Programmed Deallocation without Dangling Reference

Gianna CIONI
Istituto di Analisi dei Sistemi ed Informatica dei C.N.R.,
00185 Roma, Italy

Antoni KRECZMAR
Instytut Informatyki Uniwersytetu Warszawskiego,
00901 Warszawa, Poland

2 March 1983

Keywords: Dynamic programming languages, programmed deallocation, run-time systems, parallelism.

1 Introduction

In the last ten years new high level programrning languages have been developed, for instance, PASCAL [14], ADA [1], CLU [10] and EDISON [7]. Some of these languages allow a dynamic storage management for the data and subprogram-units, generally they are then called dynamic languages. If such languages are to be used for real-time applications, the implementation of their run-time system has to guarantee total security, without any loss of efficiency. The main goal of this paper is to discuss this general security-with-efficiency problem and, in particular, to present a new, secure approach to storage management with interesting properties in terms of computing cost. Every high level programming language allocates memory blocks to unit instances. Instances of the so called non-addressable units, like procedures and functions in PASCAL, can be allocated and automatically deallocated using a stack. On the other hand, a stack implementation is not suitable for languages which admit addressable units, like access-type in ADA. For instance, in PASCAL non-addressable units are allocated in a stack, while addressable units are allocated in a heap. In the early '60s, the programmer was fully responsible for the deallocation of unit instances [12]. Then this technique was found to be unsafe and therefore rejected. Afterwards two other strategies have been proposed: the so-called retention and deletion strategy [2]. In a pure retention strategy, all instances (addressable and non-addressable) are kept in the memory until the available space runs out. At this point a garbage collection procedure is triggered to remove all non-accessible instances. It is well known that this strategy can be very time consuming, because of the frequent calls on the garbage collector. However, this problem has been deeply analyzed in the literature, and, even if interesting solutions have been found [6,13], many open problems remain [3]. In the deletion strategy, the no-longer-needed instances are removed as soon as possible, and the run-time system is made responsible for the efficient detection of sueh instances. Unfortunately, a significant execution time overhead could result also in this case [9]. More recently, in the implementation of some languages, a new strategy: programmed deallocation by specific commands, has been introduced (e.g., dispose of PASCAL, free of ADA, kill of LOGLAN [11]). However, also this technique has a great disadvantage, known as the dangling reference problem. In fact, when a pointer refers to a deallocated instance, the run-time system cannot discover such an incorrect reference and unpredictable results could tum out. In this paper we follow the programmed deallocation strategy, and, in order to avoid the mentioned dangling reference problem, we introduce one general data structure to manage all instances, both addressable and non-addressable, in a unique, unified environment. We also define algorithms for allocation and deallocation of unit instances and analyze their complexity.

2 General data structure

Let us first assume that a program, after being loaded, obtains a contiguous frame of the memory space for its run-time data. Let $M[0], ..., M[N]$ (see Fig. 1) denote this frame. In the proposed data structure the memory M, available for run-time data and units, is divided in two areas which are allowed to grow from opposite ends. The area where instances are allocated, denoted by INS (INstance Space), grows from $M[0]$. The other area, denoted by IAT (Indirect Address Table) grows backwards from $M[N]$ and contains the so-called indirect addresses. Two system pointers, *Lastused* and *Lastitem*, indicate the last word of the area INS and the first word of the table IAT, i.e., the last indirect address, respectively. Let $M[d], M[d+1], ..., M[d+s-l]$ denote the s contiguous locations of the area INS, where a given instance of size s is allocated (starting from the relative address d). Every component of the given instance is addressed relatively to its base address d. We assume that each instance is characterized by its size, represented as its first component, namely the content of the location $M[d]$, i.e., $M[d]$ =s. Therefore all. algorithms operating on instances treat them as logically similar objects, without any specific assumption about their physical structure. According to these assumptions we can write $INS[d], ..., INS[d+s-l]$ instead of $M[d], ..., M[d+s-l]$, and $INS[d]$ instead of s. The table IAT is an auxiliary array used for checking if a referenced instance is not deallocated and, if necessary, to access it. The entries of IAT are of constant size since they have only two components: d (the base address of an instance) and *guard_counter* (an integer value). *Guard_counter* is used for checking whether a given pointer variable points at a non-deallocated instance or has the value **none** which represents undefined reference. If b is the address of an entry of IAT, $IAT[b].d$ and $IAT[b].guard\_counter$ denote these two components respectively. In order to have a unique notation, we shall denote the size of an instance $INS[d], ..., INS[d+s-l]$ by $INS[d].size$. Consider now a pointer variable x. In our data structure its value will be always an ordered pair $(b, counter)$, where b is the address of lAT entry pointed at by x, and *counter* is an integer value, manipulated according to the rules described in the next sections. The general invariant of the data structure is the following:

(i) $x \neq$ **none** iff $counter = IAT[b].guard\_counter$.

