GCHAX access to GC and storage allocation information

GCHAX contains functions that are useful for tracking down storage leaks, i.e., objects that ‘‘should be’’ garbage, but which do not get garbage collected. There are functions for examining reference counts, locating pointers to objects, and finding circularities (one of the chief culprits in storage leaks). Typically, you might turn to GCHAX when you notice that STORAGE claims there are more instances of a datatype than you believe there should be.

Understanding STORAGE

STORAGE displays, for each Lisp datatype, the amount of space allocated to the datatype, and how much is currently in use. The headings are:

Type	Assigned	Free items	In use	Total Alloc

Type is the name of the datatype. **Assigned** is how much of your virtual memory is set aside for items of this type. In the current implementation, memory is allocated in quanta of two pages (1024 bytes). The numbers under **Assigned** show the number of pages and the total number of items that fit on those pages. **Free items** shows how many items are available to be allocated (by the **create** construct); these constitute the ‘‘free list’’ for that datatype. **In use** shows how many items of this type are currently in use, i.e., have pointers to them and hence have not been garbage collected. If this number is higher than your program seems to warrant, you may want to look for storage leaks. The sum of **Free items** and **In use** is always the same as the total **Assigned** items. **Total Alloc** is simply the total number of items of this type that you have ever allocated, or at least since the last call to BOXCOUNT that reset the counter.

The information about number of items of type LISTP is only approximate, as lists are allocated in a special way that precludes easy computation of the number of items per page.

STORAGE also prints some summary information about how much space is allocated and available collectively for fixed-length items (mainly datatypes, both user and built-in) and variable-length items (arrays, bitmaps, strings). The variable-length items have fixed-length ‘‘headers’’, which is why they also appear in the printout of fixed-length items. Thus, the line printed for BITMAP says how many bitmaps have been allocated, but the ‘‘assigned pages’’ counts only the headers, not the space used by the variable-length part of the bitmap.

In the summary, the figure ‘‘Remaining Pages’’ for ‘‘Fixed-datum space’’ assumes a concrete upper limit on the number of pages allocatable to Fixed-datum space. The limit is actually flexible once those pages are exhausted, Fixed-datum space starts eating up unused Variable-datum space. Only when Variable-datum space is also nearly exhausted will you get a STORAGE FULL error.

STORAGE in Interlisp-D has two optional arguments that make it useful for tracking particular storage behavior:

(STORAGE *TYPES PAGETHRESHOLD*)

> If *TYPES* is given, STORAGE only lists statistics for those types. *TYPES* is an atom or list of types (either names or numbers). If *PAGETHRESHOLD* is given, then STORAGE only lists statistics for types that have at least *PAGETHRESHOLD* pages allocated to them.

For example, (STORAGE '(ARRAYP BITMAP)) lists only statistics for the types ARRAYP and BITMAP; (STORAGE NIL 6) lists only statistics for datatypes that have at least 6 pages allocated; (STORAGE) lists statistics for all datatypes, and prints the summary described above.

Tracking Storage Leaks

The functions in GCHAX are oriented toward finding leaks that involve items of some datatype not getting garbage collected. There are two main kinds of leaks: (a) items that are unintentionally being held onto, and

(b) items that no user structure is pointing to but are not collected because of the nature of Interlisp-D's garbage collector. Examples of the former are structures assigned to global variables and left there after the program finishes. Examples of the latter are principally circular structures structures where you can follow a chain of pointers from the object that eventually returns to the object. Circular lists, such as you get from `(NCONC A A)`, are a special case of circular structures. `GCHAX` is not very useful for finding ordinary circular lists, as the typical system has vast amounts of list structure, with nothing to disinguish the ''interesting'' ones. However, if the circular list also contains instances of user datatypes, then those datatypes will tend to show up as over-allocated, and hence amenable to the search functions in this package.

All the functions listed below have names beginning with ''\'' to remind you that you are dealing with system internals, and to proceed with at least a little caution. Although these functions are generally ''safe'', in that their casual use will not cause arbitrary damage, you certainly can produce unintended side effects. In particular, the functions `\SHOWGC` and `\COLLECTINUSE` have modes wherein they return a list of some kind of pointer; beware of unintentionally holding on to such a list (e.g., by having it get onto the history list), thereby preventing the eventual garbage collection of any of those pointers. Useful for keeping values off the history list are the lispxmacro `SHH` for completely inhibiting history list entry, and the idiom `(PROG1 NIL` operation `)`, e.g., `(PROG1 NIL (INSPECT` value`))` to inspect a structure without unintentionally holding on to a pointer to the inspect window.

