63

Heap Usage in the Topaz Environment

John DeTreville

August 20, 1990

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Heap Usage in the Topaz Environment

John DeTreville

August 20, 1990

Abstract

Topaz, the experimental computing environment built and used at SRC, is implemented in the Modula-2+ programming language, which provides garbage collection. Garbage collection simplifies the construction of complex systems, and is tied to a number of other Topaz and Modula-2+ features, such as runtime polymorphism, language safety, information-hiding, object cleanup, persistent objects, and network objects.

Although there are costs to using garbage collection, these are avoided or tolerated in Topaz. For example, because Topaz must avoid noticeable interruption of service due to garbage collection, it uses a concurrent garbage collector.

Measurements show that the use of the `REF` heap in Topaz is similar in many ways to the use of heaps in Lisp-like environments, but different in others. For example, in typical large Topaz programs, the `REF` heap contains millions of bytes, with tens of thousands of objects from among hundreds of statically-declared types; objects of only a few types predominate. Although most objects are small, most bytes are in relatively large objects. Cycles are rare; most cycles are of size 2. Most objects are short-lived, but not as short-lived as in Lisp-like environments that allocate large amounts of ephemeral data on the heap.

1 Introduction

1.1 Background

Garbage collection originated in the Lisp environment [9] and most applications of garbage collection have been in Lisp or in Lisp-like environments such as Smalltalk [6]. Garbage collection can greatly simplify the task of producing correct programs by automating the process of storage reclamation. It thereby permits the use of more complex data structures than might otherwise be practical, or allows the construction of larger or more complex programs.

Conversely, garbage collection is usually not found in systems implementation languages, such as C [7] or Modula-2 [19]. (Notable exceptions are CLU [8], Cedar [17], and Oberon [20], as well as Lisp Machine Lisp [12].) However, garbage collection's advantages do not come without associated costs, which have tended to limit its use for systems programming.

- Garbage collection can cause momentary interruptions of service. With most collectors, the program is stopped while a collection is in progress.

  Imagine the design of a replicated file server. If the primary server fails, the secondary server should take over within one second. However, if a collection could stop the server for close to one second, this might be externally indistinguishable from a server crash.

- Garbage collection increases program overhead, increasing either running time (if the collector runs frequently) or storage requirements (if it does not).

- Primitive program operations are often more expensive in garbage-collected languages than in languages that do not use garbage collection. Garbage-collected languages can also lack the ability to perform very low-level operations (*e.g.*, bit-manipulation or pointer arithmetic) easily or efficiently.

1.2 Topaz

The Topaz environment at Digital Equipment Corporation's Systems Research Center (SRC) supports garbage collection, which is used extensively by Topaz programmers. Topaz is an experimental testbed for systems research at SRC and is constantly being extended as researchers implement their ideas. Topaz also serves as SRC's main computing environment. Work on Topaz began in 1984, and it has been used by about 50 researchers at SRC since 1986.

The principal language used in Topaz is Modula-2+ [14, 15], an extension of Modula-2; Modula-2+ was derived from Modula-2 by adding support for garbage collection, exceptions, and concurrency.[1]

Topaz addresses the drawbacks of garbage collection in the following ways.

[1] Some programming at SRC is in Modula-3 [2], a recent refinement of Modula-2+; Modula-

- Modula-2+ uses a concurrent garbage collector, which allows the program to run while a collection is in progress. Since Topaz provides multiple threads of control within each address space, providing a concurrent collector is relatively simple.
- The increased overhead is typically ignored. Since Topaz is an experimental system running on powerful workstations and servers, this is not unreasonable. Moreover, as computers grow generally more powerful, it becomes more attractive to trade off ultimate performance against greater functionality and ease of development.
- Garbage collection is an addition to Modula-2+, but all the Modula-2 low-level features have been retained in the process.

To support garbage collection, Modula-2+ adds the `REF` type constructor to Modula-2. For any type T, a value of type `REF` T is either `NIL` or is a reference to an object of type T on the heap. If `x` is a variable of a `REF` type, the statement `NEW(x)` allocates a new heap object, and makes the variable `x` refer to it (*i.e.*, it assigns a `REF` to the new object to `x`). The `REF` heap is garbage-collected.

In Topaz, each program runs in its own address space. Therefore, each program has its own `REF` heap and its own instance of the garbage collector. Taos, the Topaz operating system, also resides in its own address space, and also uses garbage collection. Only the lowest level of the system (the Nub, which implements address spaces, threads, etc.) does not use garbage collection.

1.3 Outline

Section 2 of this paper discusses the reasons for using `REF`s in the Topaz environment. Some motivations, such as data sharing, are typical of garbage-collected systems; others, like the facilities for network `REF`s, are unique to Topaz.

Section 3 discusses the disadvantages of using `REF`s. Currently, the principal disadvantage is poor performance.

Section 4 outlines the current implementation of the `REF` heap in Modula-2+.

Section 5 presents measurements of heap usage in Topaz. A pair of typical large programs were instrumented and a great variety of measurements were taken. The measurements included static measurements of program structure, measurements based on snapshots of the programs' heaps, and measurements based on logs of program activity.

Section 6 lists the minor non-`REF` heaps also present in Topaz.

---

3 also provides garbage collection. Some additional programming is in other garbage-collected languages like CLU, or non-garbage-collected languages like C. Since none of these other languages are used as extensively for programming in Topaz as Modula-2+, they are not discussed here.

```
SAFE INTERFACE MODULE M;

TYPE
  T = REF some type;

PROCEDURE P(x: T);

END M.
```

Figure 1: Example of Modula-2+ interface

2 Qualitative heap usage

This section outlines the reasons for using `REF`s, including some of the facilities unique to `REF` values in Modula-2+ and Topaz.

2.1 Sharing

Like the Modula-2 `POINTER` values from which they were derived[2] (and which are retained in Modula-2+ for low-level programming), `REF` values allow data structure sharing. Since `REF` objects are garbage-collected, this allows complex data structures or complex uses of data structures to be programmed far more easily than if objects had to be explicitly freed. This complexity can be in the data structures themselves (where it may not be clear which parts are no longer in use) or in the programs using the structures (where there may be no central knowledge of which parts of the data structures may be used again).

To illustrate the second point, consider Figure 1, which shows an interface `M` that exports a procedure `P` that takes a parameter `x` of `REF` type `M.T`. (Modula-2+ separates interfaces from their implementations, to help in constructing large systems.) Imagine that Modula-2+ abandoned garbage collection. To remain equally useful, the definition of procedure `P` in this interface would also have to state the conditions under which the caller could free the object after a call to `P`, or the conditions under which `P` could retain the object, freeing it later, or the conditions under which the two would cooperate to free the object.

With garbage-collection, however, no such annotation is necessary. This simplifies the construction and use of interfaces.

Additionally, a program that frees objects explicitly might fail to free objects when they are no longer accessible (which is particularly a problem for long-running programs, like servers), or might accidentally free objects when there are still `REF`s outstanding (which is a problem for any program). Garbage collection

[2] Modula-2 `POINTER` values are like pointers in most modern Algol-like languages. A `POINTER TO` *T* is a pointer to an object of type T, which can be created by the `POINTER` version of `NEW`—and explicitly freed by `DISPOSE`—or constructed by address arithmetic.

```
TYPE
  T = REF Pair;
  Pair =
    RECORD
      first: REFANY;
      tail: T;
     END;
```

Figure 2: Definition of `List.T`

supports Modula-2+ in its purpose of building large, robust systems with no unchecked runtime errors.

2.2 Polymorphism

In addition to `REF` types, Modula-2+ also provides the `REFANY` type, which is the superset of all `REF` types. A `REFANY` variable can hold any `REF` value. Since all `REF` objects on the heap contain a typecode, naming their `REF` type, a `REFANY` value can be disambiguated at runtime.[3]

For example, the standard `List` interface defines a `List.T` as shown in Figure 2. The elements of a `List.T` are `REFANY` values. Clients of the interface can build lists of any `REF` type or mixture of types, disambiguating them at runtime.

One use of `List.T`s that uses a mixture of element types is in the `Sx` ("S-expression", as in Lisp) package. The `Sx` package exports procedures to read and write S-expressions, so that the external syntax

```
("Hello" 'a' 0 #False (x))
```

would correspond to a list of five elements: a `Text.T` (which is like a character string); a `Ref.Char` (which is a `REF CHAR`); a `Ref.Integer` (a `REF INTEGER`); a `Ref.Boolean` (a `REF BOOLEAN`); and a `List.T` containing a single `SxSymbol.T`, which represents a Lisp-style symbol.

Another use of `REFANY` is to pass arguments to procedure variables. When clients of an interface provide procedure arguments for later callback, it is convenient to provide an argument also to be passed at that time to provide some context for the operation. (This combination of procedure and context argument is sometimes called a "closure," although of course it is not as general as full closures.) If this argument is defined as a `REFANY`, the client procedure can `NARROW` it to the expected `REF` type before use.

[3] A `NIL REFANY` value has no underlying type; its "typecode" is a distinguished value.

2.3 Open arrays

Modula-2+ allows open arrays (arrays of an indeterminate number of elements) in only a few contexts: as the type of formal arguments to a procedure, and as the referent type of a `REF` type constructor. (This second use is an extension over Modula-2.) For example, a variable may have the type `REF ARRAY OF CHAR`, although only a formal argument may have the type `ARRAY OF CHAR`.

When `NEW` is applied to a variable of type `REF`-to-open-array, the number of elements must be specified; the number of elements may be any non-negative integer. At runtime, it is possible to determine the number of elements of such an open array value.

2.4 Safety

Most Modula-2+ modules (*i.e.*, interfaces and implementations) are "safe," explicitly noted by the keyword `SAFE`. In a safe module, no program errors can cause the language abstractions to be violated. For example, array accesses are checked for bounds violations.

Many `POINTER` operations must be disallowed in safe modules. For example, it is common to create a `POINTER` using address arithmetic, but such `POINTER`s cannot in general be validated.

`REF` operations, on the other hand, are all safe; new `REF`s can be created only using `NEW`. Although `REF`s can be forged using `POINTER` operations (*e.g.*, dereferencing a bogus `POINTER TO REFANY`), or using `LOOPHOLE` (which treats a value's bit-pattern as though it were of another type), such operations are disallowed in safe modules.

2.5 Opaque `REF`s

Modula-2+ allows a type to be opaque in an interface, and concrete in an implementation module. For example, Figure 3 shows an interface with an opaque `REF` type `M.T`. The only operations that clients can perform on objects of type `M.T` are `M.Create` and `M.Manipulate`, plus assignment and comparison for equality.

As illustrated in Figure 4, the concrete definition can appear in the corresponding implementation module, and the concrete definition can be used there. This allows the implementation procedures to manipulate the concrete representation of the type.

Alternatively, the opaque definition can be given in a public interface and its concrete definition can be given in a second, private interface. The public interface is available to ordinary clients of the type, while the private interface can be used by multiple implementation modules that are allowed access to the concrete representation. For example, abstract operations on a type could be defined in multiple interface modules, dividing the operations on the type among

```
SAFE INTERFACE MODULE M;

TYPE
  T = REF;

PROCEDURE Create(): T;
PROCEDURE Manipulate(x: T);

END M;
```

Figure 3: Example of opaque `REF`

a set of interfaces; or a type could be implemented in one implementation module but its concrete representation could be available in other implementation modules, for efficiency or convenience.

In any case, there can be only one textual occurrence of a type's concrete definition in a program, as Modula-2+ uses name equivalence, not structural equivalence.

