Chapter 5

The Beta Collector

When implementing and experimenting with a garbage collection algorithm, one is faced with the choice of either constructing an experimental simulation setup or incorporating the algorithm into a real programming language implementation. The first approach is usually somewhat simpler, but often not quite satisfactory because object graphs that evolve over time in a way similar to real settings are usually hard to construct. Also, while the first approach makes it simple to measure the individual garbage collection pauses caused by the algorithm, it becomes virtually impossible to obtain a realistic estimate of the relative amount of time spent on garbage collection.

In view of this, we chose to incorporate the Train Algorithm into an existing environment, the run-time system for the Mjølner implementation of the object-oriented Beta programming language. Beta is a comprehensive language with complex object formats and highly connected object graphs, and is as such a hard language to do garbage collection for.

In short, the Mjølner Beta System uses a modern generation-based collector with two generations, the young one being reclaimed using copying collection, and the old one using mark-sweep; the tenuring policy employed is adaptive, a separate area is used for holding certain types of large objects, and the area sizes and other parameters are user-definable. Technically, the Mjølner Beta system exists in several *variants* corresponding to different machine architectures, but the material in this chapter is essentially independent of this distinction.

In order to understand how the Train Algorithm was incorporated into the run-time system, a more in-depth knowledge of the existing setup is needed, but very little information about the implementation has been available. Therefore, this chapter gives an overview of the memory management aspects of the existing Mjølner Beta run-time system. The presentation is partially based on Madsen's account [KLMM92, Chapter 23], but most of all on the source code for the actual system.

In the following, a certain degree of knowledge about the Beta programming language will be assumed, and no attempts will be made to explain the details of the language itself. Readers unfamiliar with Beta are referred to Madsen, Møller-Pedersen & Nygaard's recent book [MMN93]. Readers highly familiar with the Mjølner Beta run-time system may skip the entire chapter.

The chapter is structured as follows. Section 5.1 describes the run-time layout of Beta objects, Section 5.2 provides an overview of the collector itself, and Section 5.3 contains a more detailed description of the different memory areas employed. Finally, Section 5.4 identifies a number problems in the existing collector and suggests a set of possible solutions.

5.1 Object Layout

At run-time, a Beta program consists of *code* to be executed, some *prototypes*, and a number of *objects* generated during program execution. The code and prototypes are allocated statically when the program is loaded, while the generated objects are allocated dynamically in a special heap structure. The prototypes contain the necessary run-time information about the various object types. Figure 5.1 illustrates a sample run-time memory structure for two object descriptors, `P1` and `P2`.

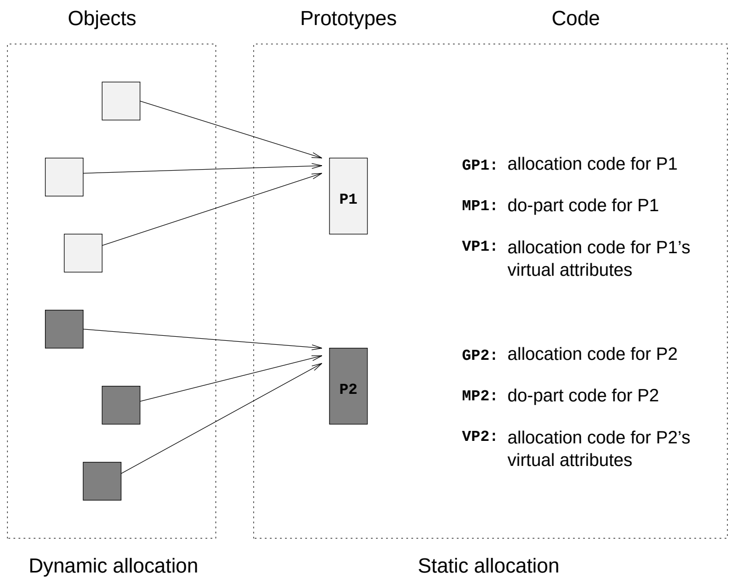
Objects
Prototypes
Code
P1
GP1: allocation code for P1
MP1: do-part code for P1
VP1: allocation code for P1's virtual attributes
P2
GP2: allocation code for P2
MP2: do-part code for P2
VP2: allocation code for P2's virtual attributes
Dynamic allocation
Static allocation

Figure 5.1: Basic memory layout

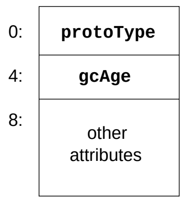

Figure 5.2: Object storage layout

The basic object storage layout is shown in Figure 5.2. The `gcAge` field is used by the garbage collector to record object ages used by the generation scavenger. The `protoType` field contains a reference to the prototype for the particular object. Instances of user-defined patterns, the so-called *item-objects*, have regular prototype pointers. In addition, there are a number of so-called *special-objects* whose prototype fields are used only for determining the general object types and therefore merely contain simple predefined values.[1] The following special-object types exist:

- *Repetition-objects* represent either *value repetitions* of the textual form `[range]@<simple pattern>` or *reference repetitions* of the textual form `[range]^<pattern>`, where the placeholder `<simple pattern>` is one of `integer`, `boolean`, `char`, or `real`.