In order to check if $x \neq$ **none** and to obtain the base address of an instance referenced by x, we should perform the actions defined by the following function member. The description of this function, and of all other programs in this paper, follows a Pascal-Iike syntax (with some slight changes). In particular we note that the elements of memory space $M[0], ..., M[N]$ are of type *address*.

```
function member( input b : address, counter: integer;
                 output d : address) : boolean;
begin
  if counter ≠ lAT[b]. guard _counter
  then
    member :=false
  else
    member := true;
    d := IAT[b].d
  fi
end member
```

It is clear that by virtue of invariant (i), the function *member* is correct, i.e., $(b, counter)$ refers to a non-deallocated instance iff $member(b, counter, d) =$ true and then the value of d yields the physical address of an instance. Thus, to prove the correctness of other procedures presented in the following sections it will be sufficient to establish that the invariant (i) always holds.

3 Instance deallocation

Let x be a pointer variable, with $(b, counter)$ as its value, and let us suppose that a deallocation operation (called $Free$) for an instance referred to by x is to be performed. First of alI, x is checked: - if $x =$ **none**, then no other actions have to be undertaken; - otherwise the corresponding instance should be deallocated. In order to preserve the general invariant, $IAT[b].guard\_counter$ is increased by one. This is the critical step of the algorithm. In fact, after this step all reference variabies, with $(b, counter)$ as values, have now the value counter different from the fresh (increased) value of $IAT[b].guard\_counter$. Of course, in this case the value $IAT[b].d$ does not point at any instance and, consequently, it may be used for other purposes. Therefore, the entry $IAT[b]$, with the new *guard_counter* value may be added to a list of available IAT entries. This list will be structured as a FIFO queue, with $IAT[Head]$ as its first element and $IAT[Tail]$ as its last one, while the corresponding $IAT[b].d$'s serve as list pointers. The last step of the algorithm releases the frame previously allocated to an instance in the memory space: - if such a frame is bordering upon the free space between the two pointers $LastUsed$ and $LastItem$, we can simply decrease $LastUsed$, - otherwise that frame may be inserted into the set of free frames. The management of this set (insertion of released frames and search for free frames of a given size) must be performed in a very efficient way. . This problem has been deeply analyzed [3,7,8]. However, for our purpose in the present paper it will be sufficient to describe two main operations: $insert(s, d)$ which inserts a free frame $INS[d], ..., INS[d + s - I]$ into the set of free frames, $search(s, d)$ which looks for a free frame of size s, and returns the physical