The binding of concrete types to opaque types is conceptually done at link time.[4] Because the bindings between opaque types and concrete types is not known at compile time, types that are considered different at compile time may become identical at link time (*i.e.*, they may be bound to the same concrete type).

This feature causes some problems. For example, in Modula-2+'s `TYPECASE` statement, which is like a `CASE` statement based on the type of a `REFANY` argument, two arms may be labeled with types that seem different at compile time (and that might seem different to the programmer) but that become the same at link time; the effect of executing the `TYPECASE` can depend on which arm occurs first, which might be unexpected.

2.6 Object cleanup

When an object is freed, object cleanup allows cleanup activities to be performed. For example, imagine a `REF`-based abstraction for an open file. If all references to the open file are dropped, the file should be closed. Object cleanup allows a type-specific cleanup routine to take such an action before the object is freed.

As a more complicated example, imagine that open files are also stored in a hash table. Just as the previous example involved applying a type-specific operation when zero `REF`s remained to the object, object cleanup can also be

[4]Because of linker limitations, however, the current Modula-2+ system does the binding at program startup.

```
SAFE IMPLEMENTATION MODULE M;

TYPE
  T =
    REF
      RECORD
        record elements
      END;

PROCEDURE Create(): T;
  BEGIN
    implementation of Create
      using the concrete definition
  END Create;

...

END M;
```

Figure 4: Example of opaque `REF` implementation

invoked when only one `REF` (the one in the hash table) remains.

Object cleanup is established for a type by the `ObjectCleanUp.EstablishCleanUp` routine. Its parameters are the `REF` type (given as the typecode, such as `TYPECODE(T)`); the threshold reference count at which cleanup should occur; and a queue into which `REF`s to the object will be placed.

Object cleanup is enabled for a particular object by `ObjectCleanUp.EnableCleanUp`. When cleanup is enabled for an object and its reference count drops to the per-type threshold, cleanup is disabled and a `REF` for the object is placed on the type's queue.

Usually, a concurrent thread reads `REF`s from the queue, and performs any necessary finalization operations. If desired, cleanup can be enabled on an object over and over.

2.7 Pickles

A pickle is an external copy of a heap structure. Heap structures can be "pickled" into a byte-stream, and the byte-stream can be "unpickled" back into a copy of the original structure. Pickles are fast to write, and very fast to read.

Pickles can therefore serve as a standard structure for long-term data storage. For example, window system fonts are stored as pickles. A font is represented as a `REF` to the font information, and the pickles are stored in files. On startup, programs may read these pickles to define fonts.

Many programs write out their internal state as a pickle, then read it back in later. Pickles can also serve to communicate structured information across a communication channel connecting two programs.

Some restrictions apply in the use of pickles. They are not human-readable or human-editable, although special editing programs can be written. Although `REF`s in the data structures are followed and reconstructed correctly during pickling and unpickling, `POINTER`s are not; the result of unpickling a pickled pointer is the original bit-pattern, which may not be useful. Similarly, procedure values cannot usefully be pickled, and data structures based on the addresses of `REF` objects (*e.g.*, a hash table based on `REF` bit-patterns) will not work.

To circumvent these problems in some cases, special type-specific procedures can be defined for pickling and unpickling. For example, a hash table might be pickled by listing the elements, and unpickled by building a new hash table.[5] An `SxSymbol.T` value, which represents a Lisp-style symbol, can be pickled by simply outputting its name; this avoids pickling secondary information associated with the symbol. Upon re-pickling, the `REF` to the "same" symbol can be reconstructed.

A final restriction is that a value can be unpickled only if the `REF` types it contains also exist in the program doing the unpickling. Types are represented in pickles as fingerprints of the definitions;[6] these are translated into typecodes during unpickling. Therefore, if a programmer changes the definition of a type, old pickles using the old definitions can become unreadable. Fortunately, translation programs are simple to write.

2.8 Network `REF`s

Network `REF`s are opaque `REF`s that stand for objects in other address spaces. (Although they are called "network" `REF`s, they may also stand for objects in other address spaces on the same machine, as well as on machines across a network.)

As an example of the uses of network `REF`s, consider an open file (*i.e.*, an `OS.File`). Inside Taos, the operating system, the object representing an open file has significant internal structure; it holds state for its clients, and it points off to lower-level abstractions. There are procedures that return files (*e.g.*, `OS.Open`), and other procedures that operate on files (*e.g.*, `OS.Read`).

Taos runs in an address space separate from its client address spaces. Even so, its clients can access Taos procedures via Remote Procedure Call (RPC) [1]. Automatically-generated or programmer-customized stub procedures on each side of the address space boundary provide the illusion that the services and the clients are in the same address space.

---

[5] In addition to solving the previous problem with `REF` bit-patterns, this can also save space.

[6] In Topaz, a fingerprint is a 64-bit quantity derived from a string. Two strings can be expected to have about a 2^{-64} chance of being mapped to the same fingerprint, even if they have similar structure.

`REF`s normally do not have meaning across address spaces. For example, when RPC passes a `REF` as an argument or a result, it normally passes a copy of the structure referenced. This obviously will not work for the `OS.File` example. Remote operations on open files must therefore use network `REF`s instead.

To provide network `REF`s, the RPC machinery inside Taos automatically maintains a table of `REF`s in Taos that are referenced from outside; in client address spaces, an `OS.File` is defined simply as an index into that table.

When the result of an `OS.Open` is to be passed back to the client address space, the stubs register the `REF` in the table at a new index, and return the index for the calling address space to construct the network `REF`. When this network `REF` is used for a future operation, the operation is reversed: the index is passed to Taos and the real `REF` is looked up. These transformations are performed automatically by the network `REF` machinery in RPC.

Although this discussion was in terms of operating system objects and services, it generalizes to any client-server relationship, across address spaces on the machines or across machines. It obviously requires that the network `REF` type be opaque in the clients.

The network `REF`s in client spaces have object cleanup enabled. When the client drops all references to a network `REF`, the server is notified, and the table entry is deleted; this may cause the object to be deleted or cleaned up on the server side. The server is also notified when a client address space terminates, and deletes that space's entries.[7]

3 Disadvantages

In addition to the advantages of using `REF`s, there are some disadvantages.

- The biggest disadvantage is performance. In the current system, allocating an object currently executes about 30 VAX instructions in the most common case; this includes assignment to a local variable. Assigning a `REF` value to a local variable requires 1 VAX instruction, but assigning to a `REF` variable that might be shared (a global, or a `REF` element of a heap object) takes about 18 VAX instructions.

  Moreover, two of the instructions executed for allocation or non-local assignment are interlocked instructions, to acquire and release a lock. (Currently, no programmer-supplied synchronization is required when sharing `REF`s between mutator threads, or between the mutator and the collector. As a result, some synchronization must be provided automatically for `REF` assignment.) Interlocked instructions are especially slow to execute.

  Assignment to potentially shared `REF`s is performed in a library routine, so the cost of procedure call and return must also be added.

[7] Since a network `REF` is considered to belong to a single client space, this limits the extent to which network `REF`s may be passed around freely among clients.

As a result, measurements show that global `REF` assignment takes an average of 13.5 μs on SRC's current workstations, while integer assignment takes only 0.5 μs. Allocating a one-word object takes an average of 45 μs.

(The instruction counts could be reduced by 5-10 instructions if the code sequences were inlined and special-cased. Further, some locking could be eliminated by the use of per-thread state. Some such improvement is planned for the future.)

- Another disadvantage is that the collector can be a bottleneck. Although the collector runs concurrently, which usually does not slow down normal program activities on a multiprocessor, the collector can fall behind a rapidly-allocating program (especially if multiple threads are allocating). At some point, the collector suspends the program until it can catch up.

  Also, if a program is run on a uniprocessor, the collector competes directly with the program for processor cycles. The current design optimizes the speed of allocation at the cost of extra time in collection.

- The final performance disadvantage is that the space required using collection can be greater than that required using explicit deallocation, since the collector lags behind the program. The heap always contains some objects that, although inaccessible, have not yet been collected; the space for these objects counts as overhead.

There are also a few problems of functionality. For example, the collector will never free an object referenced from a thread state (*i.e.*, from its registers and its stack frames), but, in the current system, there is no information as to which words in the thread state are `REF`s and which are not. Therefore, some objects might never be freed because of non-`REF`s in thread states that have the same bit-pattern as an old `REF`.[8] This can cause a small space problem, but larger problems when object cleanup performs a complex operation. For instance, there is a slight chance that if the last `REF` to an open file is dropped, the cleanup action (closing the file) will not be performed.

Also, language safety reduces functionality, as it places some restrictions on the programmer. For instance, there are several restrictions in the use of `REF` fields in variant records, to keep the programmer from changing the variant record's tag and treating a formerly non-`REF` bit-pattern as if it were a `REF`. Finally, although `REF` use in safe programs can never crash the runtime system, this is not the case with the additional operations available in unsafe programs; this can make it hard to cheat the system to gain extra speed (which can be viewed as an advantage or a disadvantage).

[8] A similar problem can happen when a `REF` variable is dead, but still appears in the thread state, and no extra information is available to the collector.

4 Heap implementation

4.1 Type representation

Types are represented at runtime by numeric typecodes. Typecodes are unique only with a particular program; they are small integers assigned during program initialization. A typecode acts as an index into a table of runtime type definitions. The definition includes a list of which object fields contain `REF` values, so that facilities like the collector and the pickler can follow references. For speed, a small interpreted language is used to walk over the objects' `REF` fields.

4.2 Object representation

Each heap object is an integral number of 4-byte words long, due to memory atomicity constraints. Topaz runs on shared-memory multiprocessors; on the machines it may soon run on, the memory coherence unit is no more than 4 bytes. If no two objects share the same word, concurrent accesses to these objects from different threads on different processors will not interfere.

Each heap object has a header of at least one word. This word contains a 2-byte typecode and 2 bytes of allocator and collector state. The typecode is assigned when the object is allocated, and although the collector runs concurrently and changes the state asynchronously, this does not affect concurrent program fetches of the typecode.

Large objects (defined in section 4.3) have an extra word of header.

If the referent type of a `REF` object is an open array, an additional one-word open array count is added to the header.

Each object, then, has a possible large-object header word, followed by the normal header word, followed by a possible open array count word. A `REF` to the object is represented as the address following the normal header; this puts the typecode at a constant offset from the `REF`. The value of `NIL` is zero.

4.3 Memory layout

Memory in the heap is divided into 8192-byte "pages," which is an integral multiple of the VM page sizes of all machines that Topaz may soon run on. New pages are allocated from VM as necessary, and allocated using a binary-buddy system.

Heap objects are either "small" or "large". Small objects occupy up to 4096 bytes, including header; large objects occupy over 4096 bytes.

There are 40 possible sizes of small object.[9] Intermediate sizes are rounded

[9] The possible sizes are 8, 12, 16, 20, 24, 28, 32, 40, 48, 56, 64, 72, 80, 96, 112, 128, 144, 160, 192, 224, 256, 320, 384, 448, 512, 576, 640, 768, 896, 1024, 1152, 1280, 1536, 1792, 2048, 2304, 2560, 3072, 3584, and 4096 bytes, including headers. These sizes were chosen to reduce proportional breakage; the spacing is finer for small sizes than for large sizes. It is perhaps unfortunate that this list includes powers of 2, since powers of 2 are popular sizes for objects,

up; the proportional breakage is less than 20% for small object sizes over 12 bytes. Small objects are packed into pages; for simplicity, all small objects on a page are the same size. Small object headers contain a 5-bit "block size index" that defines their size.

Large objects occupy their own run of pages. Their size in pages is a power of 2, and the breakage is not used for other objects. The block size index in their headers holds a distinguished "large" value; the extra word of header holds the actual size. Breakage can be up to 50% for large objects.

