SECTION 10

Garbage Collection

10.1 INTRODUCTION

This section describes Garbage Collection (GC), the process by which storage no longer in use can be reclaimed. The discussion starts out at an abstract level, covering the general GC algorithm, then discussing the notion of Temporal Garbage Collection (TGC). We then delve further into the specifics of the microcode and virtual machine implementation that support GC and TGC. The last part of the section documents some low-level GC functions and variables.

10.2 BASICS

Most modern symbolic computing systems implement some form of automatic memory management. This involves allocating memory for objects from available free memory as they are requested, and returning their storage to the pool of available memory when the objects are no longer in use. Automatic memory management removes the burden of explicitly managing storage from the programmer. New objects are easily manufactured and returned on demand. Later, these objects can be reclaimed by the garbage collector when it can be determined the they are no longer being used by any program.

Here is an extremely simple example of how storage is allocated and then later becomes garbage.

```
STATEMENT 1:    (SETQ my-array (MAKE-ARRAY 100.))

                ... code using MY-ARRAY ...

STATEMENT 2:    (SETQ my-array nil)
```

Statement 1 creates an array object which the programmer holds in the variable MY-ARRAY. Then at the end of the program the variable is set to NIL. At this point, as long as the array has not been stored somewhere else, it becomes garbage and can be reclaimed.

There are several common garbage collection algorithms, among them Reference Counting, Mark and Sweep, and Copying. A good introduction to garbage collection in general can be found in T. J. McEntee's, "Overview of Garbage Collection in Symbolic Computing" Texas Instruments Engineering Journal Vol 3, No 1 (January- February) pp. 130-139. The Explorer garbage collector is an incremental copying collector based on the Baker algorithm [H. G. Baker, "List Processing in Real Time on a Serial Computer," Communications of the ACM Vol 21, No 4 (April 1978) pp. 280-294.].

10. 2. 1 Garbage Collection Spaces.

Garbage Collection bases its actions on the various space type properties of the address space. A summary of GC space type terminology is given in Table 10-1. Before garbage collection begins, the address space consists of Newspace (allocated virtual memory holding objects) and Free Space (address space not yet allocated). The size of Newspace gradually increases as programs request new objects, and garbage accumulates when the programs relinquish pointers to these objects.

In the simplest kind of copying garbage collector, garbage collection begins at some point by converting some subset of Newspace into Oldspace. The portion so converted (which may be all of Newspace) is termed the Collection Space, and the process of converting it to Oldspace is called flipping or starting a collection. Oldspace then becomes the domain in which garbage collection takes place; that is, the garbage collector copies all live objects (objects determined to be still in use) out of Oldspace to into a Copyspace. Once this is done, any objects left in Oldspace are garbage, so all of the address space in Oldspace can be reclaimed (made back into Free Space). This three-part flip, collect, reclaim process is termed a collection cycle.

Table 10-1 Space Type Terminology

ADDRESS SPACE	All of the system's addressable memory.
FREE SPACE	Unused address space (not yet allocated).
ALLOCATED SPACE	Used address space (allocated).
NEWSPACE	The portion of Allocated Space which can be garbage collected (excludes Static Space). Objects in Newspace may be moved by GC. Also space where new objects are created.
STATIC SPACE	The portion of Allocated Space which is NEVER subject to garbage collection. Static Space objects will not be moved (collected) by GC, but may be Scavenged during collection cycle.
COLLECTION SPACE	A subset of Newspace which is flipped to Oldspace and collected during a GC cycle. This can be all of Newspace or just one generation of it.
OLDSPACE	The space from which GC is evacuating live objects. An invisible space.
COPYSPACE	The space to which GC is copying live objects. Also part of Scavenge Space.
SCAVENGE SPACE	All the spaces that must be examined by the scavenger in order to find all pointers to live objects. These are all the spaces which might contain references to Oldspace.
DYNAMIC SPACE	All visible spaces containing dynamic, movable objects (Newspace + Copyspace).

Since the goal of a garbage collection cycle is to copy everything useful out of Oldspace and then reclaim the storage, a copying collector does not really do garbage collection at all, but actually live data collection. The net savings gained by garbage collection is then the difference between the original Oldspace size and the size of the Copyspace where only the in-use Oldspace objects have been collected. In the worst case, when all of Oldspace contains still-used objects, the size of the Copyspace will be the same as the original Oldspace.