- *Struc-objects* represent patterns as first-class citizens, generated as the result of a `<pattern>##` evaluation. A struc-object can subsequently be used to create instances of the represented pattern.

- *Component-objects* are created for each coroutine in the program, and contain the information necessary for performing context switches. The object described by the component (coroutine) declaration is represented by a regular item-object. This item-object is physically allocated immediately after the component-object and is treated as a part object thereof.

- *Component-stack-objects* are technical means for saving the execution stacks of suspended coroutines.

For the purposes of this presentation, we do not need to go into further details about the actual contents of regular and special objects (for more information, see [KLMM92, Chapter 23]). However, we do need to take a closer look at how pointers are located and followed.

Dynamic references are stored as single words, and the collector must be able to unambiguously identify these cells. When encountering a special-object, the type marker contains enough information to determine pointer locations, since all special-objects have pointers stored at fixed locations. For normal item-objects, the situation is more complicated: Each dynamic reference declared in a user-defined pattern introduces a pointer word in the layout of the corresponding object; also, one or more pointers are needed to identify the statically enclosing objects.[2] Therefore, each prototype is equipped with a *dynamic reference table*, containing all pointer location offsets. On encountering a normal item-object, the garbage collector indexes this table via the object's prototype pointer.

For efficiency reasons, most part objects are allocated in-line in the enclosing object. Only part objects whose size can not be determined at compile-time, such as instances of virtual patterns, are allocated separately. This strategy complicates garbage collection a little, since the collector can not move a part object without moving the enclosing autonomous object along with it. The

[1] Currently, these markers are simply negative numbers ranging from -1 to -8.

[2] Beta objects may have more than one static origin because subclasses of a given class can be declared at different block levels.

collector therefore needs a way to differentiate part objects from autonomous objects. The solution adapted is to store a *negative offset* in the `gcAge` attribute, pointing out the enclosing object, as illustrated in Figure 5.3.

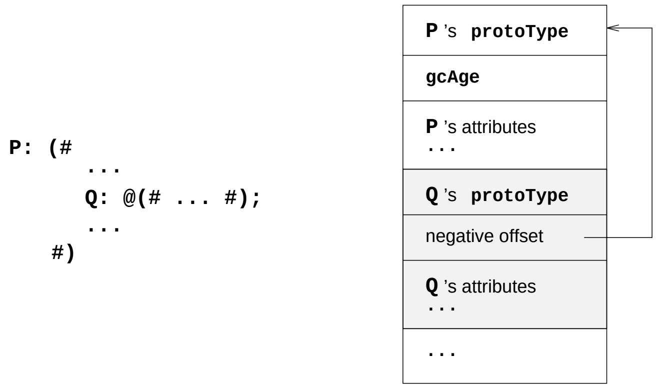

Figure 5.3: Storage layout for part objects

When following references in an object, any references contained in part objects must also be examined. Each prototype therefore further contains a *part object table*, containing the offsets to all part objects. For garbage collection purposes, this is not absolutely necessary, since the location of pointers in part objects could be stored in the dynamic reference table of the enclosing autonomous object itself. The part object table is, however, also needed at object allocation time.

As a further optimization, simple objects are allocated as raw data values: Characters and booleans are allocated as a single byte of memory, integers are allocated as single words, and reals are allocated as double words; also, dynamic references stored as single words should be added to this category.[3] This precludes language features such as pointers to integers or pointers to pointers, since it would not be possible for the garbage collector to determine the enclosing autonomous object, at least not without an unrealistically large time or space overhead.

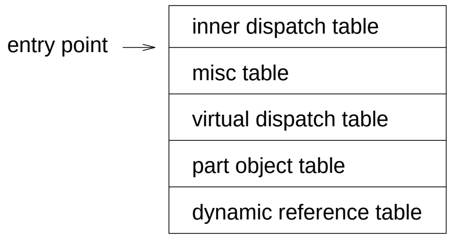

Figure 5.4: Storage layout for prototypes

To complete the picture, the general prototype structure is illustrated in Figure 5.4. The *inner dispatch table* is used for performing `inner` calls, the *misc table* contains various information (such as the fixed size of the object),

[3]Alignment constraints are enforced on single and double word entities, and autonomous objects are aligned on double word boundaries.

the *virtual dispatch table* is used for creating the correct virtual instances, and the part object and dynamic reference tables are used as described above. As illustrated, a prototype reference actually points to the start of the fixed-size misc table rather than the beginning of the prototype itself. This enables indexing in the variable-sized inner dispatch table by fixed negative offsets and in the variable-sized virtual dispatch table by fixed positive offsets.

5.2 Collector Overview

An overview of the Mjølner Beta System's memory organization is given in Figure 5.5. Notice how the layout quite closely resembles Ungar's classic generation scavenger [Ung84].