Objects created by reading a pickle have a special memory layout, corresponding to the layout in the pickle's byte stream. The objects are laid out contiguously, regardless of their sizes. Small pickles are packed into small pickle pages; large pickles occupy their own run of pages.

`Text.T` (character string) constants are stored in the program's read-only text section. Their layout is the same as if they were on the heap.

4.4 Collector operation

The collector runs concurrently, triggered by program activity. The collector combines reference-counting (following [5] and [13]) and mark-and-sweep collection; the collector is described in detail in [4]. Objects that are not referenced are reclaimed and returned to free lists.

There is one free list per size of small object. Free entries for the same page are kept adjacent, to concentrate allocations on some pages while allowing others to be freed. When all objects on a small page are free, the page is returned to the page allocator.

When a large object is freed, its pages are returned to the page allocator.

When individual objects in a pickle are freed, they are not returned to a free list. When all the pickle objects on a whole page or run of pages are free, the page or pages are returned to the page allocator. The goal of this policy is to reclaim pickle space as quickly as possible when an entire pickle is freed at once.

`Text.T` constants are not collected.

5 Quantitative heap usage

This section outlines the results of measurements on two programs at SRC, performed to measure the heap usage of real Modula-2+ programs in Topaz. The programs measured were *Taos*, the Topaz operating system, and the *Ivy* text editor. Each program has been developed and used for some while. Each has multiple authors, and each includes significant amounts of library code written by others. Therefore, each program can be expected to represent a microcosm of SRC software; their measurements can be expected to be representative of Topaz as a whole.

---

not including header.

Referent type	REF types	Percentage
Records	290	67.3%
Open array	104	24.1%
Scalar[a]	23	5.3%
Array	9	2.1%
Set	4	0.1%
REF	1	0.0%

[a] "Scalar" includes BOOLEAN, INTEGER, CARDINAL, UNSIGNED, REAL, LONGREAL, enumerations, and subranges.

Table 1: Distribution of referent types of REF types in Taos

5.1 Taos

The first Topaz program measured was Taos, the Topaz operating system [10, 11]. Taos packages a large number of facilities (*e.g.*, process management, the file system, the window system, networking protocols, and a large number of libraries) that were written by a number of different people over a long period of time. Taos includes an emulation of the Unix[10] kernel.

5.1.1 Static measurements

Taos version 88.1, the current release at the time of the measurements, had 431 different REF types.[11]

Table 1 breaks down the referent types of the 431 REF types. Most referent types are records; almost all are records or open arrays.

Table 2 breaks down the sizes of the referent types of the 327 REF types whose referent types were not open arrays. The first column shows the object size on the heap, including header and breakage; the second column shows the number of REF types having this size; the third column lists the sizes of the referent types ignoring overhead. The median size of the referent types is 20 bytes, corresponding to a 24-byte object including header.

Similarly, Table 3 lists the sizes of elements of the 104 open array types that were the referents of REF types. The median element size is 8 bytes, as for a linked list of one-word values.

Object cleanup was established on 14 REF types. The threshold reference count was 0 for two types, and 1 for the other twelve.

There were 375 global variables that contained REF values (*e.g.*, a global REF, or a global record containing REFs). In all, these global variables contained

[10] Unix is a registered trademark of AT&T Technologies.

[11] Modula-2+ uses name equivalence, not structural equivalence, to define type equivalence, and some of these 431 REF types were structurally equivalent; there were only 367 structurally different referent types.

Object bytes	REF types	Referent type bytes	Object bytes	REF types	Referent type bytes
8	41	3 bits, 1 (3), 1.5, 4 (36)[a]	512	1	500
12	40	5 (2), 8 (38)	576	2	512, 568
16	39	9, 9.5, 12 (37)	640	2	576, 612
20	31	13 (2), 13.5, 14 (3), 16 (25)	768	0	
24	25	17 (2), 18 (2), 20 (21)	896	2	812 (2)
28	17	22 (2), 23, 24 (14)	1,024	0	
32	9	25 (2), 26, 28 (6)	1,152	4	1036, 1048, 1108, 1144
40	18	29 (2), 32 (11), 36 (5)	1,280	1	1244
48	12	40 (6), 42, 44 (5)	1,536	1	1512
56	10	48 (6), 50, 52 (3)	1,792	1	1600
64	15	56 (6), 57, 58 (2), 60 (6)	2,048	2	1796, 1812
72	8	64 (3), 66 (2), 68 (3)	2,304	0	
80	3	69, 73, 76	2,560	1	2461
96	6	80, 81, 84 (2), 88, 92	3,072	1	2816
112	3	96, 104, 108	3,584	0	
128	0		4,096	3	4084 (3)
144	1	136			
160	4	144 (2), 156 (2)	8,192	8	4108, 4112, 4116 (2), 4128,
192	2	164, 184			4368, 4648, 5128
224	2	196, 212	16,384	1	8236
256	1	228	32,768	2	16392, 18040
320	4	256	65,536	0	
384	1	370, 281, 296, 300	131,072	0	
448	1	404	262,144	1	158928

[a]That is, one REF type with a 3-bit referent, three with 1 byte, one with 1.5 bytes, and 36 with 4 bytes.

Table 2: Sizes of referent types of REF types in Taos

Elt. bytes	# of REF types	Elt. bytes	# of REF types	Elt. bytes	# of REF types	Elt. bytes	# of REF types
1 bit	1	3	2	9	1	20	2
3 bits	1	4	30	12	10	21	1
1	24	6	1	13	1	24	1
2	2	8	20	16	4	32	2

Table 3: Element sizes of open array REF referent types in Taos

REF slots	# of globals	REF slots	# of globals	REF slots	# of globals	REF slots	# of globals
1	298	11	1	45	1	256	10
2	20	12	1	52	1	257	3
3	2	13	1	64	2	768	1
4	3	18	2	100	1	1024	1
5	6	20	2	101	2	1025	1
6	1	23	1	127	1	1029	1
7	2	31	1	128	2	2043	1
8	1	32	1	201	1	2048	1
10	1						

Table 4: Number of `REF` slots in `REF`-containing global variables in Taos

23,002 `REF`s, or an average of 61 `REF`s per global. However, the distribution is quite skewed, as shown in Table 4; there are a few large structures that contain a great number of `REF` slots.

5.1.2 Heap count measurements

For the following measurements, a total of 40 instances of Taos running at SRC were examined; each instance was running on a personal workstation.[12] The statistics were averaged over the 40 cases.

There was a mean of 8,983 objects on the Taos heap. On the 40 Taoses examined (all of which were in a quiescent state), only 264 types had objects allocated at that time. Almost half the objects were of just 5 types, as listed in Table 5 along with the distribution of number of objects by deciles.

There was an average of 1,081,969 bytes of objects on the heap (including headers and breakage.) Almost half the bytes were in objects of just 6 types, listed in Table 6 with the distribution.

The mean object size is 120 bytes, including headers and breakage. Figure 5 shows the distribution of object sizes in Taos, by number of objects. The median object size is 20 bytes, including headers and breakage.

Figure 6 shows the same distribution, but by number of bytes instead of by number of objects. The median of this graph is 576 bytes; about half of the bytes are allocated to objects of less than 576 bytes, and half of more.

[12]The instances of Taos chosen to be examined were those personal workstations that had been used since they were booted, but that had not been used in the last hour, since the examination was intrusive. The examination was performed using Loupe, the Modula-2+ teledebugger. A Loupe running remotely was attached to each Taos, and a full garbage collection was initiated before measurements were taken. The actual measurements were performed using debugger-callable routines built into the Modula-2+ runtime that report on heap usage; these routines are usually used by programmers to diagnose heaps that grow too large. Finally, each workstation was released.

Objects	Type
2050	`Ref.Integer`[a]
1344	`Text.T`[b]
402	`Thread.T`[c]
357	`TEmDpy.RowRef`[d]
329	`ActiveFile.T`[e]

Number of Types	Cum. Count
1	22.8%
2	37.8%
3	42.3%
5	49.9%
9	60.9%
14	70.5%
25	79.9%
55	90.0%
264	100.0%

[a] A `Ref.Integer` is a `REF INTEGER`. Taos contains a static table of 2048 `Ref.Integers`; each Taos examined contained 2 more.

[b] A `Text.T` is an immutable text string. A `Text.T` may share internal structure with other `Text.Ts`. Many operations in the Modula-2+ libraries operate on `Text.Ts`.

[c] A `Thread.T` is a handle for a thread of control in this address space.

[d] A `TEmDpy.RowRef` represents a row in a terminal emulator window.

[e] A `ActiveFile.T` represents an open file. (`ActiveFile.Ts` can be cached after they are closed.)

Table 5: Top 5 `REF` types in Taos, by number of objects, and distribution

Objects	Bytes	Type
357	114,248	`TEmDpy.RowRef`
329	105,368	`ActiveFile.T`
176	101,376	`Dir.BufferRef`[a]
118	75,680	`LocalFile.File`[b]
402	64,332	`Thread.T`
7	60,211	`REF ActivePipe.Buffer`[c]

Number of Types	Cum. Bytes
1	10.6%
2	20.3%
3	29.7%
5	42.6%
6	48.2%
9	60.7%
12	70.2%
18	80.4%
34	89.9%
264	100.0%

[a] A `Dir.BufferRef` is a `REF` to a buffer holding a directory block.

[b] A `LocalFile.File` represents an open file on the local file system. (`LocalFile.Files` can be cached after they are closed.)

[c] A `REF ActivePipe.Buffer` is a `REF` to a buffer for a pipe.