`(\SHOWGC` *ONLYTYPES COLLECT FILE CARLVL CDRLVL MINCNT* `)`

Displays on *FILE* (default `T`) all objects in the gc reference count table whose reference count is at least mincnt, default 2. If *ONLYTYPES* is given, it is a list of datatypes (name or number) to which `\SHOWGC` confines itself. If *COLLECT* is true, `\SHOWGC` returns a list of all the objects it displays. *CARLVL* and *CDRLVL* are print levels affecting the displaying of lists; they default to 2 and 6, respectively. In the listing, collision entries in the reference count table are tagged with a *. Reference count operations on pointers in collision entries are much slower than on non-collision entries.

Note that if *COLLECT* is true, then the reference count of all the collected items is now one greater, due to the pointer to each from the list returned.

`(\REFCNT` *PTR* `)`

Returns the current reference count of *PTR*. Pointers that are not reference counted (e.g., litatoms and smallp's) are considered to have reference count 1. Since pointers from the stack (e.g., `PROG` variables) do not affect reference counts, it is possible for the reference count of an object to be zero without the object being garbage collected.

`(\#COLLISIONS)`

Returns a list of three elements: the number of entries in the table, the number which are in collision chains, and the ratio of these numbers.

`(\COLLECTINUSE` *TYPE)*

TYPE is a datatype name or number other than `LISTP` = 5. `\COLLECTINUSE` returns a list of all objects of that type that are thought to be in use, i.e., not free. It is useful when `(STORAGE` *TYPE* `)` shows more objects ''In use'' than you think is right, but you can't find any such pointers yourself. This obviously should be used with care.

In a correctly functioning system, `\COLLECTINUSE` is generally safe. However, if the freelist of *TYPE* has been smashed so that some free objects are not on it, this function can make matters much more confused, especially if the first 32-bit field of the datatype in question contains a pointer field.

`(\FINDPOINTER` *PTR COLLECT/INSPECT? ALLFLG* `)`

A brute-force approach to answering the question, ''Who has a pointer to *x*?'' `\FINDPOINTER` searches virtual memory, looking for places where *PTR* is stored. The search is not completely blind:

unless *ALLFLG* is true, it does not look in places that cannot have reference-counted pointers, such as pname space, or the stack. Prints out a description of each place *PTR* is found. If it is found in a list, asks whether to recursively search for pointers to the list, so you can track lists back to a more identifying place, such as a litatom value cell, or some datatype. Stops searching once it thinks it has found enough places to account for *PTR*'s reference count (unless *ALLFLG* is true).

If *COLLECT/INSPECT?* is true, `\FINDPOINTER` saves the identifiable pointers in a list, which it presents for inspection. If *COLLECT/INSPECT?* = `COLLECT`, the list of pointers is returned as value instead of being inspected.

The current version does not know how to parse array space, so it is really only helpful if the pointer is stored in fixed-length data space (i.e., in a field of a datatype, or as the top-level value of a litatom), but that handles most of the interesting cases except pointer arrays.

Of course, since it touches (potentially) a huge percentage of your virtual memory, `\FINDPOINTER` is completely disruptive of your working set.

`(\FINDPOINTERS.OF.TYPE` *TYPE FILTER* `)`

Calls `\FINDPOINTER` on each pointer of type *TYPE* that satisfies *FILTER*, which may be a function of one argument, the pointer, or a list form to evaluate, in which the variable `PTR` may be used to refer to the pointer in question. A *FILTER* of NIL is considered the true predicate. Essentially the same as `(for PTR in (\COLLECTINUSE` *TYPE* `) when` *FILTER* `do (\FINDPOINTER PTR))`, except that `\FINDPOINTERS.OF.TYPE` takes care to discard the cells of the list returned from `\COLLECTINUSE` before calling `\FINDPOINTER`, to avoid seeing one extra reference per object.

For example, `(\FINDPOINTERS.OF.TYPE 'WINDOW '(NOT (OPENWP PTR)))` searches for pointers to all windows that are not currently open.

`(\SHOWCIRCULARITY` *OBJECT MAXLEVEL* `)`

Follows pointers from *OBJECT*. If it finds a path back to itself, it prints that path. This function is not exceptionally fast, and deliberately (for performance reasons) does not detect circularities in lists; it simply bottoms out on lists at *MAXLEVEL*, which defaults to 1000. Circular lists are usually obvious enough anyway.

`(\MAPGC` *MAPFN INCLUDEZEROCNT* `)`

Maps over all entries in the gc reference count table, applying *MAPFN* to three arguments: the pointer, its reference count (an integer), and collisionp, a flag that is T if the entry is a collision entry. Entries with reference count zero are not included unless *INCLUDEZEROCNT* is true. This function underlies `\SHOWGC`. Some care is required in the writing of *MAPFN*; it should try to minimize any reference counting activity of its own, and in particular avoid anything that would decrement the reference count of the pointer passed to it.