From *Artificial Intelligence, An MIT Perspective, Vol. 2*. Winston & Brown, *eds*., MIT Press, 1979, pp. 391-396.

Optimizing Allocation and Garbage Collection of Spaces

Henry G. Baker, Jr.

Space Management Problems

MACLISP, unlike some other implementations of LISP, allocates storage for different types of objects in noncontiguous areas called *spaces*. These spaces partition the active storage into disjoint areas, each of which holds a different type of object. For example, *list cells* are stored in one space, *full-word integers* reside in another space, *full-word floating point numbers* in another, and so on.

Allocating space in this manner has several advantages. An object's type can easily be computed from a pointer to it, without any memory references to the object itself. Thus, the LISP primitive `ATOM(x)` can easily compute its result without even paging in `x`. Another advantage is that the type of an object does not require any storage within the object, so that arithmetic with hardware data types such as full-word integers can use hardware instructions directly.

There are problems associated with this method of storage and type management, however. When all data types are allocated from the same heap, there is no problem with varying demand for the different data types; all data types require storage from the same pool, so that only the total amount of storage is important. Once different data types must be allocated from different spaces, however, the relative sizes of the spaces becomes important.

We would like to find optimal policies for deciding in what ratios the spaces should be allocated so that garbage collection is minimized. Suppose, for example, that a program is in an equilibrium situation, where the rate of storage allocation for each data type is equal to the rate of garbage generation for that data type. Suppose further that the rate r_1 for data type 1 is twice the rate r_2 of data type 2, and that the number of free words in both spaces is the same. Then the program will continually run out of data type 1 before data type 2. Suppose now that we halve the free words in space 2. The user program will now run out of both kinds equally often, Furthermore, the timing and amount of garbage collection will be the same as before because the additional free words in space 2 were never used.

This analysis gives the key to optimal allocation for an equilibrium situation: balance the free storage for each data type against the rate of use of that data type. In other words, make all spaces run out of free words at the same time.

The calculation of optimal space size is now simple algebra. Let:

r_i be the rate of word usage for data type i;
F_i be the free words available for data type i;
F be the free words available to all data types.

Then for optimal allocation,

$$F_i = (r_i/\Sigma r_j)F$$

Intelligent Allocation

The question now is: "How can the rate of free storage usage for each data type be measured?" A "cons-counter" could be implemented for each data type, which would count the cells allocated for that data type, but in MACLISP this measurement is better made by the garbage collector. The `gc-daemon` interrupt, which is triggered by each garbage collection, invokes the `gc-daemon` with an argument which associates with each space four quantities: the number of words free and the size of that space, both before and after the garbage collection. This information, together with the information from the `gc-daemon` argument at the previous garbage collection, allows us to calculate the average rates of free storage usage for each space since the last garbage collection. This information allows us to use the `ALLOC` function to reallocate free storage to each space in proportion to its usage.

PDP-10 MACLISP presents another problem when reallocating. Since it does not use a compacting garbage collector, the spaces can he expanded, but never contracted. Therefore, the `gc-daemon` must be conservative in its reallocation strategy, because it can not back down on an allocation.

Suppose that we wish to achieve an overall garbage collection efficiency of m words allocated to each word traced. This means that if the total storage used by accessible structures consists of T words, then we wish to have (1+m)T words allocated to the various spaces, in total. In other words, F=mT words are free and should be divided among the spaces for their free areas. Now we have determined that the free storage for each space should be proportional to the rate of storage allocation in that space. Therefore,

$$F_i = (r_i/\Sigma r_j)mT$$

Now since spaces can be expanded but not contracted, we need only make sure that space i has *at least* F_i free words. This is achieved through the `gcsize`, `gcmax`, and `gcmin` parameters of `ALLOC`.

In MACLISP, one can communicate one's intentions with regard to the management of a space to the system by calling a function `ALLOC` with the space name and 3 parameters. The `gcsize` parameter specifies how large the space is allowed to grow before the garbage collector is called. `Gcmax` specifies the maximum size the space may grow to before triggering the `gc-overflow` interrupt. Finally, `gcmin` specifies how much of the space should be free after performing a garbage collection; if the free storage is less than `gcmin`, the garbage collector immediately allocates more storage to the space. (`Gcmin` may be specified as either an absolute number of words or a percentage).