Table 6: Top 6 `REF` types in Taos, by number of bytes, and distribution

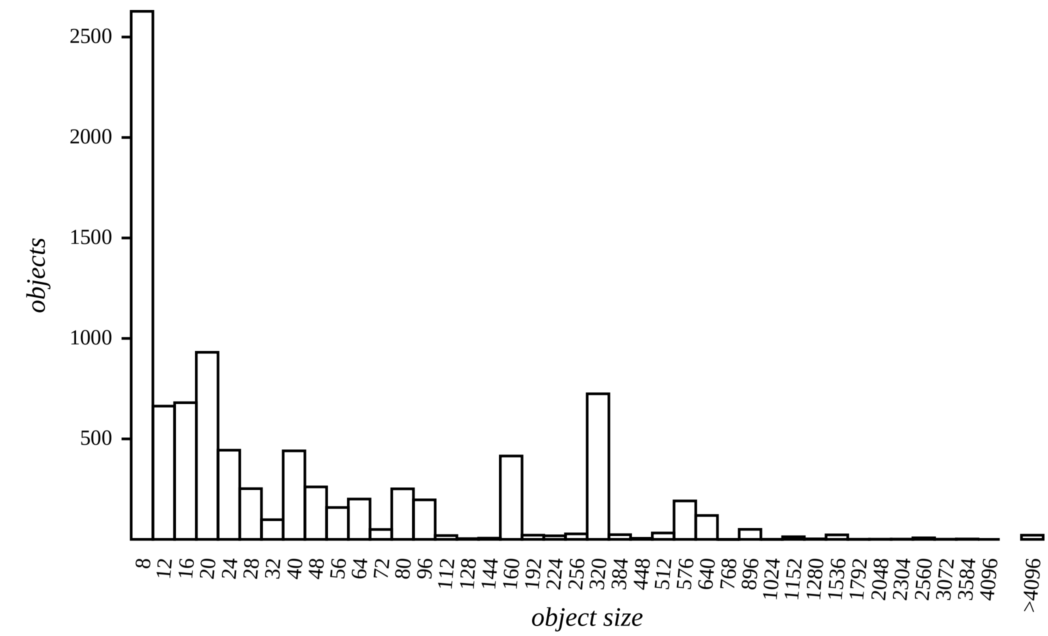

Figure 5: Distribution of objects in Taos, by object size

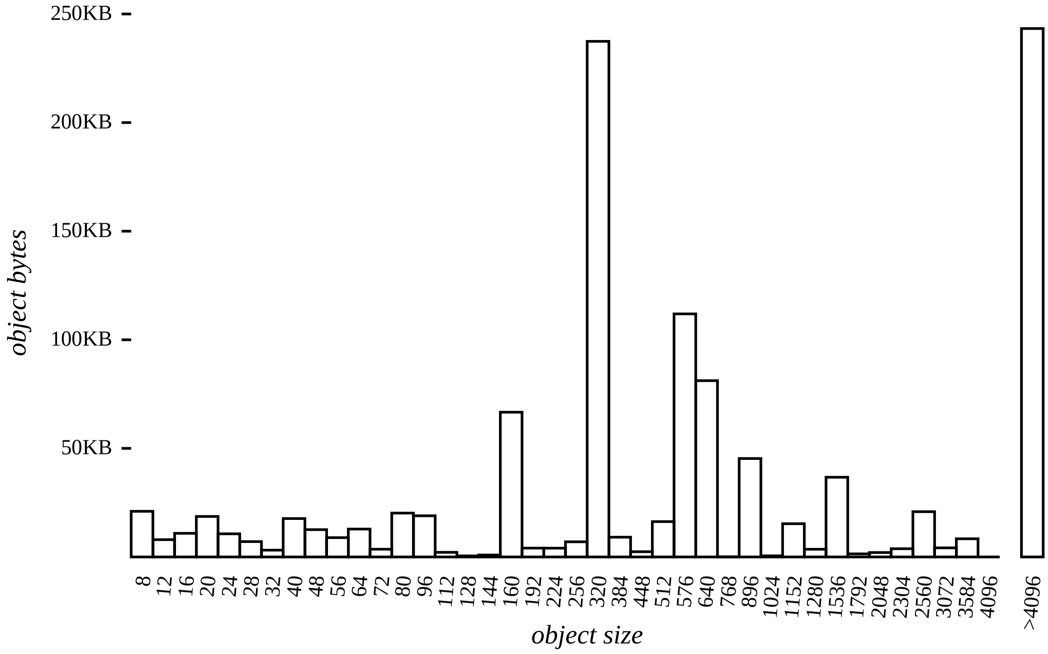

Figure 6: Distribution of object bytes in Taos, by object size

REF slots	Count	REF slots	Count	REF slots	Count	REF slots	Count
0	7207.4	10	344.8	32	1.0	103	2.0
1	434.6	12	2.8	34	466.2	129	39.0
2	634.8	13	0.6	40	1.0	130	10.8
3	967.6	16	2.8	48	0.8	199	1.0
4	248.6	17	4.8	62	10.8	200	1.0
5	236.6	18	11.2	64	2.0	201	1.0
6	23.8	20	58.4	67	1.0	257	4.6
7	56.0	22	8.6	80	0.8	258	2.0
8	63.2	31	0.8	101	1.0	512	0.2
9	171.4						

Table 7: Distribution of number of `REF` slots in Taos objects

The breakage on the Taos heap was not measured, but can be inferred to be between 0 and 247,223 bytes. The actual value can be expected to be about half the maximum, or 123,611 bytes. This is 13.8 bytes per object, or 11%.

The mean number of objects enabled for cleanup is 182.

5.1.3 Heap shape measurements

The following measurements are based on 5 snapshot instances of Taos, fewer then before; these measurements required modification of Taos to collect statistics.[13] In these measurements, there were 11,025 objects per Taos heap. (Unless otherwise stated, results given are means.)

Of the 23,002 `REF` slots in global variables, only 4,303 (18.7%) were non-`NIL`; these slots held 4,179 different `REF` values. Most of the `NIL` global `REF` slots were in large tables.

The mean number of `REF` slots in heap objects was 3.7. The mean number of `REF` slots in objects with at least one `REF` slot was 10.6. Table 7 shows the distribution of the number of `REF` slots.

The mean in-degree of objects in the heap is 1.3; this is the number of `REF` values in globals or in objects that reference the object.[14] The mean in-degree of objects with non-zero in-degree is 1.44. The mode of the in-degrees (*i.e.*, the most common in-degree) is 1. Table 8 shows the distribution of in-degrees.

The mean out-degree of objects was 0.9; this is the number of non-`NIL` `REF`s in the object. (The mean out-degree is greater than the mean in-degree because

[13] The mark-and-sweep collector was modified to log all objects and `REF`s found during the sweep phase, and to store the log in memory; each collection overwrote the results of the previous. After an instance of Taos had been put through a representative workload, a Loupe was attached to the Taos, and the contents of the array were dumped.

[14] References from `REF` variables that are local to procedure activations are not considered here; doing so would be difficult with the current Modula-2+ system. Considering these references would increase the in-degrees.

In-degree	Count	In-degree	Count	In-degree	Count	In-degree	Count
0[a]	1226.0	14	5.0	29	0.2	91	0.2
1	8053.6	15	2.6	30	1.0	92	0.2
2	773.8	16	0.2	32	0.8	108	0.2
3	727.0	17	0.8	34	0.6	146	0.2
4	110.8	18	0.6	35	0.2	156	0.2
5	39.2	19	0.8	39	0.2	174	0.2
6	16.8	20	0.4	41	0.2	236	0.2
7	14.6	21	1.6	47	0.2	335	0.2
8	10.8	22	1.8	50	0.2	370	0.2
9	9.4	23	0.2	59	1.0	379	0.2
10	7.2	24	0.4	70	0.6	393	0.2
11	3.6	25	0.8	84	0.2		
12	4.6	26	0.2	87	0.2		
13	3.2	27	0.4	89	0.4		

[a]The objects listed with 0 in-degree are those that were nevertheless accessible from outside the heap. Thread states referenced an average of 443 additional heap objects—according to a conservative scan—and the rest became inaccessible during statistics-gathering.

Table 8: Distribution of in-degrees of Taos objects

the in-degree includes global `REF`s.) The mean out-degree for nodes containing at least one non-`NIL` `REF` is 2.7. Table 9 shows the distribution of out-degrees.

Most objects on the heap are a short distance from a root.[15] Table 10 shows the frequencies of the minimum distances from a root. There are relatively few long chains; most objects on the heap are accessible in a few `REF`s from a root.

There are relatively few cycles on the Taos heap. (Until 1989, the Modula-2+ collector used reference-counting alone, and did not collect cyclic structures. Therefore, programmer action was necessary either to avoid cycles, or, more typ-

[15]Here, a root includes not only global `REF`s and `REF`s in thread states, but also some of the `REF`s referenced during the data collection.

Out-degree	Count	Out-degree	Count	Out-degree	Count	Out-degree	Count
0	7413.0	9	7.4	25	0.2	40	0.2
1	1198.6	10	6.2	26	0.2	44	1.0
2	1231.8	11	1.4	27	0.2	47	0.2
3	449.2	13	1.0	29	1.4	48	1.0
4	233.4	14	1.0	31	0.2	49	0.2
5	271.4	15	1.0	33	0.2	50	0.6
6	114.6	16	1.4	36	0.4	58	10.8
7	59.2	17	0.8	37	1.0	70	0.2
8	13.6	20	1.2	39	0.6	476	0.2

Table 9: Distribution of out-degrees of Taos objects

Depth	Count	Depth	Count	Depth	Count	Depth	Count
0	5690.6	12	32.2	32	5.6	63	2.4
1	1821.4	13	21.8	33	5.0	64–66	2.0
2	1186.6	14	17.2	34	4.2	67–69	1.8
3	1013.0	15	13.0	35–41	4.0	70–71	1.6
4	351.2	16	10.0	42	3.8	72	1.4
5	194.2	17	7.8	43	4.0	73–74	1.2
6	99.0	18–21	7.6	44–53	3.8	75–80	1.0
7	66.6	22	7.4	54–59	3.4	81–90	0.8
8	51.0	23	6.4	60	3.2	91–143	0.6
9	56.6	24–28	5.4	61	3.0	144–152	0.4
10	65.8	29	5.2	62	2.6	153–170	0.2
11	48.2	30–31	5.4				

Table 10: Distribution of depths of Taos objects

Size	Count	Size	Count	Size	Count	Size	Count
1[a]	10127.4	12	0.2	70	0.2	114	0.2
2	13.4	14	0.2	74	0.2	119	0.2
3	1.4	15	0.2	82	0.2	166	0.2
4	0.8	16	1.0	91	0.2	198	0.4
5	0.8	26	0.2	92	0.2	199	0.2
6	3.0	30	0.4	94	0.2	332	0.2
7	1.0	31	1.0	98	0.2	424	0.2
8	1.0	32	0.2	100	0.2	429	0.2
9	0.8	35	0.6	106	0.2	431	0.2
10	0.4	40	0.8				

[a] None of the objects in these singleton components contained `REF`s to themselves. Overall, only 8.6 objects (0.08%) contain `REF`s to themselves.

Table 11: Numbers of strongly connected components in Taos

ically, to allow collection by explicitly breaking cycles when a structure should be freed.) Table 11 shows the frequencies of the sizes of strongly connected components. Only 8.1% of the objects belonged to strongly connected components of size greater than 1 (*i.e.*, belonged to cycles). Moreover, only 11.2% of the objects either belonged to cycles or were reachable from cycles. (These are the objects that a purely reference-counting collector would fail to reclaim without programmer intervention. Over time, of course, they would accumulate to a larger fraction.)

5.1.4 Running measurements

The following measurements were made using a specially instrumented version of Taos that logged allocator and collector events.[16] A single instance of this Taos ran for approximately 6 hours of interactive use.

During this time, there were 1,422,626 objects allocated in Taos. At the end of the measurement, only 9,884 (0.7%) of these remained allocated; the remaining 1,412,742 (99.3%) had been freed. Since the trace did not end with a full collection, we can expect that some of the remaining 9,884 objects were in fact inaccessible.

Counting bytes, there were 172,851,756 bytes allocated (including headers and breakage); 1,353,292 (0.8%) remained allocated at the end, while 171,498,464 (99.2%) had been freed. The average object allocated was 122 bytes in size, including headers and breakage.

Figure 7 shows the size of the Taos heap, in bytes, as a function of time. After initialization, the heap size remained relatively constant; Taos had entered a steady state.

There were 9,661,068 assignments to global `REF`s or to `REF`s on the heap. This is an average of 6.8 such assignments per object allocated.

There were 3,306 reference-counting collections during the measurement. This is an average of one collection every 430 objects allocated, or every 52,284 bytes. (The collection interval was made smaller than usual during these measurements to reduce the interval between when an object becomes free and when it is collected, since only the point of collection can be measured.) There was a mark-and-sweep collection approximately every 10,000,000 bytes allocated.

In the discussion below, time is measured arbitrarily, by number of bytes allocated by the program. This metric most directly drives the collector's actions; it can also be assumed to be highly correlated with program CPU time, especially since Taos's actions are roughly the same at all points in time. The time of an object's allocation is considered to be at the end of the allocation (*i.e.*, an object could be considered to be allocated and freed at the same "time," if no other objects were allocated in between).

Figure 8 is a scatter-graph showing object allocations and deallocations. Each axis represents time, measured in bytes allocated. Each dot represents one or more objects; the x-coordinate is the time of allocation, and the y-coordinate is the time of collection. All points in the scatter-graph naturally have $x \leq y$, since objects are allocated before they are freed.