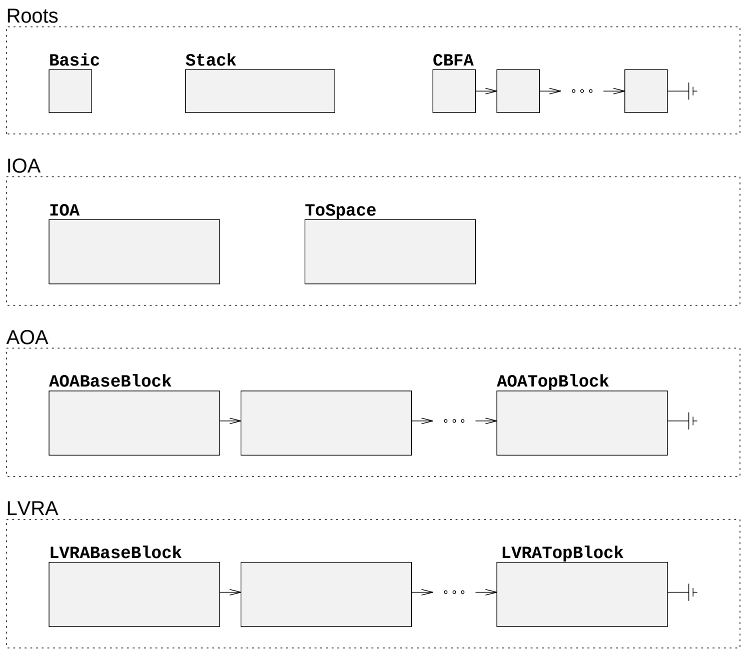

Figure 5.5: Memory organization overview

New objects are allocated in the *infant object area* (*IOA*). This area is divided in two equally sized parts, `IOA`[4] and `ToSpace`, as opposed to Ungar's three areas for new objects, past survivors, and future survivors. A scavenge copies live objects from `IOA` to `ToSpace`, followed by a swap of the two areas.

When an object reaches a certain age, measured in the number of scavenges it has survived, it is promoted to the *adult object area* (*AOA*). This area consists of a number of *blocks*, and is dynamically extended (following a strategy further

[4]We shall write IOA in Roman style when referring to the entire infant object area, and `IOA` in typewriter style when referring to the from-space part. A better term might have been `FromSpace`, but we shall henceforth adhere to the nomenclature used in the existing system.

described in Section 5.3.4). When necessary, garbage collection is performed among the tenured objects using a variant of mark-sweep collection with sliding compaction. The threshold age for promotion is adaptive and calculated using demographic feedback-mediated tenuring as described in Section 3.1.6.

As discussed in Section 3.1.4, it is generally a good idea to handle large objects in a special way to avoid the large overhead incurred by copying them. In the Mjølner Beta System, large value arrays are allocated in a separate *large value repetition area* (*LVRA*).[5] The memory in this area is managed using a free-list and compaction strategy (further described in Section 5.3.5); the fact that the area does not contain pointers makes the bookkeeping relatively simple.

The starting roots for a scavenge are found in the basic Beta environment and on the run-time stack. In addition, pointers in the *call-back function area* (*CBFA*) must be taken into consideration (as further described in Section 5.3.1).

The sizes of the different areas are user-definable, as are various other system parameters. Nevertheless, to give an impression of the proportions, the default values for the Mjølner Beta UNIX variants are as follows: The two IOA areas, the AOA blocks, and the LVRA blocks are each of size 512K bytes, the threshold size defining a large repetition is 200 elements, and the IOA survival rate adaptively aimed for is 10%. The AOA, LVRA, and CBFA areas are only allocated if needed.

5.3 Memory Areas

This section provides a more detailed description of the individual object areas and the different collection strategies employed.

5.3.1 Root Areas

The basic Beta environment has the following structure:

```
(#
  ...
  <<SLOT LIB: attributes>>
  ...
  theProgram: @| <<SLOT PROGRAM: descriptor>>
  ...
do
  ... theProgram ...
#)
```

The actual user code is placed in the `LIB` and `PROGRAM` slots, and executed when the `theProgram` component is attached by the basic environment, which is

[5]This is only a partial approach because it does not allow other big objects, such as reference repetitions or very large item-objects, to be allocated in LVRA. This strategy resembles the one used by Ungar & Jackson whose large object space is only used to hold strings and bitmaps [UJ88].

actually a component itself. There are only two basic root pointers, `BasicItem` and `ActiveComponent`. The first root references the outermost item-object shown above, which in turn refers to the basic component-object that it is part of. By following references transitively, all objects reachable from program variables are found. The second root points to the currently active component, which is not necessarily directly referenced anywhere in the program.