We will make use in our `gc-daemon` of only the `gcsize` allocation parameters of each space. `Gcmin` will be set to 0 (or as small as possible), and `gcmax` will be set to infinity (or as large as possible). We ignore `gcmin` because the garbage collector uses it to to allocate space before the `gc-daemon` has had a chance to think things over. Setting `gcmin` instead of `gcsize` would mean that any decision by the daemon would not take effect until after the next garbage collection, which would greatly reduce the responsiveness of the `gc-daemon` to the current situation.

The Garbage Collection Deamon

The `gc-daemon` needs to be able to calculate two quantities for every space—the current number of accessible words and the gross number of words allocated since the last garbage collection. The current number of accessible words for a space can easily be calculated by subtracting the number of words free at the end of the current garbage collection from the size of the space at that time. The gross number of words allocated since the previous garbage collection can be calculated as the difference between the number in use at the beginning of the current garbage collection(size before minus free before) and the number accessible at the end of the previous garbage collection.

With these figures calculated for each space, it is easy to calculate the total number of accessible words and the differential rates of allocation. Taking the total free storage to be a percentage of the total accessible storage, we can divide up this free space among the spaces based on their differential rates of allocation. MACLISP is informed of this decision by setting the `gcsize` of each space to the sum of the accessible cells in that space plus the new free storage allotment just calculated. (We also round `gcsize` up to the next multiple of 512 because 512 words is the smallest unit of allocation in MACLISP).

The system could be improved by varying the allocate/mark ratio as the total number of accessible words grew. The idea is to garbage collect more to keep the working-set size small. However, since the paging depends so heavily on the current operating system load, one would need information from the operating system to make that decision properly.

The `gc-daemon` tries to divide up the free storage among the various spaces based on their relative allocation histories. This strategy hopes that the future will be like the near past(since the previous garbage collection). However, in practice, programs go through phases, with one phase requiring a drastically different mix of cell types than another. Therefore the `gc-daemon` can be wrong. Since the costs of misallocation are larger with the larger spaces, and since storage can never be retracted from a space once allocated, the `gc-daemon` may wish to hedge its bets by giving the larger spaces only partial allocations.

A `gc-daemon` having the characteristics described above is presented below in V. Pratt's CGOL input notation for LISP. This daemon stores all the information about a space on the property list of that space's name. For example, the normalized rate of list consing can be accessed by `(GET 'LIST 'ALLOCRATE).` Summary information is stored on the property list of `TOTAL-STORAGE`. The only user-settable parameter is the variable `ALLOCMARKRATIO`. This value must be a positive floating point number less than 5.0. It is set initially to 1.0. Making it smaller decreases working set size and increases garbage collection time.

```
define "GC-DAEMON" (spacelist);
  let totalaccessible = 0.0,
      totalconsed = 0.0;
  % Go through spaces and accumulate consed and accessible information. %
  for element in spacelist                  % Argument is "alist" of spaces. %
      do (let space = car(element),              % Give names to parameters. %
              freebefore = cadr(element),
              freeafter = caddr(element),
              sizebefore = cadddr(element),
              sizeafter = car(cddddr(element));
          % Compute consed since last gc and accessible new for this space. %
          consed ofg space := sizebefore-freebefore-accessible ofg space;
          totalconsed := totalconsed + consed ofq space;
          accessible ofq space := sizeafter-freeafter;
          totalaccessible := totalaccessible + accessible ofq space);
  % Store total consed, total accessible and compute total free. %
  consed ofq "TOTAL-STORAGE" := totalconsed;
  accessible ofq "TOTAL-STORAGE" := totalaccessible;
  let totalfree = allocmarkratio * totalaccessible;
  free ofq "TOTAL-STORAGE" := totalfree;
  % Go through spaces and reallocate where necessary. %
  for element in spacelist
      do (let space = car element;
          allocrate ofq space := consed ofq space / totalconsed;
          free ofq space := fix(totalfree * allocrate ofq space);
          let spcsize = accessible ofq space + free ofq space + 511.;
              if spcsize>511. then alloc([space,[spcsize,262143.,32.]]))
```