[16] The allocator and the collectors were modified to log all events, such as allocations and assignments. (Because of the implementation of `REF` assignment in Modula-2+, it was possible to log only assignments to global `REF`s and to `REF`s in heap objects; assignments to `REF` variables local to a procedure were not logged. Also, pickle allocations and deallocations were not logged, but Taos allocates few pickles, and only at initialization.) The log was buffered in memory, and asynchronously flushed to the file system by a separate thread. Care was taken that flushing the log would have minimal impact on the measurements.

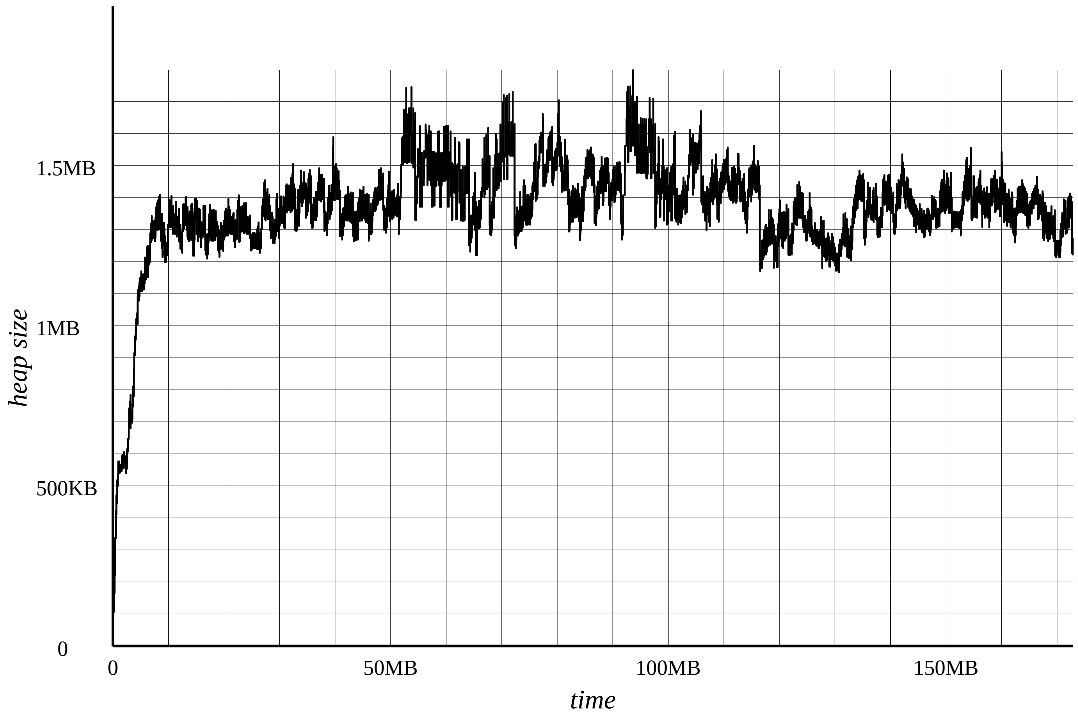

Figure 7: Heap size in Taos over time

The closely-spaced horizontal stripes in Figure 8 are reference-counting collections, where many objects are freed at about the same time. These stripes are usually indistinguishable in this figure, but sometimes are visible when the program is allocating at a high rate.[17]

We see in Figure 8 that most objects are freed shortly after they are allocated, since most objects' dots are near the $x = y$ diagonal. This repeats the Smalltalk experience reported by Ungar [18], in which most objects in Smalltalk programs were found to be short-lived. (Similar results were reported for Lisp systems by Shaw [16] and Zorn [21].) Modula-2+ is different from Smalltalk, though, in that all Smalltalk data structures are stored on the heap. Modula-2+ also provides non-heap data structures, which are used for most program operations: for instance, Modula-2+ does not use the heap for procedure activations and local variables, which are usually ephemeral. Still, even though the heap in

[17] The coarser horizontal patterns are mark-and-sweep collections, which collect cyclic structures, as well as some non-cyclic structures probabilistically missed by earlier reference-counting collections. Since the mark-and-sweep collection interval is relatively large, these objects' lifetimes may be significantly overcounted.

The coarse vertical patterns are due to many objects being allocated at about the same time and freed at different times. These patterns are synchronized to the horizontal patterns by object cleanup; the mark-and-sweep collections trigger object cleanup, and the type-specific routines in Taos for some of these objects cause new data structures to be allocated.

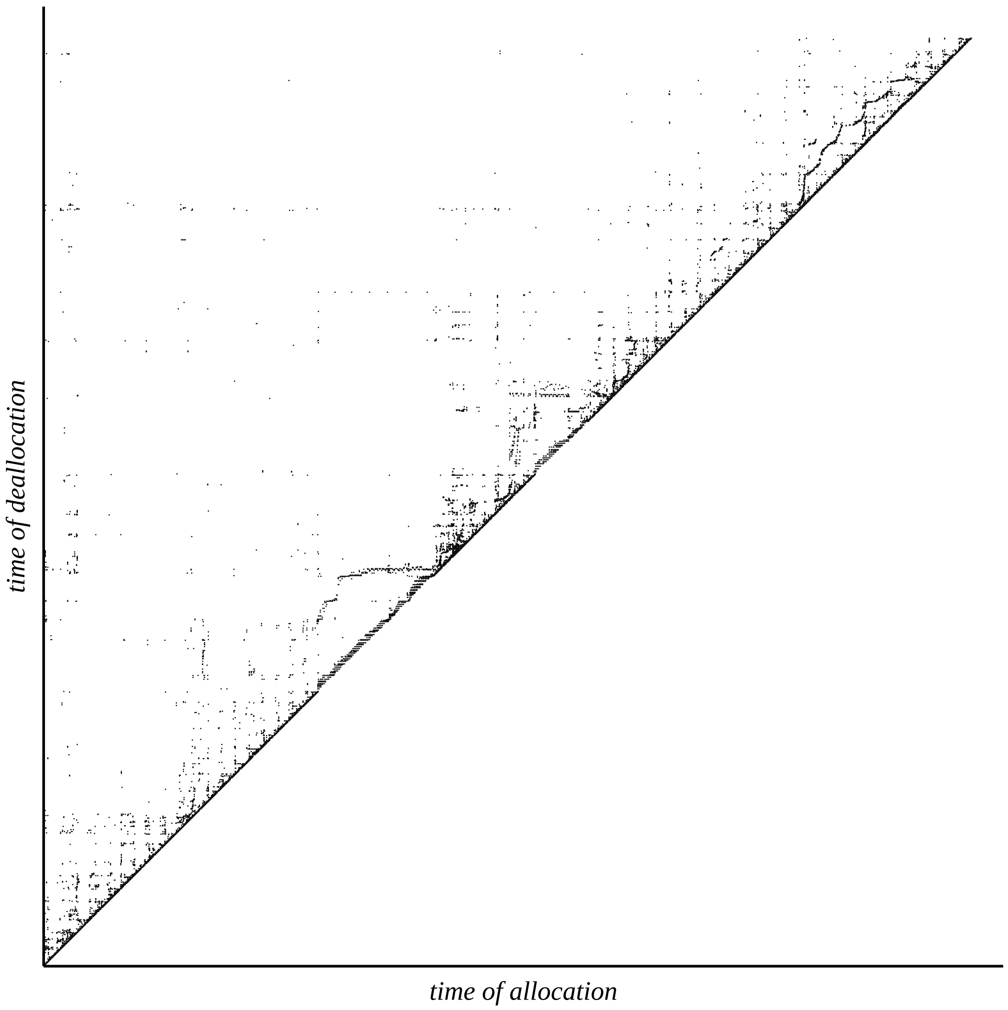

Figure 8: Object allocation time vs. deallocation time in Taos

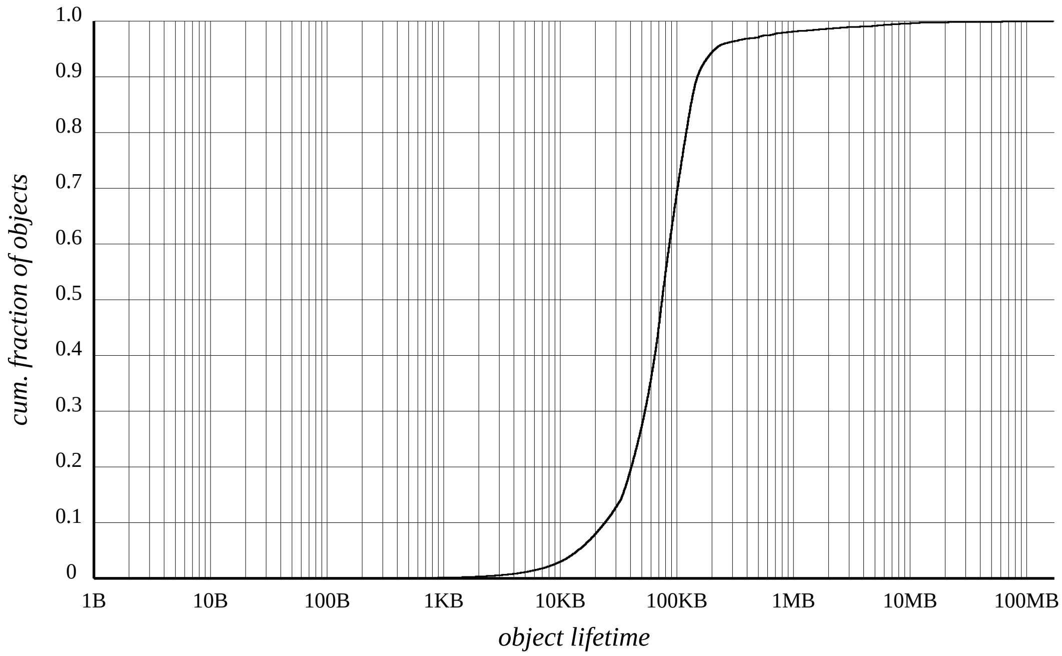

Figure 9: Object lifetimes in Taos, by cumulative objects

Modula-2+ is less biased toward holding temporaries, most objects in Taos are still relatively short-lived.

Figure 9 shows the cumulative distribution of object lifetimes in Taos. The horizontal axis is logarithmic, and shows lifetime, in bytes allocated. The vertical axis is cumulative, and counts the fraction of objects with that lifetime or less. (This curve does not reflect the lifetimes of objects that had not been freed at the end of the measurements.) The mean object lifetime measured was 58,745 bytes of allocation. Note that about 90% of the objects are freed within 3 reference-counting collections; the quantization of reference-counting collections has added an average of about 1/2 of a collection interval, or 26K bytes, to measured lifetimes.

Similarly, Figure 10 also shows the cumulative distribution of objects' lifetimes, but where the vertical axis counts the fraction of bytes allocated that were in objects with that lifetime or less. The curve is shifted downward in this figure relative to Figure 9, showing that large objects are longer-lived than small objects, as would be expected.