Because Copyspace consumes portions of the Free Space in existence at the time of the Newspace-to-Oldspace flip, this worst-case assumption means that available Free Space must be at least the size of Newspace Collection Space in order to guarantee that the collection cycle can finish. If there is a lot of garbage in the Collection Space, then Copyspace will require much less free memory than the Collection Space Size. However, to be safe, all the GC space requirement computations make the worst-

case assumption.

There are some long-term system data structures that are always in use and would never be made into garbage. Since such objects would simply be copied and re-copied by the garbage collector, they are isolated in a space called Static Space. Static Space is set aside to hold objects intended to be permanent and is never considered for Collection Space. Its object will not be collected, although it must still be purged of any Oldspace references; as with Copyspace, such references must be replace by references to the object's new location in Copyspace. The benefit of Static Space is that GC will not expend effort repeatedly copying its live objects. However this also means that Static Space objects which do happen to become garbage cannot be collected (or at least not until action is taken to convert Static Space to collectable Newspace).

10.2.2 Scavenging.

The process of identifying all live objects in Oldspace and assuring that they are transported to Copyspace is known as Scavenging. The Scavenger begins with a set of well-defined objects which form the "root" of the tree of all live Lisp objects. This root is copied to Copyspace by the flip process itself. From the root the scavenger can traces through the tree, examining every object for Oldspace references. As more Oldspace objects are found, they are copied to Copyspace and references to them are replaced with the new Copyspace location. A marker is left behind in Oldspace to indicate that this object has been traced and to redirect other references. Any future references to the same Oldspace object found will detect the "already-traced" marker, hence avoiding cycles in the tree. Since the work of copying the object has already been done, all that is required is to update the reference with the redirection to Copyspace. The object evacuated to Copyspace may now itself contain more Oldspace references, so it too must be scavenged. Scavenging will continue until the tree has been traced entirely; that is, until all Copyspace objects have been examined and any Oldspace references in them eliminated.

10.2.3 Incremental Collection.

The type of collection described so far assumes that the entire collection takes place in one stop and collect; that is, no new objects are created while the collection is in progress. While such an assumption greatly simplifies the internals of the garbage collector, it is less than useful since collections on a large virtual address of, say, 50 MB can take an hour or more. The actual Explorer garbage collector is an incremental collector. Using such a collector, the collection process can occur while other programs are running on the machine. The

incremental collection proceeds gradually, without long delays between operations noticed by running programs.

An incremental collector is more difficult to implement because care must be taken to ensure that there is no unwanted interaction between the garbage collector and the other programs running on the system, called mutators, that are creating new objects and altering old ones while the garbage collector is active. This is accomplished by enforcing a set of barrier rules which define where new objects are created, where Oldspace references can occur and what happens when a mutator makes a reference to an object in Oldspace. These rules are outlined below.

1. All new objects created after a flip occur in Newspace.

2. Oldspace references may only occur in Oldspace itself and in the as-yet unscavenged portion of Copyspace.

3. No Newspace-to-Oldspace references are allowed, although there may be Newspace-to-Copyspace references.

For simplicity's sake, these barriers are implemented by a procedure called the Read Barrier; so called because it dictates an action to be performed whenever an object is read from virtual memory. The Read Barrier states that no Oldspace references may be seen by a mutator. Any such reference will be trapped, the object will be copied out to Copyspace if necessary, and the reference actually provided to the mutator will be the object in Copyspace. Since any reference the mutator has is guaranteed not to be to Oldspace, any new objects created after the flip cannot have Oldspace objects stored into them. Therefore, Newspace allocated after a collection cycle begins does not need to be scavenged.

In such a scheme the mutator ends up doing some of the work of the scavenger; that is, the copying of an object may take place because of a mutator's dynamic reference to it rather than because of the scavenger's tracing of the static tree. For the scavenger, this just means that the first reference it sees to that object will find it already copied, saving it some work. But we will see in later discussion that there are important differences in the order that objects are copied by the two mechanisms, and that the differences can have a significant impact on system performance.

It turns out to be convenient to tie the rate of incremental garbage collection to the rate at which new storage is being requested. Thus, a certain amount of collection work (scavenging) is done for each new storage word allocated. Thus the rate of garbage collecting is proportional to the rate of consing in the system. Appropriately, this gives the consing primitives semantics both of allocating storage and working to

reclaim it.

An account is kept of how much work the scavenger needs to do. This balance is increased when new objects are created, and decreases as the scavenger does its job. Any work accomplished due to dynamic referencing of objects acts is also applied to as a credit against the scavenger's work allotment.