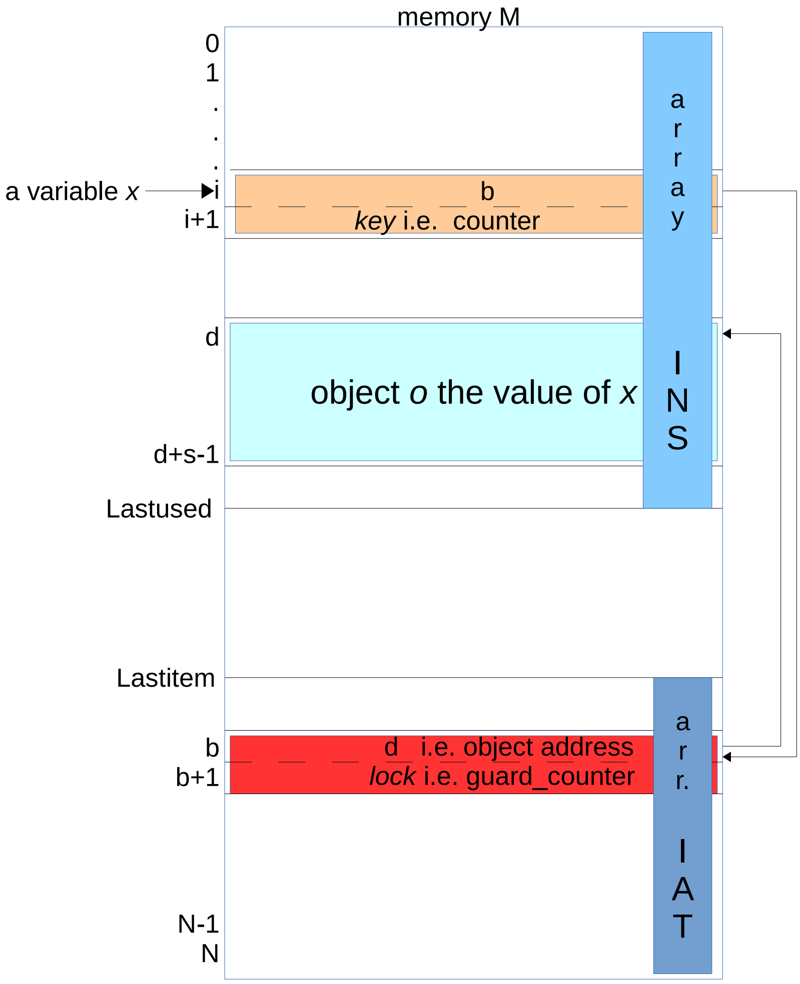

Figure 1: A variable x and its value: object o

address of a frame via output parameter d (if such a frame is found, then search is true, otherwise search is false and d is undefined). Below we present procedure $Free$.

```
procedure Free (input b: address, counter: integer);
    var d : address;
begin
  if counter ≠ IAT[b].guard_counter then return fi;
  IAT[b].guard_counter := IAT[b].guard_counter + l;
  d := IAT[b].d;
  lAT[Tail].d := b; IAT[b ].d := 0; Tail := b; {put on FIFO}
  if d + INS[d].size = LastUsed + l {bordering upon Free Space}
  then
    LastUsed := LastUsed - INS[d].size
  else
    call insert(INS[ d].size, d)
  fi
end Free
```

To prove the correctness of the procedure $Free$, it is sufficient to show that the invariant (i) always holds. For this purpose we first have to introduce three more invariants:

(ii) $0 \leq LastUsed < LastItem$,

(iii) if $x = \langle b, counter \rangle$, then $Lastltem \leq b \leq N$, $counter \leq IAT[b].guard\_counter$ and (i) holds,

(iv) $IAT[b]$ belongs to the FIFO structure if there is no $x = \langle b, counter \rangle$ such that $x \neq$ none.

The invariant (ii) guarantees that the arrays lAT and INS do not overlap, so that we do not have to consider any influence of table INS modifications on the tab le lAT, and vice versa. Moreover, the invariant (iii) is stronger than (i). Therefore, in order to guarantee the validity of (i) it will be sufficient to prove (iii). FinaIly, the invariant (iv) tells that only released entries of the lAT table are stored in FIFO. This invariant will be used in the correctness proof of the instance allocation procedure. In order to prove (ii), we observe that $LastUsed$ may be decreased by the size of an existing instance only, therefore $0 \leq LastUsed < LastItem.$ Let us consider an arbitrary pointer variable x' with $(b', counter')$ as its value. In order to prove (iii) we observe that if $b = b'$, then (iii) holds before $lAT[b].guard\_counter$ is advanced, by the inductive assumption. Then:

$$counter' < IAT[b].guard\_counter < IAT[b].guard\_counter + 1$$

. This shows that $x' =$ **none** and that (i) holds. It is clear that the conditions

$$Lastltem \leq b' \leq N \text{ and } counter' \leq IAT[b'].guard\_counter$$

are also satisfied. On the other hand, if $b \neq b'$, then (iii) immediately foIlows from the inductive assumption. Finally, (iv) is also satisfied, because lAT[b] is put on the FIFO structure when an instance is deallocated and, by (iii), we immediately obtain (iv).