Figure 11 presents the lifetime distributions differently, showing the effect of the object lifetime distribution on a generational garbage collector. The horizontal axis is generation size, from 0 to 1MB; the vertical axis shows the amount of storage in the most recent generation that would not be reclaimed. As the generation size increases, the amount of storage unreclaimed also increases,

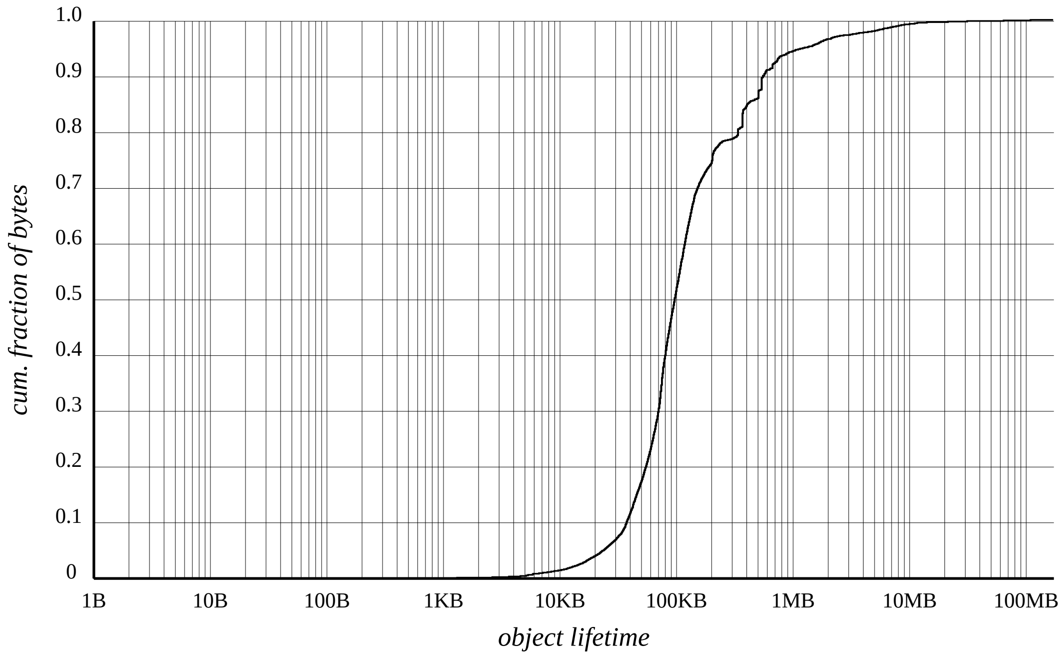

Figure 10: Object lifetimes in Taos, by cumulative bytes

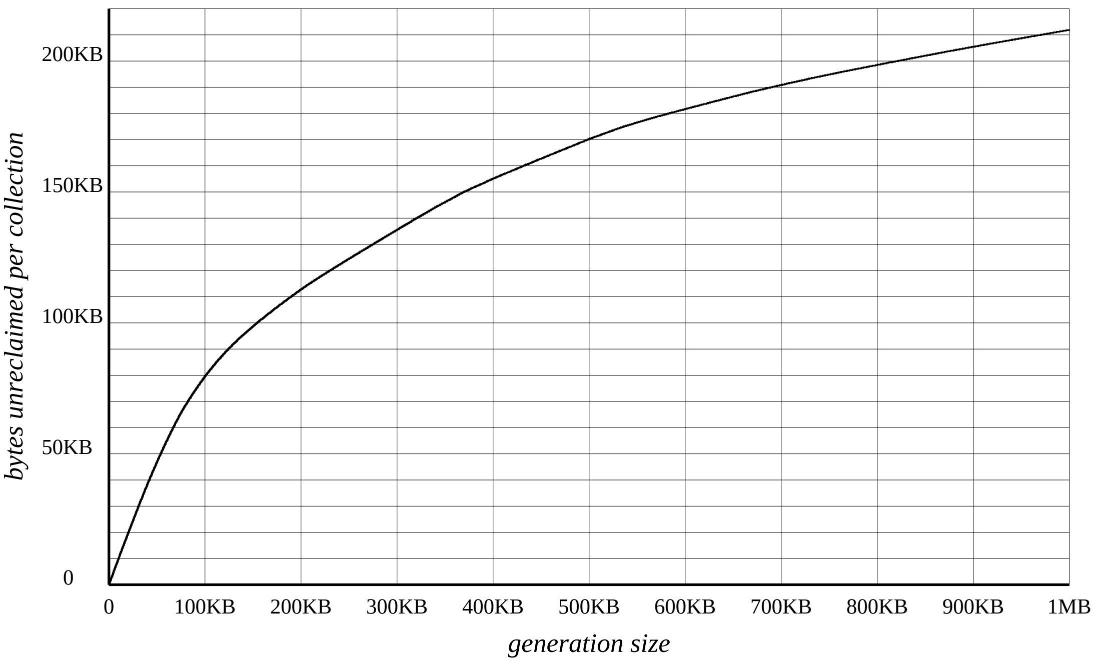

Figure 11: Storage retention in Taos with a generational GC

Figure 12: Remaining life expectancy for Taos objects

RHS	Count	Percentage
NIL	1,446,285	31.5%
Text.T constants	5,388	0.1%
into pickles	126	0.0%
backward	1,965,623	42.8%
to self	99	0.0%
forward	1,173,170	25.6%

Table 12: Distribution of RHS's in assignments to `REF` slots in Taos objects

but the fraction unreclaimed decreases. If the generation size were 1MB, then about 21% would not be reclaimed per collection of the most recent generation.

Ungar also noted that in Smalltalk, the longer an object has already lived (*i.e.*, the longer since it has been allocated), the longer it can be expected to live yet. Figure 12 shows the corresponding relation for Taos. The horizontal axis is the age of an object; the vertical axis shows the expected remaining lifetime for an object of that age. The curve has positive slope for small to medium lifetimes; the behavior after about 15MB of allocation is presumably due to the finite duration of the trace.

Finally, Ungar noted that by far most references between objects on the Smalltalk heap went from newer objects to older objects, facilitating generational garbage collection. Taos does not directly share this characteristic, as shown in Table 12, which classifies the right-hand sides of the 4,590,691 assignments to `REF` slots in heap objects. "Backward" references are from newer to older objects; "forward" pointers are from older to newer objects. Taos's assignments establish almost as many forward references as backward references, suggesting that Taos is not programmed in an applicative style.[18] The great

[18] For example, an object may be created and initialized by code such as

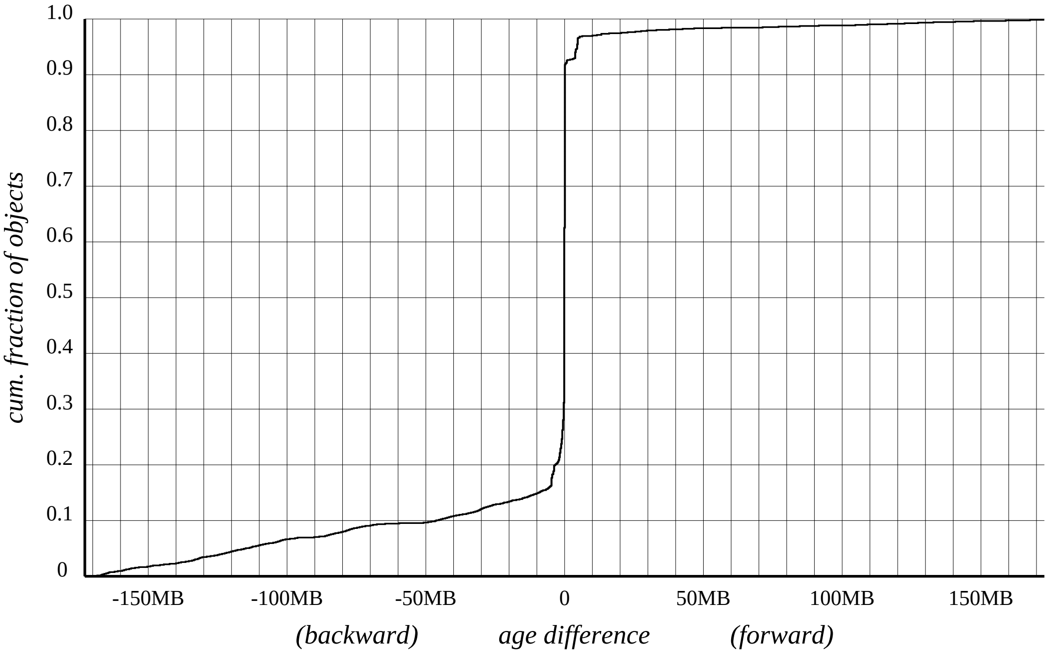

Figure 13: Cumulative distribution of age differences in Taos assignments

number of forward references might be expected to complicate the use of a generational collector with Taos because there would be so many forward references into the recently-allocated part of the heap.

However, most forward assignments do not point very far forward. Figure 13 shows the cumulative distribution of the difference in Taos's assignments between the age of the target of a **REF** and the age of the source. Negative differences denote backward references; positive differences are for forward references. Since few **REF** assignments point very far forward, there should be relatively few references from old generations to new generations. Figure 14 shows an approximation of the expected number of forward references into the new generation as a function of the generation size.[19] For example, if generations were 1MB in size, there might be an expected 600 references from older generations into the new generation at the time of a collection.

```
NEW(object);
object↑a := NewA();
object↑b := NewB();
```

Here, the fields of "object" will hold forward references.

[19]Figure 13 is based on a simple model derived from the trace measurements. It assumes that object in-degree is independent of size and lifetime, which seems unlikely.

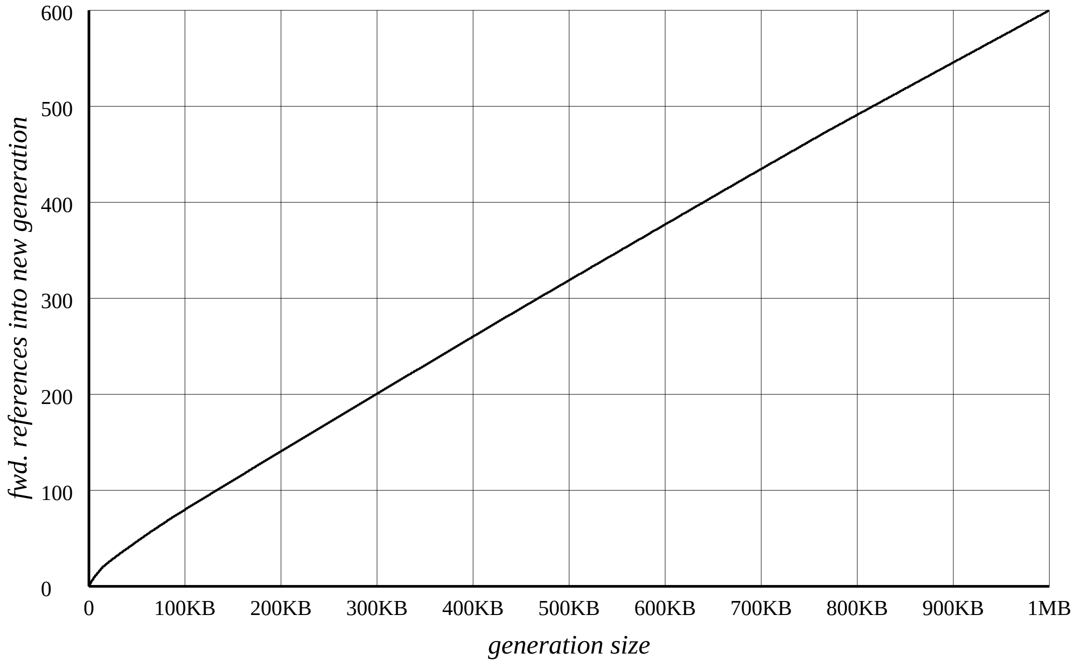

Figure 14: Expected number of forward references into new generation

5.2 The Ivy text editor

The second Topaz program measured was Ivy, a text editor; the measurements are a subset of those for Taos. Ivy is packaged as a single multithreaded server per user; it provides any number of concurrent editing windows on files. Its extension language is Tinylisp, a dialect of Lisp; it is simple to provide Tinylisp bindings for Modula-2+ procedures, such as the editing primitives.