10.2.4 Generational Garbage Collection.

The amount of work done in a collection cycle is proportional to the size of the Collection Space and, even more strongly, proportional to the amount of live data in the Collection Space. In light of this fact a scheme which concentrates GC efforts on small, well-defined Collection Spaces which are the most likely to contain garbage can greatly increase the efficiency of garbage collection and reduce the GC overhead imposed on the mutators.

A Generational Garbage Collector defines the Collection Space on the basis of the observation that a high proportion of newly created objects become garbage quickly, while older objects tend to have a much smaller proportion of garbage. This observation can be explained simply by realizing that memory representing such things as compiled system routines, editors, and window managers, rarely become garbage, while dynamic data structures of the currently operating program tend often to be used briefly then discarded.

The generational collector partitions the address space into a number of generations, ranging from very young to very old, each of which is small when compared to the entire virtual memory space. New objects are created in the youngest generations, which is where the generational collector concentrates the majority of its efforts and where the payback in garbage collected per unit of work done is likely to be the highest. As objects survive these collections, they may be promoted into higher and higher generations where collections need be less frequent since the concentration of garbage is lower. Finally, in the oldest generations there is almost no garbage at all; they are populated by long-lived objects which have proven their worth by surviving several collections.

10.2.5 Scavenge Space.

It is important to guarantee that there is no useful object remaining in Oldspace when it is reclaimed. This requires a proper definition of Scavenge Space, the spaces which must be subjected to scavenging in order to find all references to live Oldspace objects. In other words, when a collection begins, it must have the proper root set for reaching all live data in the Collection Space.

The definition of a proper, minimal Scavenge Space is the key to efficient generational collection. It would not be worthwhile simply to isolate memory with a high concentration of garbage if it were still necessary to trace the entire tree of live Lisp objects in order to collect that space. Some mechanism must be defined for remembering and isolating references only to the live young objects. Then, just these smaller live-object subtrees need to be traced.

This task is simplified by the observation that there are three sorts of references in a Collection Space made up of one generation:

1. Reference among objects in the collected generation.

2. References from objects in a younger generation to the generation being collected.

3. References from objects in an older generation to the generation being collected.

Consider each type of reference. Note that the first set of intra-generational references are guaranteed to be taken care of by the normal scavenging mechanism as long as all references into the generation are traced. As for the second set, it is possible with little expense to scavenging all younger generations in order to find these young-to-old references. There are two reasons for this. First, since most collections are of the youngest generations there will usually only be a small number of generations younger than the one being collected (possibly even none). Secondly, the size of these younger generations is kept small by collecting them frequently and promoting survivors to higher generations. The hard part, then, is the third set of old-to-young references because they are so sparsely scattered populated over the largest amount of space.

To keep track of these references it is convenient to implement a Write Barrier which tests all objects written to memory. If the barrier detects that a younger object is being stored into an older one, a trap is taken and the reference is recorded in a generational reference list. When this generation is later collected, its Scavenge Space consists of this reference list plus all younger generations plus the Copyspace created during the collection.

10.3 EXPLORER TGC IMPLEMENTATION

The discussion so far has provided an outline and motivation for the type of garbage collection algorithm in the Explorer system,

but has glossed over all the nitty gritty implementation details. As anyone versed in the art of garbage collection implementation can attest, such an explanation seems so simple and elegant on paper that one wonders if it is an accurate way to portray the incredibly intricate code which actually embodies it. The rest of this section attempts to address these low-level details.

10.3.1 Generations.

The Explorer garbage collection implementation new with Release 3 is termed Temporal Garbage Collection (TGC). It is an extension of the incremental Release 2 garbage collection algorithm which is very low-cost because of its concentration on memory in the lower generations. There are six "logical" generations listed in Table 10-2. In contrast, the Release 2 garbage collector only defined the equivalent of the Generation 3 and and Static Generation 3 levels (plus the super-temporary Extra-PDL number consing generation, described later).

Table 10-2 TGC Generations

YOUNGEST
Extra-PDL
Generation 0
Generation 1
Generation 2
Generation 3
Static Generation 3
OLDEST

10.3.2 Areas and Regions.

All the space-type properties for Explorer virtual memory are defined on a per-region basis. Regions, which are a basic storage management unit, are described along with their attributes in the section on Storage Management. Every object is in some region that is part of an area. Each region has a space type property (NEW, OLD, COPY, STATIC, FIXED, EXTRA-PDL) which roughly corresponds with the abstract garbage collection spaces described above. FIXED space is like STATIC space for most garbage collection purposes, and the EXTRA-PDL type exists simply to flag the super-temporary number consing generation.