4 Instance allocation

Let us consider now the problem of allocating a new instance of size s. First of all, a free indirect address entry has to be found: - if FIFO is not empty, a free address is taken from FIFO, - otherwise LastItem is decreased, if possible, and a new entry is initialized, i.e., its guard Jounter is set to 0, - when there is no enough space ($LastUsed + 2 \geq LastItem$), the compacting algorithm is triggered. When a new indirect address entry $lAT[b]$ is found, its *guard_counter* value is correct. In fact, either $IAT[b].guardJounter$ is at least greater by 1 than the *counter* of any other pointer value $(b, counter)$, or the address *nb* has not bee used because *LastItem* is decreased. At this point, to obtain a new frame of size *s*, the searching procedure is activated: - first we try simply to push *LastUsed*, - if this is not possible, the function *search* is applied, - if this application yields false, then the compacting procedure is caIled, - if, after the compaction, there is no sufficient free space, the computation, of course, will be stopped. Let us present the allocation procedure (called new):

```
procedure new (input s: integer; output b: address, counter: integer);
  var c : boolean d : address;
begin
  c := false;
  if Head = 0 {FIFO empty}
  then
    if LastItem - LastUsed ≤ 2 {no space for lAT entry}
    then
      c := true; call compactor
    fi;
    if LastItem - LastUsed ≤ 2 then {end of computation} fi;
    LastItem := LastItem - 2; b := LastItem;
    IAT[b].guard_counter := 0; {initialize new lAT entry}
  else {take from FIFO}
    b := Head; Head := IAT[b].d
  fi;
  if Lastltem - LastUsed ≤ s {Free Space too small}
  then
    if search(s, d) frame found
    then
      IAT[b].d := d; counter := lAT[b+1]. guard_counter; return
    fi;
    if c then {end of computation} else call compactor fi;
    if LastItem - LastUsed ≤ s then {end of computation} fi;
  fi;
  d := LastUsed + 1; INS[d].size := s; LastUsed := LastUsed + s;
  IAT[b].d := d; counter := IAT[b].guard_counter
end new
```

We shall prove the correctness of this procedure by proving again the invariants (ii)-(iv), assurning, of course, the correctness of procedures compactor and search. If an entry $IAT[b]$ is found in FIFO (Head $\neq 0$), then, by (iv) and (iii), $lAT[b].guard\_counter \geq counter'$ for any $x' = (b', counter')$. Thus a pair $(b, counter)$, returned via output parameters, yields a unique reference. Sirnilarly, if an $IAT[b]$ entry is obtained from M by decreasing $Lastltem$, then the pair $(LastItem - 2, 0)$ is unique, because b' can be equal to $LastItem - 2$ for any $x' = (b', counter')$. So for our $x = (b, counter)$, we have (iii) since $x \neq none$, $Lastltem \leq b \leq N$ and $counter \leq IAT[b].guard\_counter$. For any other $x' = (b', counter')$ and $b' = b$, (iii) holds by the inductive assumption. Invariant (ii) holds because $LastItem > LastUsed - 2$ when $LastItem$ is decreased by 2, and $LastItem > LastUsed + s$, when $LastUsed$ is increased by s. Invariant (iv) holds since no new $IAT[b]$ entry is inserted into FIFO.

5 Time and space cost

First of all, let us consider the question of extra space cost. The values of $guard\_counters$ and $counters$ may be quite small, e.g., one byte for each may be sufficient. To protect the system against a quick overloading of $guard\_counters$, the list of free entries from lAT is arranged as FIFO. Then during the phases where the storage management works in FIFO fashion, these free entries are taken from the other end, in order to decrease the probability of using the same entry several times. However, if a $guard\_counter$ reaches its maximal value, the corresponding entry cannot be put on the list of free entries and should remain unchanged until the compactor is applied. This causes the general invariant of the data structure to be satisfied. The space for direct and indirect addresses must be large enough to hold any reasonable address. Thus this extra space cost depends strongly on the computer. On some computers the pair $(b, counter)$ may be packed into a single word; sirnilarly we may compact the pair $(d, guard\_counter)$. Then, the pointer variables do not need extra space, although each instance needs one extra word for its indirect address (allocated at the indirect address table). The time cost of the function member is, of course, constant. This operation is called whenever a remote access is needed, therefore it should be extremely efficient. If the pair $(b, counter)$ is packed into a single word, then by storing in IAT the pairs $(b-d, guard\_counter)$ rather than pairs $(d, guard\_counter)$, we can obtain the direct address d and the difference $counter - guard\_counter$ by a single subtraction. Thus the whole operation may be performed in two or three machine instructions, depending on the computer. As far as operations $Free$ and new are concerned, their costs depend on the internal representation of the set of free frames. If we are able to perform the operations insert and search in a constant time, then the operations $Free$ and new also have constant execution times.

6 Parallelism

When several processors proceed in parallel on a common data structure, some special security measures should be taken [13]. If we want to design a secure run-time system, none of the possible parallei calls of new, Free and member should be able to destroy the data structure invariants. For each processor it will be assumed that the examination of $M[i]$, assignment to $M[i]$, advancing $M[i]$ by 1 etc. are indivisible operations (with respect to the other processors actions). A direct analysis of the operations new, $Free$ and $member$ indicates that new and $Free$ should be mutually exclusive, while $member$ may be active simultaneously with any of the other two operations. Then $Free(x)$ may be performed iff all the calls of $member(x)$ have been terminated. These constraints are collected in Table 1.