Also, the run-time stack needs to be examined for root pointers. This is done by traversing the stack activation records, locating and following all Beta references.[6]

Finally, root pointers may be contained in the CBFA area. When Beta code is interfaced with, say, a user interface toolkit, the toolkit's event manager, typically written in C, must be able to call Beta code when certain events occur. This is facilitated by handing out a *call-back function address* which must subsequently remain unchanged. Since introducing immovable objects would complicate garbage collection considerably, the address handed out is instead an immovable indirection entry located in the CBFA area. This entry in turn refers to a struc-object through which call-backs generate and execute pattern instances. Unfortunately, it is generally impossible to determine when a CBFA entry expires. Since the CBFA area is usually very small, it is not considered too important that the entries themselves are not reused, but obsolete CBFA entries may artificially keep large heap structures alive, so it is currently necessary to perform free calls to explicitly nullify unused entries.[7]

5.3.2 Remembered Set Representation

As described in Section 3.1.5, generation-based collectors must maintain enough run-time information to quickly identify all references from older to younger generations, used when a generation comes up for collection. In the Mjølner Beta System, this is done by maintaining a hash table, `AOAtoIOATable`, containing the addresses of all references from AOA to IOA.

Normally, a hash table is implemented by a fixed array of pointers to linked lists, sometimes called *collision lists* or *buckets* [ASU86]. Storage for the list elements can be drawn from the general storage allocator or, usually more efficiently, from a specially allocated element array. Often, the number of entries m is chosen as a prime number, and the hash function then maps the inserted element onto an integer h, and performs an h modulo m operation to determine the appropriate bucket.

[6]This is technically quite complicated, and if stack frames from external calls are involved, the approach becomes rather fragile. Currently, fairly good heuristics are employed, but in general, more information should be maintained to enable safe and unambiguous identification of the stack roots. A possible solution would be to maintain a bit vector over double word aligned memory addresses showing allocated objects.

[7]An obvious solution would be to associate a *destructor* with each object, automatically called when the object becomes garbage and is reclaimed. However, this approach would seem to rule out one of the main advantages of copying collection, namely avoiding having to deal with unused objects [Sak92]. A possible solution would be to maintain two young generation areas, one for objects without destructors (being scavenged as usual), and another for objects with associated destructors (which would have to be explicitly traversed after the evacuation phase to call the destructors associated with dead objects).

The elements inserted in hash tables are often strings, but in the Beta collector the elements are 32-bit pointers. Using the above approach would give a significant space overhead, so the `AOAtoIOATable` is implemented in a somewhat different manner: The entire hash table simply consists of an array of size m containing 32-bit entities. An address h is directly inserted at position h modulo m, provided that this entry is not already occupied. Otherwise, the address is shifted 4 bits and another insertion is tried. If this also fails, a fixed number, n, of entries following the first position are scanned, looking for a free position. Finally, if all the above fails, the entire hash table is *re-hashed* into a larger table.[8]

5.3.3 Infant Object Area

Apart from large value repetitions, all new objects are allocated from bottom to top in the `IOA` area until it runs full. This triggers an IOA collection, as illustrated in Figure 5.6.

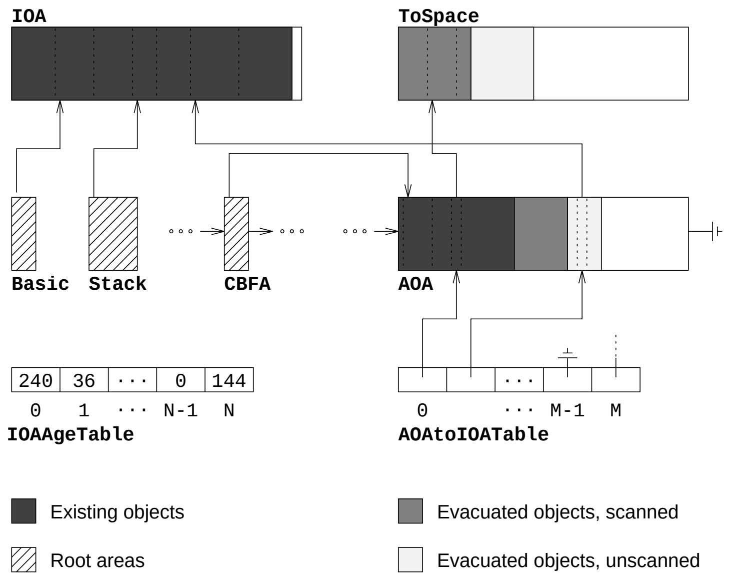

Figure 5.6: Infant Object Area

The collection proceeds as follows. First, all young objects referenced from older ones are found in the `AOAtoIOATable` remembered set and evacuated. Next, all objects referenced from the basic roots are evacuated, followed by stack and CBFA roots. The referenced objects are copied to `ToSpace` or tenured to AOA, leaving forwarding pointers behind.[9] As will be explained in Section 5.3.4, AOA space may not always be available, in which case the objects ready for promotion are temporarily moved to `ToSpace` instead.

In order to improve object locality, the scanning of evacuated objects is interleaved with the evacuation of root objects. The policy chosen is to complete

[8]The first value used for m is 5879, and the following prime numbers are chosen such that $m_{i+1} \simeq 1.5 \times m_i$. The default value for n is 100 entries.

[9]Technically, the `gcAge` field is used to hold this information.

a Cheney scan in `ToSpace` after all the AOA roots, after the basic roots, after each stack frame, and after all the CBFA roots. The scanning of newly tenured objects is, however, postponed and performed collectively at the end.

