980.

[9] Adele Goldberg and Dave Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[10] J. H. Horning, editor, "Report from Second Programming Environment Working Group," Internal Memo, December 13, 1978.

[11] D. H. Ingalls, "The Smalltalk-76 Programming System: Design and Implementation," *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, 1978.

[12] J. Kral, "The Equivalence of Modes and the Equivalence of Finite Automata," ALGOL Bulletin 35, (March 1973).

[13] Butler W. Lampson, *The Cedar Language Reference Manual*, to appear.

[14] Butler W. Lampson and Kenneth A. Pier, "A Processor for a High-Performance Personal Computer," Xerox Palo Alto Research Center Report CSL-81-1. January 1981 (also in *Proceedings of Seventh Symposium on Computer Architecture*, SigArch/IEEE, La Baule, May 1980, pp 146-160).

[15] Robert M. Metcalfe and David R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, **19** 7, July 1976.

[16] James G. Mitchell, William Maybury, and Richard Sweet. "Mesa Language Manual, Version 5.0," Xerox Palo Alto Research Center Report CSL-79-3, April 1979.

[17] Eric Emerson Schmidt, "Controlling Large Software Development In a Distributed Environment," Xerox Palo Alto Research Center Report CSL-82-7, December 1982.

[18] G. I. Steele, "Multiprocessing Compactifying Garbage Collection," CACM **18** 9 (September 1975).

[19] Warren Teitelman, *The Interlisp Reference Manual*, revised 1978, Xerox Palo Alto Research Center.

[20] Warren Teitelman, "The Cedar Programming Environment: A Midterm Report and Examination," Xerox Palo Alto Research Center Report CSL-83-11, June 1984.

[21] John Warnock and Douglas K. Wyatt, "A Device Independent Graphics Imaging Model for Use with Raster Devices," *Computer Graphics*, **16** 3, July 1982.

XEROX

On Adding Garbage Collection and Runtime Types
to a Strongly Typed, Statically-Checked, Concurrent Language

Paul Rovner

CSL-84-7