Ivy version 28.3, the current release at the time, had 348 different `REF` types. Object cleanup was established on 13 `REF` types. The threshold reference count was 0 for one type, 1 for eleven, and 6 for one.

The static measurements performed for Taos are not repeated here; Ivy's static measurements are similar to Taos's, presumably due to the large amount of library code in both.

5.2.1 Heap count measurements

For the following measurements, a total of 33 users' instances of Ivy running at SRC were examined. As with Taos, all Ivies were quiescent.

There was a mean of 60,681 objects on the Ivy heap. Of the 33 Ivies examined, only 216 types had objects allocated at that time. Over half the objects were of just 7 types, as listed in Table 13 along with the distribution of the number of objects by deciles.

Objects	Type
11329	`List.T`[a]
4051	`Text.T`
3336	`TextTable.Entry`[b]
3114	`RefTable.Entry`[c]
3058	`SxSymbol.T`[d]
2928	`IntTable.Entry`[e]
2928	`IvyCmd.BnToCmdRef`[f]

Number of Types	Cum. Count
1	18.7%
2	25.3%
3	30.8%
5	41.0%
7	50.7%
9	58.7%
14	71.1%
20	79.7%
42	90.1%
216	100.0%

[a] As mentioned earlier, a `List.T` is a linear list of `REFANY`s

[b] A `TextTable.Entry` is an entry in a `TextTable.T`, a table with `Text.T` keys and `REFANY` values.

[c] A `RefTable.Entry` is an entry in a `RefTable.T`, a table with `REFANY` keys and values.

[d] As mentioned before, an `SxSymbol.T` represents a Lisp-style symbol, as used by Tinylisp.

[e] A `IntTable.Entry` is an entry in a `IntTable.T`, a table with integer keys and `REFANY` values.

[f] An `IvyCmd.BnToCmdRef` represents a command with a key-binding.

Table 13: Top 7 `REF` types in Ivy, by number of objects, and distribution

There was an average of 3,581,585 bytes of objects on the heap (including headers and internal fragmentation.) Almost half the bytes were in objects of just 4 types, listed in Table 14 with the distribution.

The mean object size is 57 bytes, including headers and breakage. Figure 15 shows the distribution of object sizes in Ivy, by number of objects. The median object size is 12 bytes, including headers and breakage.

Figure 16 shows the same distribution, but by number of bytes instead of by number of objects. The median of this graph is 384 bytes; about half of the bytes are allocated to objects of less than 384 bytes, and half to objects of more.

The breakage on the Ivy heap can be inferred to be between 0 and 913,215 bytes. The actual value can be expected to be about half the maximum, or 456,608 bytes. This is 7.5 bytes per object, or 13%.

The mean number of objects enabled for cleanup is 2809, of which 2555 are for Tinylisp `TLProcedure.T`s, which represent compiled Tinylisp procedures.

5.2.2 Heap shape measurements

As for Taos, the following measurements are based on 5 snapshots of Ivy, fewer than before. In these measurements, there were 68,727.6 objects per Ivy heap. (Unless otherwise stated, results given are means.)

Ivy's 1,810 global `REF`-containing variables contained 19,003 `REF` slots, only 4,977.8 (26.2%) of which were non-`NIL`; these slots held 4,755.4 different `REF` values. There were fewer large tables of `REF` values than in Taos.

The mean number of `REF` slots in heap objects was 2.7. The mean number of `REF` slots in objects with at least one `REF` slot was 3.6. Table 15 shows the

Objects	Bytes	Type
4,051	1,259,676	Text.T[a]
169	199,904	IvyB.PTable[b]
1,440	138,284	VBT.T[c]
11,329	135,951	List.T[d]

Number of Types	Cum. Bytes
1	36.2%
2	42.0%
4	49.9%
8	59.7%
15	70.5%
24	79.9%
41	90.2%
216	100.0%

[a] `Text.Ts`, immutable text strings, are used as the leaves in the representation of editing buffers.

[b] An `IvyB.PTable` is a "partition table;" it divides buffers (*e.g.*, typescripts) into partitions (*e.g.*, the history, any queued typeahead, and the line(s) currently being typed).

[c] A `VBT.T` (a "virtual bitmap terminal") is a display system window or subwindow.

[d] Most `List.Ts` in Ivy are presumably used for Tinylisp source, which is kept in Ivy for debugging purposes; the Tinylisp source appears in its original form and macro-expanded.

Table 14: Top 4 **`REF`** types in Ivy, by number of bytes, and distribution

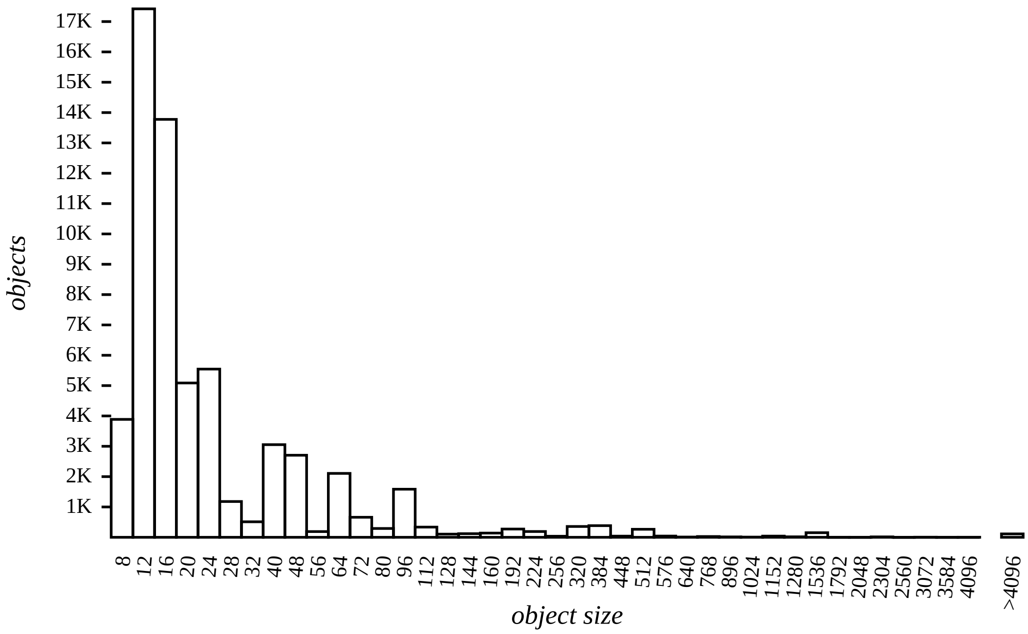

Figure 15: Distribution of objects in Ivy, by object size

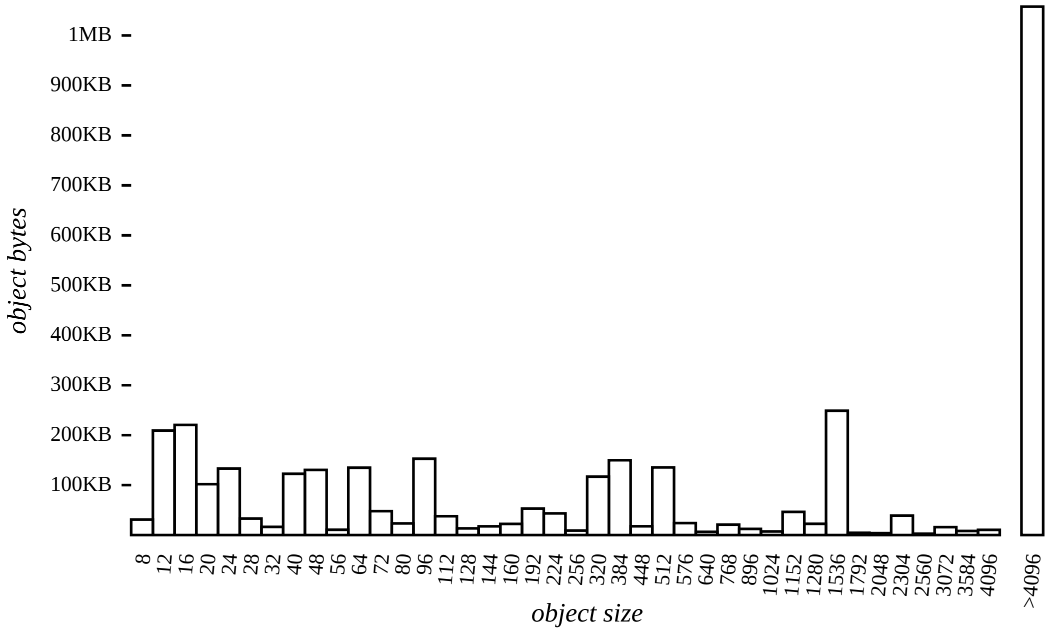

Figure 16: Distribution of object bytes in Ivy, by object size

distribution of the number of **REF** slots.

The mean in-degree of objects in the heap is 1.8. The mean in-degree of objects with non-zero in-degree is 1.9. The mode of the in-degrees is 1. Table 16 shows the distribution of in-degrees.

The mean out-degree of objects was 1.7; this is the number of non-**NIL** **REF**s in the object. The mean out-degree for nodes containing at least one non-**NIL** **REF** is 2.3. Table 17 shows the distribution of out-degrees.

Most objects on the heap are a short distance from a root. Table 18 shows the frequencies of the minimum distances from a root. There are relatively few long chains; most objects are accessible in a few **REF**s from a root.

There are far more cycles on the Ivy heap than on the Taos heap. Table 19 shows the frequencies of the sizes of strongly connected components. Overall, 31.6% of the objects belonged to strongly connected components of size greater than 1 (*i.e.*, belonged to cycles). Moreover, a full 22,716.4 (33.1%) of the objects either belonged to cycles or were reachable from cycles.

6 Other heaps

In addition to the **REF** heap, there are three less commonly used heaps in Topaz: the **POINTER** heap, the `malloc` heap, and the Nub heap. None provides garbage collection; clients are responsible for explicitly freeing storage when it is no

REF slots	Count	REF slots	Count	REF slots	Count	REF slots	Count
0	16669.4	9	1303.2	20	26.0	88	5.0
1	5480.2	10	2.2	29	113.4	114	8.0
2	27583.8	11	121.2	32	54.0	128	29.0
3	12757.8	12	2.0	34	160.4	240	78.2
4	2933.8	16	445.6	45	1.0	256	25.0
5	217.2	17	1.0	49	2.0	512	5.0
6	44.8	18	2.0	60	15.0	514	5.0
7	46.8	19	24.8	64	95.6	1024	1.0
8	468.2						

Table 15: Distribution of number of `REF` slots in Ivy objects

longer needed.

6.1 The `POINTER` heap

The `POINTER` heap coexists with the `REF` heap. When the `POINTER` heap needs to expand, it allocates new pages from VM, the same as the `REF` heap. As a result, `POINTER` heap pages and `REF` heap pages may be interspersed in memory.

Fortunately, allocations in the `POINTER` heap are rare, so that the `REF` heap remains largely contiguous, making it more likely that large objects can be allocated. The main use of the `POINTER` heap is in storage management below the level of the `REF` heap, such as maintaining internal data structures for the allocator and the collector. For example, the collector maintains various unbounded lists of objects; it allocates space as needed for these objects from the `POINTER` heap.