At this point, the IOA collection is completed, and this is the only point where an AOA collection can be triggered, as explained in the next section. The `IOA` and `ToSpace` pointers are now swapped, and the new tenuring threshold is calculated. When an object is copied from `IOA` to `ToSpace`, its size is added to an entry in a small integer array, `IOAAgeTable`, indexed by object age. If less than the maximum desired percentage of `ToSpace` is occupied, the tenuring threshold is set to infinity. Otherwise, the accumulated sum of the table entries is calculated and the new threshold is found. As proposed by Hudson et al. [HMDW91], stack frames in the form of component-stack-objects are always kept in IOA, and are therefore excluded from the threshold calculation.

5.3.4 Adult Object Area

During AOA collections, the memory areas are utilized as illustrated in Figure 5.7.

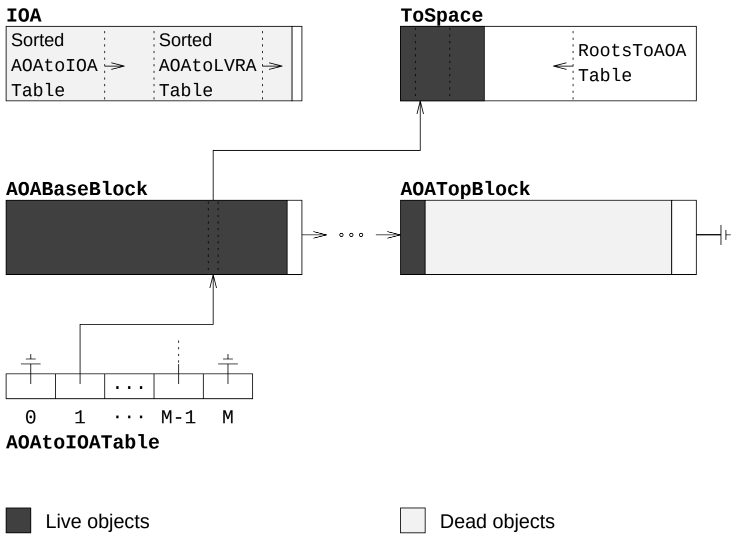

Figure 5.7: Adult Object Area

Tenured objects are allocated from bottom to top in the linked AOA blocks. When the capacity of these blocks is depleted, one has the choice of either allocating a new block or invoking an AOA collection to hopefully reclaim some AOA space. This choice is governed by a flag, `AOACreateNewBlock`. Each collection is expected to reclaim a certain (user-definable) percentage of space. If this is not accomplished, the flag is set; otherwise, it is cleared, just as the actual creation of a new block clears the flag. All AOA allocation occurs as tenuring during IOA collections, and if the previously mentioned flag is cleared, the allocation may fail, in turn causing another flag, `AOANeedCompaction`, to be set. At the end of an IOA collection, this flag determines whether an AOA collection should be invoked. Thus, the strategy is to only allocate one block at a time, and to have at least one collection with an unsatisfactory outcome before the next block is allocated.

As described in Section 3.1.5, the roots for an AOA collection must be obtained during the preceding IOA scavenge. Whenever the IOA pointer scanning process encounters a reference into AOA, it is therefore explicitly saved in a table, `RootsToAOATable`, which for economical reasons is stored from top to bottom in `ToSpace`.[10] Thus, this table is only correct immediately after an IOA collection.

The AOA collection itself is composed of a marking phase and two sweep phases. The marking phase is based on Barra's variant of Thorelli's pointer reversal technique [Tho76, Bar87, Bar88a]. The AOA object graph is scanned recursively from the roots, reversing pointers so that at the end of the traversal, each live object refers through a linked list to all the objects that originally referenced it, as illustrated in Figure 5.8.