10.3.2.1 Volatility.

Each region also has two temporal attributes: a Generation and a Volatility. The Generation indicates the age of the objects in the region, with 0 being the youngest and 3 the oldest. Volatility specifies the kinds of references allowed by objects in this region; more specifically, it is the age of the youngest generation an object in this region can point to directly (references younger that this will be stored indirectly). A summary of the different volatility level meanings is given in Table 10-3. Normal Newspace regions are always created with volatility equal to generation. This means they can contain references to other objects in their own generation and to any older generation.

Table 10-3 Volatility Level Meanings

Volatility 3	Can point to oldest objects only.
Volatility 2	Can point to generation 2 or 3 objects.
Volatility 1	Can point to generation 1, 2 or 3 objects.
Volatility 0	Can point to object in any generation (excluding Extra-PDL).

Object Creation on the Explorer takes place on a per-area basis. Object creation in an area will cause a young Newspace region to be created (usually generation 0). When generation 0 is flipped, these regions become Oldspace. When an object is copied out of Oldspace to Copyspace, it will always be to a Copyspace region in the same area. Garbage collection does not change the area in which an object resides.

10.3.2.2 Default Cons Generation.

Each area has a default cons generation attribute which specifies the generation in which objects in this area will first be created. As they survive collections they may be promoted to higher generations (but always in the same area). Nearly every non-FIXED area in the shipped configuration has a default cons generation of 0. Extensive testing has shown that this is the best policy from a performance standpoint. An area's default can be modified with the %SET-AREA-DEFAULT-CONS-GENERATION primitive, but this is not recommended.

Because we attempt to keep the size of generation 0 small enough that collection of it can take place entirely in main memory, there is an additional policy that objects above a certain size threshold (the value stored in the %MAX-GENERATION-0-OBJECT-SIZE counter) will be not be created in generation 0. Instead, they will be consed in the generation specified by MIN(1, default-cons-generation).

10.3.3 Automatic Collection Mode.

Automatic collection means that a special GC process will monitor generation sizes and cause collection cycles to occur automatically, when certain threshold sizes are reached, invisibly to the user. The system is shipped with automatic GC on (started up by the GC-ON function). The following policies are followed by the automatic collector.

The generation 0 flip threshold is computed as a fraction of installed physical memory in an attempt to limit the size of generation 0 to an amount that minimizes the number of pages needing to be swapped in or out during the collection. The controlling variable is GC-FRACTION-OF-RAM-FOR-GENERATION-0.

The highest generation that will be collected automatically is limited to 2 (a value of *GC-MAX-INCREMENTAL-GENERATION* higher that this will be normalize to 2). This is because an incremental collection of generation 3 would greatly interfere with interactive response.

The automatic collector will always promote survivors of generation 0 into generation 1, and survivors of generation 1 into generation 2 in an attempt to keep the size of the two youngest generations manageable. Generation 2 survivors, however, will not be promoted by the automatic collector.

Flip thresholds for generations 1 and 2 are computed using worst-case 100% survival assumptions and the maximum virtual memory size of the current configuration (which is roughly the smaller of 128 MB and the amount of swap space available). Because generation 2 survivors are not promoted, generation 2 can "fill up" such that a collection of it cannot be guaranteed to complete under the worst-case assumptions. In this case, generation 2 will be "shut down" (no longer collected) and the user will be notified.

A collection of an older generation will not start (even if the flip threshold is exceeded), unless the last collection was of a younger generation. This is intended to maximize the free space available for the older generation collection and avoid thrashing.

10.3.4 Batch Collections.

The FULL-GC and GC-IMMEDIATELY functions still exist to perform batch collections. When invoked, they will turn automatic GC off (after completing any pending generational collection), then collect each generation up to a user-specified maximum with the option to promote or not. While both can be used to perform any combination of max-gen/promote collection, they are meant to have different semantics.

GC-IMMMEDIATELY by default does not promote and collects generations 0, 1, and 2. Since these generations will usually contain most of the objects created since the system was booted, this is meant to be roughly "batch collect my working set".

FULL-GC by default promotes and collects all generations including 3. It assumes you intend to DISK-SAVE afterwards so it also cleans out some big data structures in order to try to reduce band size. It is also meant to be used after doing MAKE-SYSTEMS in order to clean up the garbage in the environment, and promote all the (sure-to-be-long-lived) code just loaded into generation 3 where it won't be subject to automatic collections.