6.2 The `malloc` heap

The `malloc` heap is used by programs calling the C library function `malloc`. It coexists with the `POINTER` heap and the `REF` heap, but is little-used in Topaz. Part of the reason is that the implementation of `malloc` used was designed for single-threaded clients; it therefore requires multithreaded Modula-2+ clients to perform their own serialization.

6.3 The Nub heap

The Nub heap is used inside the Nub, the lowest level of the Topaz system. It provides functionality similar to the `POINTER` heap, but its implementation is customized for the more impoverished environment of the Nub. For example, the Nub heap cannot call VM to allocate more space when needed, because the Nub heap lies structurally below VM.

In-degree	Count	In-degree	Count	In-degree	Count	In-degree	Count	In-degree	Count
0	3824.0	43	3.6	88	1.0	156	0.6	315	0.4
1	50919.0	44	2.6	89	0.6	158	1.0	316	0.2
2	6491.4	45	3.2	91	0.2	161	0.2	317	0.2
3	3926.4	46	2.6	92	1.4	162	1.0	318	0.2
4	1190.6	47	5.0	95	0.8	165	0.4	319	0.2
5	624.4	48	2.0	96	1.2	167	1.2	320	0.4
6	410.4	49	0.6	97	1.4	169	0.4	327	0.2
7	228.8	50	1.6	98	0.2	170	0.2	333	0.2
8	163.8	51	0.8	100	0.2	174	0.2	342	0.2
9	181.6	52	4.4	101	1.2	176	0.2	345	0.2
10	55.2	53	4.2	103	0.4	177	1.0	351	0.2
11	61.0	54	1.2	104	1.0	178	1.2	374	0.2
12	47.8	55	2.2	108	1.0	179	0.2	379	0.2
13	33.6	56	1.2	109	0.2	180	1.0	421	0.2
14	26.4	57	3.2	110	1.6	181	0.2	422	0.2
15	50.8	58	4.2	111	1.0	182	0.2	453	0.4
16	27.0	59	0.6	112	0.2	183	0.2	454	0.2
17	23.8	60	2.4	113	0.2	192	1.0	457	0.2
18	24.0	61	1.4	114	1.0	193	0.2	458	0.2
19	14.8	62	4.2	116	1.0	199	0.2	476	0.2
20	16.6	63	1.4	117	0.4	202	1.0	479	0.2
21	13.4	64	1.0	119	1.0	208	0.2	480	0.2
22	15.0	65	4.2	121	1.2	210	0.4	495	0.2
23	8.6	66	1.0	122	0.2	211	0.2	496	0.2
24	8.8	67	1.6	123	1.4	214	0.2	627	0.2
25	7.8	68	1.2	125	0.2	215	0.2	636	0.2
26	18.6	69	2.6	126	1.2	217	1.2	641	0.2
27	10.6	70	0.8	127	0.2	222	0.4	646	0.2
28	8.2	71	0.2	128	0.2	228	1.0	658	0.2
29	80.8	72	2.4	130	0.2	232	0.2	666	0.2
30	14.8	73	1.2	136	1.0	240	0.6	669	0.2
31	3.8	74	1.4	138	0.2	241	0.2	679	0.2
32	5.2	75	2.4	140	1.2	244	0.2	690	0.2
33	5.2	76	1.8	141	0.2	245	0.2	695	0.2
34	2.8	77	1.2	142	0.4	249	1.0	708	0.2
35	7.0	78	1.4	143	0.6	251	1.0	713	0.2
36	5.2	79	0.6	144	0.4	252	2.0	765	0.2
37	2.8	80	0.4	145	0.2	258	0.2	860	0.4
38	5.8	81	0.4	146	0.4	274	1.0	861	0.6
39	7.4	83	0.2	147	0.2	286	0.2	863	0.2
40	4.2	84	0.4	151	0.4	304	0.2	1052	0.2
41	3.4	85	1.2	152	0.2	313	0.4	1436	1.0
42	5.0	86	0.2	153	0.2	314	0.2	1902	0.2

Table 16: Distribution of in-degrees of Ivy objects

Out-degree	Count	Out-degree	Count	Out-degree	Count	Out-degree	Count
0	18017.8	24	2.0	60	25.6	116	1.0
1	15598.6	25	6.2	61	1.0	118	1.0
2	24773.4	26	2.2	62	2.0	125	1.0
3	7496.2	27	3.0	64	1.0	128	1.0
4	972.2	29	1.0	65	0.2	135	1.0
5	877.8	30	1.0	70	1.0	136	1.0
6	158.4	31	0.4	71	2.0	137	1.0
7	230.6	32	3.0	75	0.2	145	1.0
8	74.8	33	3.0	76	1.0	160	1.0
9	69.4	34	1.0	79	1.0	163	1.0
10	34.2	35	1.0	80	3.0	167	12.0
11	40.2	36	3.0	81	2.0	168	1.0
12	21.2	37	2.0	82	2.0	188	1.0
13	14.0	39	2.0	83	1.0	208	1.0
14	18.0	40	2.2	85	1.0	209	1.0
15	34.0	41	3.2	86	1.0	221	1.0
16	10.6	42	1.6	88	1.0	238	1.0
17	6.0	44	2.0	89	1.0	260	1.0
18	6.0	45	2.0	91	1.0	261	1.0
19	3.4	47	0.2	94	1.0	299	1.0
20	28.8	48	1.2	98	1.0	320	1.0
21	65.0	50	2.0	100	0.2	373	1.0
22	20.4	55	1.0	106	1.0	695	1.0
23	11.2	57	12.0	110	2.0		

Table 17: Distribution of out-degrees of Ivy objects

Depth	Count	Depth	Count	Depth	Count	Depth	Count
0	8736.8	13	927.8	26	202.4	39	17.8
1	3865.2	14	875.4	27	170.2	40	12.4
2	3247.0	15	846.2	28	143.2	41	11.2
3	3954.2	16	779.8	29	125.8	42	11.0
4	6041.8	17	694.6	30	114.8	43	6.0
5	5970.6	18	631.6	31	103.2	44–46	3.0
6	5919.8	19	579.2	32	92.8	47–56	2.8
7	5131.2	20	565.4	33	80.0	57–65	1.8
8	6120.8	21	523.4	34	62.8	66–78	1.6
9	4858.8	22	385.4	35	48.8	79–123	1.4
10	1996.6	23	329.8	36	37.2	124–255	1.2
11	1112.8	24	279.4	37	26.4	256–1804	1.0
12	976.0	25	244.8	38	22.4	1805	0.4

Table 18: Distribution of depths of Ivy objects

Size	Count	Size	Count	Size	Count	Size	Count
1[a]	47026.2	17	2.8	66	0.2	181	1.0
2	614.2	18	0.2	68	1.0	187	0.2
3	3.0	19	0.4	74	0.2	188	0.2
4	50.2	20	1.6	76	1.0	189	0.8
5	0.2	22	1.0	97	1.0	237	1.0
6	13.8	26	0.2	101	1.0	293	1.0
7	1.2	28	0.2	103	1.0	310	1.0
8	31.4	29	0.2	126	1.0	670	0.2
9	0.2	30	0.2	148	0.2	691	0.2
10	17.8	31	1.0	162	0.2	692	0.2
11	1.0	34	1.0	164	0.6	13347	0.4
12	0.2	35	1.0	165	19.4	13502	0.4
14	1.2	37	1.0	166	0.4	13519	0.2
15	1.6	43	1.2	167	1.4		

[a] A total of 52 of the objects in these singleton components contained REFs to themselves. Overall, 157.2 (0.23%) of the objects on the Ivy heap contained REFs to themselves.

Table 19: Numbers of strongly connected components in Ivy

7 Summary

Garbage collection is used extensively in the Topaz computing environment. Automatic storage management simplifies the construction of large software systems, and Topaz provides a number of additional useful facilities tied to garbage collection.

The challenges of providing appropriate garbage collection are different in Topaz than in most Lisp-like environments. In particular, noticeable interruptions of service are not allowed, so concurrent collection must be used.

It seems plausible that improvements in technology will eventually make garbage collection widespread in general-purpose languages, and in systems implementation languages as well. The Topaz experience illustrates how it is possible to extend a systems implementation language with garbage collection, and successfully use it to build a large software environment. Could garbage collection work better for systems programming? Almost certainly, and the measurements listed here, along with the discussion in the companion report [4], should help define the problem. For example, the added overhead of garbage collection can be annoying in Topaz, but it is uncertain how well traditional approaches from Lisp-like environments, such as generational collection, would improve matters. New approaches may be needed.

8 Acknowledgments

Many people have contributed to the Topaz system over several years. The Modula-2+ garbage collector was initially designed by Paul Rovner and Butler

Lampson and was implemented by Paul Rovner; it was redesigned and reimplemented by John DeTreville. Pickles were designed by Butler Lampson and Paul Rovner, and implemented by Violetta Cavalli-Sforza and Bill Kalsow. Network `REF`s are due to Ted Wobber and Andrew Birrell. Ivy was designed and implemented principally by Mark Brown, Patrick Chan, and Mary-Claire van Leunen. Tinylisp and related packages like `List` and `Sx` were designed and implemented by John Ellis.

Many thanks to Sue Owicki for reviewing this paper, John Ellis for his many valuable comments on earlier drafts, and Cynthia Hibbard for many editing passes.

References

[1] A. D. Birrell, and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39-59.

[2] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. *Modula-3 Report (revised).* Research Report 52, Digital Equipment Corporation Systems Research Center, November 1989.

[3] Douglas W. Clark, and C. Cordell Green. An empirical study of list structure in Lisp. *Communications of the ACM 20,* 2 (Feb. 1977), 78-87.

[4] John DeTreville. *Experience with Concurrent Garbage Collectors for Modula-2+.* Research Report 64, Digital Equipment Corporation Systems Research Center, August 1990.

[5] L. Peter Deutsch, and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM 19,* 9 (Sept. 1976), 522-526.

[6] Adele Goldberg, and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[7] Brian W. Kernighan, and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[8] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1984.

[9] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, second edition, 1965.

[10] Paul R. McJones and Garret F. Swart. "Evolving the UNIX System Interface to Support Multithreaded Programs." Research Report 21, Digital Equipment Corporation Systems Research Center, September 1987.

[11] Paul R. McJones and Garret F. Swart. Evolving the UNIX System Interface to Support Multithreaded Programs. Proceedings, 1989 Winter USENIX Technical Conference.

[12] David Moon, Richard Stallman, and Daniel Weinreb. *LISP Machine Manual.* MIT Artificial Intelligence Laboratory, fifth edition, 1983.

[13] Paul Rovner. *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language.* Xerox Palo Alto Research Center, CRL-84-7, July 1985.

[14] Paul Rovner, Roy Levin, and John Wick. *On Extending Modula-2 For Building Large, Integrated Systems.* Research Report 3, Digital Equipment Corporation Systems Research Center, January 1985.

[15] Paul Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software,* 3(6), November 1986.

[16] Robert A. Shaw. Improving Garbage Collector Performance in Virtual Memory. Technical Report CSL-TR-87-323, Stanford University, March 1987.

[17] Daniel Swinehart, Polle Zellweger, Richard Beach, and Robert Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems 8*, 4 (October 1986), 419-490.

[18] David Michael Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, 1987.

[19] Niklaus Wirth. *Programming in MODULA-2.* Springer-Verlag, third edition, 1985.

[20] N. Wirth, and J. Gutknecht. The Oberon System. *Software—Practice and Experience 19*, 9 (September 1989), 857-893.

[21] Benjamin G. Zorn, *Comparative Performance Evaluation of Garbage Collection Algorithms.* Technical Report UCB/CSD 89/544, Computer Science Division (EECS), University of California, Berkeley, December 1989.