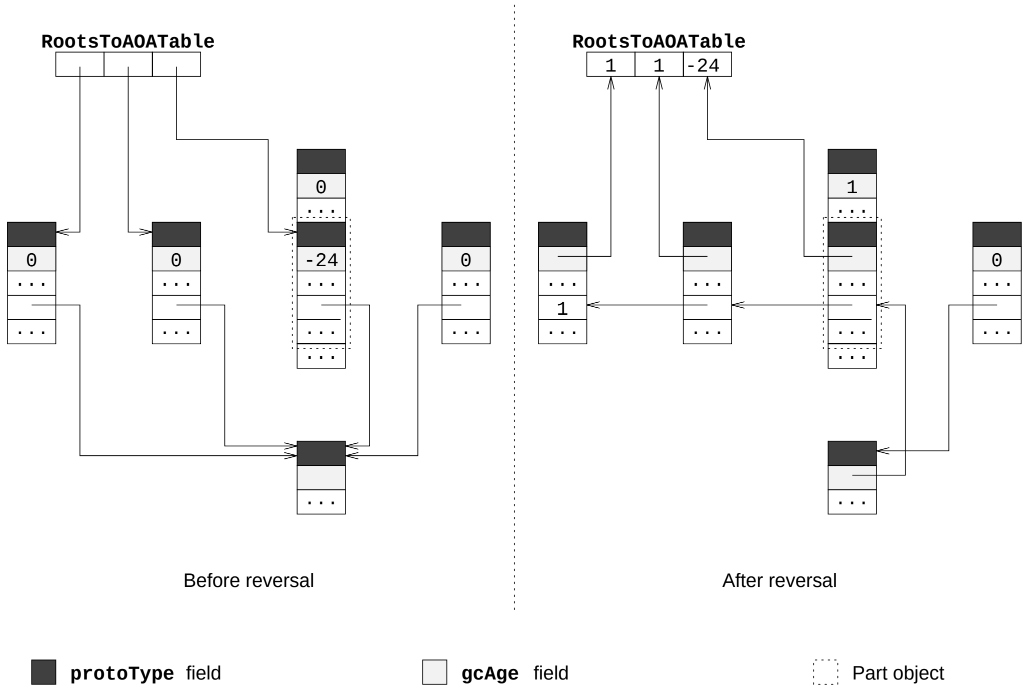

Figure 5.8: Pointer reversal in AOA

Initially, all autonomous objects are unmarked, indicated by a zero `gcAge` field. After the graph reversal, all reachable objects have obtained a non-zero `gcAge` value. The value `1` is used as a chain end-marker along with negative offsets from part objects, since these need to be reinstalled when the original object graph is restored. Also, the value `1` is used as a marking bit for objects which are not themselves referenced but contain part objects that are.

Note that this method is somewhat different from Schorr & Waite's tra-

[10]In extreme situations, `ToSpace` may not be large enough to hold both the young survivors and the `RootsToAOATable`. In this case, the table is allocated separately and deallocated after use.

ditional pointer reversal technique [SW67]. The advantage of the employed technique is that it renders the usual pointer adjustment sweep superfluous, since all pointers can be updated directly when a referenced object is moved. Thorelli also performs the object graph traversal using classic pointer reversal to embed the call stack in the object graph. In Beta, however, this traversal is done using standard recursion because the general format of Beta objects complicates standard pointer reversal considerably.

When marking is completed, the first sweep phase traverses the AOA area linearly and calculates the new object addresses for the approaching sliding compaction. At the same time, the linked lists attached to live objects are followed so that pointer fields are updated and the object graph restored by reversing pointers.

In the second sweep phase, all live objects are copied to their new locations, thereby completing the AOA collection. However, a little extra work is still needed. After having moved the AOA objects, the remembered set `AOAtoIOATable` is no longer correct. Also, since references are only added to this set at each reference assignment, not removed, it must be brought up-to-date occasionally. In the Beta collector, this is done at AOA collection time. The table is therefore copied to the lower half of the `IOA` area[11] and sorted, after which the `AOAtoIOATable` is cleared. During the second sweep phase, the sorted pointers are examined, and if they refer to objects in `ToSpace`, they are reinserted in the remembered set. The upper half of the `IOA` area is used for sorting and updating certain LVRA pointers, as described in the next section.

5.3.5 Large Value Repetition Area

The LVRA area is relatively independent of the other memory areas, which is made possible by the fact that there are only simple values and no pointers in LVRA objects. However, one still needs a way to locally determine whether a given repetition is alive. Since a Beta repetition has the property that there can exist at most one reference to it, the policy employed is to let each repetition point back to the pointer cell from which it is referenced, using the repetition's `gcAge` field to store the backwards pointer. At any point, the collector can then check to see whether this cycle is intact. If this is not the case, the LVRA object is no longer referenced and may be reclaimed, and otherwise the object may easily be moved since the single reference to it can be updated directly.[12] This setting is illustrated in Figure 5.9.

An LVRA object may die in two different ways: The reference to it may be changed to point to another repetition, for instance by extending the range of the repetition, or the object containing the reference may itself become garbage. In the first case, the backward pointer is not explicitly nullified, but the chain is nevertheless broken because the old reference is changed. In the second case, there are two situations, depending on where the referencing object resides. If a

[11]An AOA collection occurs after an IOA collection is completed, but before `IOA` and `ToSpace` are swapped. At this point, the `IOA` area contains only garbage.

[12]This strategy is actually unsafe and may lead to erroneous program behavior, as will be described in Section 5.4.4.