10.3.5 Scavenging for TGC.

The scavenger starts out at the beginning of a Copyspace region or other region in the Scavenge Space and proceeds forward in a basically linear fashion. Every boxed word of Copyspace containing a pointer type must be checked for reference to Oldspace. When such a reference is found, the object is copied out (if not already done) and words of data type DTP-GC-FORWARD are left behind in every slot of the structure copied (even unboxed ones). The pointer field of the GC-FORWARD points to the corresponding cell in the Copyspace representation. The GC-FORWARD serves to indirect further references to the new location in Copyspace, ensuring that after garbage collection, the copied object will still be shared in the same way the original object in Oldspace was shared. Finally, the original Oldspace reference is replaced with the a reference to the new object in Copyspace.

As scavenging progresses in the region, it updates a scavenge pointer (the REGION-GC-POINTER) which delimits the portion of the space that has been scavenged so far. The storage before the scavenge pointer cannot refer to Oldspace since it has already been scavenged. Storage beyond the scavenge pointer has not been scanned, so may still contain pointers to Oldspace. When the scavenge pointer catches up with the allocation pointer (the REGION-FREE-POINTER) in all scavengable regions of all areas, scavenging is completed. This marks the end of the collection cycle and Oldspace can be reclaimed.

As has been noted in the GC literature, the scavenger in a normal copying collector works breadth-first, which is the least desirable from an object placement point of view. In breadth-first tracing, sibling nodes in a tree are made adjacent in Copyspace. But such objects are not very likely to be near to one another ("related") in a dynamic reference chain. A depth-first or "approximately" depth-first algorithm is preferable because it tends to copy objects closer to their offspring in the tree, and this is more likely to indicate that adjacent objects are part a dynamic reference sequence. This is important since object compaction in a virtual memory system can have a

significant impact on paging performance.

The TGC scavenger is object-oriented ("approximately" depth-first). Scavenging starts at the beginning of copy space and scans each word. If a word refers to an object in old space, it is copied to the end of Copyspace and pushed on an object stack in the SCAVENGER-STATE area, along with a count of the number of boxed Qs in the object needing to be scavenged. The pushed object is now scavenged, which might again cause a copy operation and other object stack push. This recursive scavenging continues until either the depth of the stack is reached or an object is completely scavenged. In the latter case the object is popped from the stack and scavenging of its predecessor continues. If the stack becomes empty, the scavenge pointer (the REGION-GC-POINTER) beyond the end of the first object pushed, and the next linear word of Copyspace is scavenged. The object-oriented scavenging process means that some words after the scavenge pointer may already have been scavenged, but we will scavenge them again. There is no extra work done the second time, of course, outside of the memory reference.

While we have found that this algorithm is better than the undirected breadth-first scavenger, it still leaves a lot to be desired. This may be because the object tree is so bushy, and good heuristics for deciding which offspring is most important to make adjacent are difficult. However, we have found that the order in which objects are dynamically referenced does tend to provide a good heuristic for objects placement. In other words, the mutator tends to be much better at causing objects to be copied than is even the best scavenger.

As a consequence, whenever a flip takes place instead of starting up the scavenger right away, we arrange to delay for a time in order to allow the mutator to move (maybe the most frequently referenced) objects according to usage pattern. The amount of the delay is expressed in terms of a scavenger work bias. In other words, we give the scavenger a work "credit," and it will not actually kick in until enough consing has been done to cancel this credit. (The two internal counters for cons work and scavenger work are %COUNT-CONS-WORK and %COUNT-SCAVENGER-WORK). The amount of credit is the same for all generations, and is equal to the worst-case assumption amount of consing that would have to be done to drive the entire generation 0 collection. Note that because of the very high concentration of garbage in generation 0, the actual amount of consing required to drive the collection is really quite small, so that if consing were allowed to drive scavenging the collection would complete practically right away.

10.3.6 Indirection Cells.

TGC uses an indirect reference in the form of a special "super-invisible" forwarding pointer in order to implement its generational reference list for old-to-young references. Detecting such references is done as part of a Write Barrier. Every object written to virtual memory is subjected to this Write Barrier. If an attempt is made to store an object younger than a region's volatility, into an object in a region, the Write Barrier action will be to store the young object in an Indirection Cell instead. A forwarding marker is then placed in the location where the object store was intended. This marker is a word of type DTP-GC-YOUNG-POINTER whose pointer field points to the indirection cell actually containing the object. Any dynamic reference to this cell will be automatically forwarded to the indirection cell, and the object there will be referenced instead.