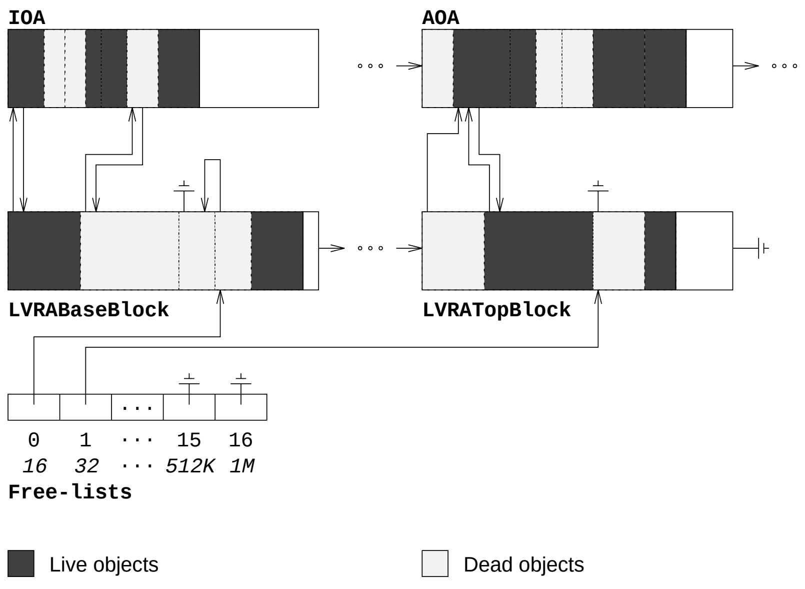

Figure 5.9: Large Value Repetition Area

repetition is referenced from a recently deceased object in IOA, the pointer cycle can not be explicitly broken since corpses in IOA are not scanned; instead, the cycle is broken at a later time when the IOA pointer field is overwritten. Pointer cycles from dead AOA objects are, however, explicitly broken since mark-sweep collectors can easily examine dead objects: When the AOA marking phase encounters an LVRA reference, it is saved in a table, `AOAtoLVRATable`, stored in the upper half of the `IOA` area. These references are subsequently sorted, and during the second sweep phase, the pointers are examined. If the referencing AOA object is still alive, the pointer cycle is updated; otherwise, it is explicitly broken.

For allocation purposes, a number of free-lists recording holes of different sizes are maintained.[13] When processing an allocation request of a certain size, the first step is to try to find a suitable hole in the free-lists. If this is not possible, the next step is to look for space at the end of the top LVRA block, since this area is not inserted in the free-lists. If this fails, and the free-lists have not been reconstructed recently, they are discarded and rebuilt. The reconstruction consists of a scan over the entire LVRA area, recording holes indicated by broken LVRA cycles. After this, the previous process is repeated, and if it fails also, a new block may be created. This choice is again governed by a flag, `LVRACreateNewBlock`. Finally, if all of the above fails, a sliding compaction is performed. The compaction process is rather straightforward, and as with AOA collections, if the collection reclaims a certain user-definable percentage of space, the `LVRACreateNewBlock` flag is cleared, or otherwise set, just as the actual creation of a new block clears the flag.

[13]Currently, the hole sizes employed are 16, 32, 64, ..., 1024K bytes. The lists are stored in the dead LVRA objects themselves.

5.4 Discussion

During the process of getting acquainted with the internal details of Beta's garbage collector, we discovered a number of errors. Most of these were relatively simple programming mistakes, but a few serious problems were also identified. This section discusses the positive and negative sides of the collector, and presents suggestions for remedying some of the latter. We would, however, like to underline that we were generally quite impressed by how well and flawlessly the current collector works in practice, especially considering its considerable complexity.

5.4.1 IOA Strategy

The parts handling young object space seem very well designed, and the performance is quite satisfactory, as will be seen in Chapter 7. A minor inexpediency is the fact that the AOA root table is constructed at every scavenge, but only used occasionally. The overhead is, however, not very large and the problem could be removed by slightly changing the AOA collection invocation strategy.

Also, the scheme employed for improving object locality by repeatedly invoking Cheney scans seems have been chosen in a somewhat ad-hoc manner. The scanning of AOA objects and subsequent evacuation of referenced IOA objects probably ought to be more intertwined.

5.4.2 Remembered Set Strategy

The pointer assignments test used to maintain the `AOAtoIOATable` remembered set has generally been carefully designed, but is sometimes disturbed by a significant error in the hash table implementation itself: Each pointer insertion in the table should result in at most one modulo operation, but may, in fact, cause up to 102 of these costly operations! However, this can be easily remedied. A more subtle error in the implementation is that a table re-hash operation in certain situations could throw away part of the original table! However, this can only happen when successive re-hashes are needed, and with the parameters currently used, the problem probably never occurs in practice.

More generally, it is not obvious that the employed hash table technique is better than the more traditional collision list approach. A few simple experiments could be conducted to decide on this issue, and to provide appropriate default values for the table sizes.

5.4.3 AOA Strategy

The generational principle seems to work rather well since most objects promoted to AOA stay alive for long periods of time, as will also be seen in Chapter 7. A fundamental problem with the old object space is, of course, that collections quite quickly become disruptive. This problem is magnified by the way the recursive AOA traversal is performed. When following a graph link, each pointer needs to be remembered, but the actual implementation stores three procedure activation records on the call stack to do this.[14] This excessive use of stack space causes thrashing and slows the AOA collections considerably, and also provokes occasional system stack overflows.

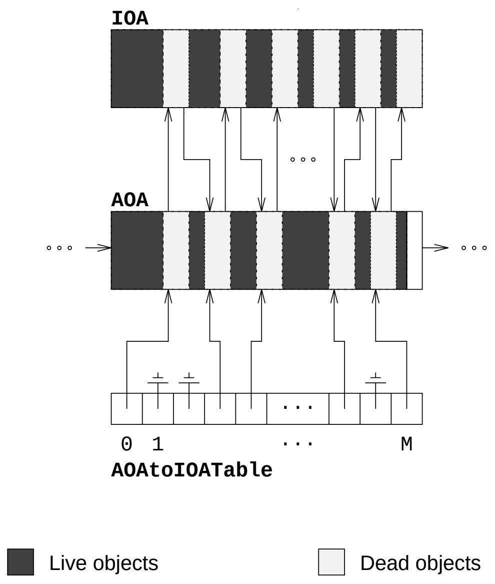

Figure 5.10: Problematic garbage structure

Another problem is that the roots to an AOA collection are inaccurately determined. Consider the garbage structure shown in Figure 5.10. The n roots for IOA from `AOAtoIOATable` induces $n-1$ roots for the next AOA collection, and it will therefore take $n-1$ AOA collections to reclaim the structure shown. This is a general problem in collectors with separate objects areas. However, since an AOA collection in Beta is already disruptive, one might as well take the time to use the real heap roots rather than building the `RootsToAOATable`. Thus, the marking phase could commence at the roots described in Section 5.3.1, and traverse both AOA *and* IOA using the current marking technique. The `gcAge` fields from IOA objects should be used as pointer chain end-markers in the same way as negative offsets from part objects. The first sweep phase should then sweep AOA as before, but also sweep IOA to reverse all pointer chains and restore `gcAge` values. Since it would probably not be worthwhile to compact the IOA area, the second sweep phase should simply move AOA objects as before.

A final potential problem with the adult object area is the non-linear behavior in the second sweep phase caused by the sorting of references. This could be avoided altogether by rescanning the slid live objects in the second sweep phase, reinserting IOA references and updating LVRA pointer cycles. Saving the old `AOAtoIOATable` and the `AOAtoLVRATable` would then no longer be needed. However, it is not immediately clear which method is most efficient in practice, so experiments would be appropriate to decide on this issue.

[14]On Motorola 680x0 architectures, three activation records occupy 48 bytes; and on Sun SPARC architectures, a massive 288 bytes are used. Thus, the storage overhead incurred by using three procedure calls to store a simple 4 byte entity is rather significant.

5.4.4 LVRA Strategy

Concerning large objects, it would probably be a good idea to use a general area in which they were all held. Currently, reference repetitions may become rather large, and when object repetitions of the form `[range]@<pattern>` are introduced in Beta, the problem will undoubtedly have to be dealt with.

Also, it seems odd that the threshold used to determine which objects are large enough to be kept in LVRA is expressed in terms of the number of elements, rather than the total size of the array. Further, the current limit of 200 elements seems somewhat arbitrary and should certainly be raised to at least 256, since the basic Beta environment contains a couple of ASCII character repetitions with 256 elements, which causes a 512K bytes LVRA block to be allocated in *all* Beta program executions! Furthermore, the heuristics used in processing allocation requests are not obviously the best strategy. Also, it seems wasteful that repetitions are not immediately inserted in the free-lists when their pointer cycles are explicitly broken or changed, but can first be reused after a free-list reconstruction.

For reasons of completeness, we should also mention that we have found and corrected a (fairly serious) memory leak in this part of the collector. However, the leak was caused by a simple programming mistake, and was easily fixed.

A more serious problem is that the LVRA pointer cycle principle is incorrect in its present state. As mentioned in Section 5.3.5, when an IOA object referencing an LVRA object dies, the pointer cycle is not explicitly broken. Instead, what normally happens is that the IOA pointer field is overwritten, thereby breaking the cycle. However, the value stored in the field could potentially be, say, an integer with the same bit pattern as the previous pointer. If an LVRA collection occurs in this situation, the cycle looks alive and the collector will slide the repetition down and erroneously change the integer's value.

Also, even though dead AOA objects are scanned, breaking AOA to LVRA cycles directly, the same problem may also occur there, albeit more indirectly: When an LVRA pointer in IOA or AOA is changed to point to another repetition, the old repetition's `gcAge` field is not nullified but still points back to the old pointer field. Of course, this field will never again be used to refer to the old repetition (which is now garbage). However, the object containing the pointer field may be moved by a subsequent IOA or AOA collection, potentially causing the previously described problem by installing an object containing an integer field in the old position.

In the current setting, the problem could be avoided by the following workaround: First, when a pointer cycle is changed or broken, the old repetition's backward pointer should be nullified. Second, before performing an LVRA collection, a scan over the LVRA blocks should nullify all backward references pointing to IOA. An IOA collection should then be provoked, which would automatically give the side effect of updating only live IOA-to-LVRA cycles! At this point, no false pointer cycles could exist, and a regular LVRA collection could therefore safely be performed.

The above method is definitely rather awkward. With a general large object area, the pointer cycle principle should probably be abandoned in favor of the more general method of using remembered sets. This is potentially more costly, but a check to determine whether a given pointer lies inside one of the LVRA blocks needs actually not be very expensive, as will be explained in Section 6.2.2.