Indirection cells are kept in a special INDIRECTION-CELL-AREA which contains them and nothing else. Regions in this area have a special interpretation for their generation and volatility properties; these attributes are used to classify the old-to-young references of the cells in the regions.

For example, consider Figure 10-1. An attempt to store a newly-created list into the value cell of a symbol in generation 3 has caused a GCYP to be stored in the value cell instead. The GCYP points to an indirection cell which now actually contains the generation 0 list object. The indirection cell is created in an INDIRECTION-CELL-AREA region with a generation of 3 (the generation of the older structure containing the GCYP) and with volatility of 0 (the generation of the young object).

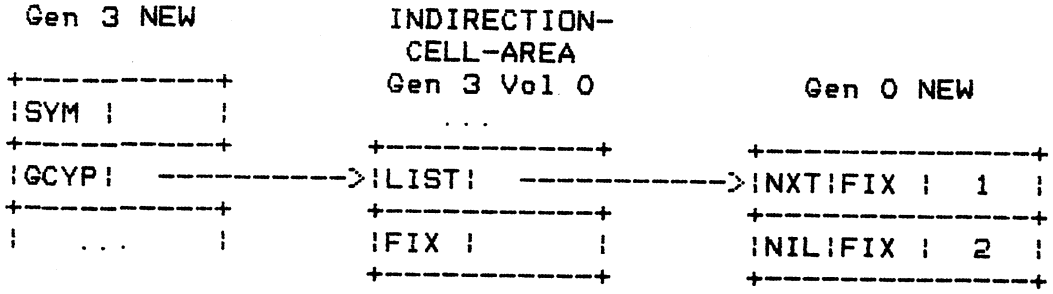

Figure 10-1 Indirection Cell Forwarding

Now when generation 0 is next collected all regions with volatility 0 will be part of the Scavenge Space because it is this in set of spaces where generation 0 references are confined. When the indirection cell is scavenged the list storage will be copied to Copyspace and the list pointer will be updated with the object's new location.

If the mutator sets the symbol to a new value before scavenging reaches the indirection cell, the new value will overwrite the indirection cell, not the value cell, even if the new value is a type which does not point to storage, say a FIXNUM. The symbol value cell will still point to the indirection cell which will then contain the FIXNUM. Now when scavenging occurs, no action is taken on this indirection cell since it contains an immediate object. This may indicate that the list previously pointed to is now garbage, if there are no other references to it.

If the mutator stores a new list, say one in generation 2, into the symbol, this same indirection cell may be reused. This is because volatility 0 means "can point to any object in generation 0 or higher"; in other words "can point to anything".

On the other hand, let's imagine the indirection cell is volatility 1. This can occur if a large object which has been created in generation 1 was originally stored in the symbol. If at this point a new generation 0 object is stored into the symbol, then a new indirection cell will need to be made because a volatility 1 location is not allowed to point to a younger generation 1 object. Now in order to satisfy the object indirection rules, a GCYP will be placed in the old volatility 1 indirection cell. This GCYP will forward references to the new volatility 0 indirection cell.

The GCYP in the symbol value cell will remain there until a collection of generation 3 takes place in which the object pointed to by the symbol is no longer in a younger generation (that is, is itself now in generation 3 or is an immediate value).

Currently there is a back pointer stored with every indirection cell object, making each cell two words long. The back pointer is simply a FIXNUM containing the address of the GCYP which created this cell in its pointer field. This back pointer exists for debugging reasons only and has no other use in the TGC internals.

10.3.7 Following GCYP Forwarding.

The GCYP is like a single-cell forwarding pointer such as DTP-ONE-Q-FORWARD in that it indirects only a single location. It is followed on both reads and writes. On writes it must be followed so that the location overwritten is the indirection cell. Without this preread, any storage pointed to by the indirection cell would be "anchored" by the pointer and could not be garbage collected.

An indirection cell contains all significant fields of the displaced object, even the cdr code. This means that the GCYP-forwarding even in cases where just the cdr code is being looking

cdr code.

NOTE

The presence of GCYPs in memory has significant consequences on the class of pointer-manipulating subprimitives that can be used to modify fields of arbitrary memory words (generally the ones beginning with %P-). TGC has changed the semantics of these subprimitives. Consult the section on storage subprimitives for more details.

10.3.8 Promotion.

When young objects are promoted into higher generations after surviving a collection they may now be in the same generation as the objects which point to them. If this is the case, when this generation is collected any GCYPs pointing to them may be snapped out. The scavenging sequence here is: read the GCYP word; follow it to the actual object. If that object points to Oldspace copy it out to Copyspace. Now attempt to store the Copyspace reference into the word where the original GCYP was read from and invoke the Write Barrier. If a volatility violation still exists, the object will be stored in an appropriate indirection cell and the GCYP's pointer field will point to it. If there is no longer a volatility violation, then the object can be stored safely into the location. In this case the indirection cell was in Oldspace and will be reclaimed when the collection is finished.

In all but a few specially marked regions, promotion changes both the generation and the volatility of objects. Generation 0 volatility 0 regions are made generation 1 volatility 1, and so forth. Volatility must track generation so that objects can continue to point freely to other objects in their own generation. The exceptions to this rule are the locked-volatility regions described later.

Whether a collection will promote or not is dictated by a flag to the microcode flipper process. When promotion occurs, say in a generation 0 collection, all generation 0 Newspace is actually made into generation 1 Oldspace whereas a non-promoting flip would leave it generation 0. When copyspace is created for this area it too will be marked as generation 1. Then the process of reclaiming oldspace simply converts Copyspace generation 1 to Newspace generation 1 and we're done.

One problem that can arise is fragmentation of generation 1 Newspace due to repeated promoting generation 0 collections. Even if only a few words survive into Copyspace, an entire 16-page region must be allocated to hold these few words. This fragmentation is avoided by converting existing, partially-filled generation 1 Newspace into Copyspace before the collection. The scavenge pointer of such a region is initialized to the region's free-pointer rather than to 0 so as to avoid unnecessary scavenging of Newspace.

10.3.9 Locked Volatility 0 Areas.

There are certain data structures that are so rapidly changing and are so likely to contain younger references that it is impractical to allow them to contain GCYPs. Runtime stacks represented by Regular PDL arrays are an example of such a structure. The overhead of managing GCYP references, especially in light of the special virtual memory reference that is done in the top, cached portion of PDLs, would be great. Similar arguments can be made for the stack group structures themselves and for Special PDL arrays. In order to prevent GCYPs from ever being created in these structures they are isolated in a few areas (the PDL-AREA for Regular PDLs and the SG-AND-BIND-PDL-AREA for stack groups and Special PDLs) and these areas are made volatility 0 (can point to anything) and have a special Volatility Locked attribute which specifies that the volatility will never change (they will always be allowed to point to anything). The special FIXED system areas in low memory are also locked volatility 0 so that GCYPs can never be stored there. This avoids the overhead of looking for GCYPs when accessing the frequently-used system structures stored there. Moreover, since these regions contain system structures which can never become garbage, they are generation 3 and are treated as Static Space. The cost paid for these locked volatility 0 regions is that they must be scavenged for every flip (because by definition they may contain references to any generation 0). Since the amount of virtual memory in these regions generally remains small, this is an acceptable cost to pay for the processing simplicity it buys.

10.4 GC SUBPRIMITIVES AND VARIABLES

This subsection documents a number of internal garbage collection primitives and variables. Since they are part of the low-level GC implementation they are subject to change without notice. All are in the SYSTEM package.

10.4.1 Flipping, Scavenging and Reclaiming Oldspace.

%gc-flip-ready Variable
Set to T by the microcode when all of Oldspace has been scavenged and can be reclaimed. Set to NIL at flip time when scavenging is started.

gc-oldspace-exists Variable
T if there is any Oldspace anywhere meaning there is a collection in progress. Set to NIL when there is no Oldspace.

gc-maybe-set-flip-ready
Scans all regions looking for Oldspace and sets the values of %GC-FLIP-READY and GC-OLDSPACE-EXISTS appropriately. Called after warm-boot and by anyone before attempting a flip.

inhibit-scavenging-flag Variable
If NIL, the scavenger is enabled and will be invoked after every consing operation. Set by GC-ON and reset by the microcode when scavenging is done.

inhibit-idle-scavenging-flag Variable
If NIL, the scavenger may be called from the scheduler if the machine is idle. When T, idle scavenging is prohibited.

%scavenger-ws-enable Variable
When a generation collection begins scavenging is temporarily disabled to allow the mutator to move objects dynamically. When this variable is true the scavenger is on hold. When NIL, scavenging can occur (when INHIBIT-SCAVENGING FLAG is also NIL).

gc-flip-now (gc-type &optional (scav-work-bias 0))
The Lisp function that initiates a collection cycle. Does some housekeeping to prepare Copyspace then calls the microcode flip routine %GC-FLIP-NOW. The generation flipped is specified by FLIP-TYPE (2 bits generation, low bit promote).

Call this to flip a generation but leave the scavenger off. See its use in START-TRAINING-SESSION.

%gc-flip (flip-type-fields)
Microcode function that flips Newspace to Oldspace according to FLIP-TYPE-FIELDS. Bit 0 of FLIP-TYPE-FIELDS is the promote bit (1 = promote). Next two bits are the generation to flip. Next 21 bits are the scavenger bias amount in words.

This function is also responsible for setting the scavenge-enable flag in any regions that will be part of the collections Scavenge Space, and for flushing the subjecting

the contents of all registers in the machine state to the Read Barrier in order to get Copyspace bootstrapped. It is called from GC-FLIP-NOW.

%gc-scavenge (work-units)
Scavenge for WORK-UNITS worth of work. This is called by the batch collection routines with an arg of a a few thousand--big enough so that scavenging will complete as quickly as possible but will come up for air occasionally in case the user wants to try and do some work while it is going on.

When automatic GC is on scavenging occurs as a side-effect of consing.

gc-reclaim-oldspace
Lisp function that reclaims the address space of any existing Oldspace. Does nothing if there is no Oldspace. Otherwise batch scavenges until %GC-FLIP-READY is true (meaning scavenging done and OK to reclaim the Oldspace). Then for all Oldspace regions of all areas, deallocates any swap space associated with the address space, unlinks the Oldspace regions from their area region lists, and returns them to the free region pool by calling microcoded %GC-FREE-REGION.

%gc-free-region (region)
Microcoded function that returns REGION to the free region pool and flushes any hardware maps still set up by its virtual pages. Used by GC-RECLAIM-OLDSPACE on Oldspace regions after scavenging is complete.

10.4.2 GC Predicates.

gc-in-progress-p
True if we're in the middle of a collection (Oldspace exists and can't yet be reclaimed).

gc-active-p
True if the automatic flipper process is active; else NIL.

current-collection-type
Returns generation and promote flag for current collection if automatic GC is active; else NIL.

10.4.3 Starting and Stopping Automatic GC.

arrest-gc reason

unarrest-gc reason

Arrest (or unarrest) the GC flipper process for reasons REASON.

gc-off-temporarily

gc-off-temporarily-back-on
Use to turn automatic GC off temporarily then back on as, for example, might be required during a patch.

10.4.4 TGC Control.

%set-area-default-cons-generation area generation
Primitive for changing the default generation in which new objects will be created in this area.

disable-tgc
Disables TGC by setting the default cons generation of all areas to 3 so there is no more young consing. Then does a full promoting collection to get rid of any indirection cells. Not recommended. Can be undone by ENABLE-TGC.

10.4.5 Number Consing.

number-cons-area Variable
The area number of the area where BIGNUMS, ratios, full-size floats and complex numbers are are consed. Normally this variable contains the area number of the EXTRA-PDL-AREA. This enables number consing, the low-overhead garbage collection of extended numbers. To disable number consing, set this variable to the number of another area.

number-gc-on (&optional on-p t)
Used to turn number consing on or off. Actually sets the variable NUMBER-CONS-AREA to the EXTRA-PDL area for true ON-P, and to the BACKGROUND-CONS-AREA for ON-P of NIL.

10.4.6 Miscellaneous.

%gc-generation-number Variable
A FIXNUM which is incremented whenever the garbage collector flips, converting one or more regions from Newspace to Oldspace. If this number has changed, the address of an object may have changed. Comparing this number with a hash table's internal GC generation number is used to cause EQ hash tables to rehash after a GC. The value cell is actually forwarded to a slot in the System Communication Area so that the changes to its value can live across a DISK-SAVE.

%region-cons-alarm Variable
Counter incremented whenever a new region is allocated.

%page-cons-alarm Variable
Counter incremented whenever a fresh page is allocated.

%gc-cons-work (nqs)
Informs the GC microcode that nqs Qs have been allocated. There is no need to do this if storage is allocated via the normal microcoded consing routines.

%gc-scav-reset (region)
Tells the scavenger to forget any work done so far in REGION and remove REGION for the cons cache. Returns T if the scavenger was actually looking at region; otherwise NIL.