Table 1: Constraints among *new, Free* and *member*

	new(s)	*Free*(x)	*member*(x)	*member*(y)
new(s)	C	C		
Free(x)	C	C	C	
member(x)		C		
member(y)				
			x≠ y,	C for collision

It is quite evident that the synchronization between $Free(x)$ and $member(x)$ is similar to the readers-writers problem [4]. But in our case only one writer (i.e., $Free(x)$ is to be considered, since the mutual exclusion of the operations $Free$ and new guarantees that at most one Free call waits to enter the corresponding critical region. Let now extent each entry $IAT[b]$ with three new components: two boolean semaphores r and w, and one integer m. Moreover, let g be a global boolean semaphore which synchronizes the calls of $Free$ and new. We propose the following solution to the problem (in our proposal, one can find elements of the standard solution to the readers-writers problem; P and V denote usual semaphore operations).

```
function member(x);
begin
  P(IAT[b].r);
  IAT[b].m := IAT[b].m + 1;
  if IAT[b].m = 1
  then
    P(IAT[b].w)
  fi;
  V(IAT[b].r);
  ...
  lAT[b ].m := lAT[b ].m - 1;
  if IAT[b].m = 0
  then
    V(IAT[b].w)
  fi
end member

procedure Free(x);
begin
  P(g);
  P(IAT[b].r)]
  P(IAT[b].w);
  .
  .
  .
  V(IAT[b].w);
  V(IAT[b ].r);
  V(g)
end free

procedure new(s);
begin
  P(g);
  .
  .
  .
  V(g)
end new
```

Semaphore $IAT[b].r$ guarantees that when $free(x)$ passes through $P(IAT[b].r)$ no $member(x)$ may enter the corresponding critical region, hence no $free(x)$ waiting on $P(IAT[b].w)$ ever suffers an infinite wait. Semaphore $IAT[b].w$ synchronizes the calls of $free(x)$ and $member(x)$. Semaphore g synchronizes the calls of $new(s)$ and $free(x)$. On the other hand, $member(x)$ may proceed in parallel with another $member(x)$ (only a small critical region guarded by $lAT[b].r$

is common to many $member(x)$'s), as well as with another $member(y)$. Similarly, there is no critical region for $new(s)$ and $member(x)$, and for $free(x)$ and $member(y)$. In this way we fulfilled all conditions displayed in Table l. It must be observed that in the readers-writers problem a proper implementation has to guarantee the priority of writers over readers. The collision problem between $free(x)$ and $member(x)$ is somewhat different. In fact, the user wanting to release a frame, while simultaneously also trying to access the same frame an infinite number of times, generates a problem which is not a run-time problem. In this case, the user program is incorrect (as in the case of infinite loop). Therefore, assuming that the primitive statements are indivisible, we can significantly simplify the previous procedures as follows:

```
function member(x);              procedure Free(x);   procedure new(s);
begin                            begin                begin
  P(IAT[b].r);                     P(g);                P(g);
  IAT[b].m := IAT[b].m + 1;        P(IAT[b].r)]         .
  if IAT[b].m = 1                  P(IAT[b].w);         .
  then                             ...                  .
    P(IAT[b].w)                    V(IAT[b].w);         V(g)
  fi;                              V(IAT[b ].r);      end new
  V(IAT[b].r); ...                 V(g)
  lAT[b ].m := lAT[b ].m - 1;    end Free
  if IAT[b].m = 0
  then
    V(IAT[b].w)
  fi
end member
```

This solution does not require the semaphore $IAT[b].r$. When the semaphore $IAT[b].w$ is accessed by $free(x)$ and many $member(x)$'s, the correctness directly follows from indivisibility of primitive operations on $IAT[b].m$ and from the assumption that the value $IAT[b].m$ becomes 0 after a finite period of time. However, we would prefer the first solution. In fact, we also want to prevent incorrectness actions undertaken by the programmer, as we have already mentioned above.

7 Conclusion

We have presented a new algorithmic approach to the storage management problem for run-time systems, in line with the programmed deallocation strategy, without dangling reference. The proposed data structure with its related algorithms turns out to be completely secure, and therefore immediately usable in every run-time system for dynamie languages, even for parallel computation. The specific problem of finding a good free frames structure to perform searching in constant time is still open. Recent results [6] which guarantee constant time searching are, in fact, relevant only for completely static structures, while the need to manipulate free frames in a run-time system is typically dynamic.

References

[1] *Reference Manual for the ADA Programming Language*, United States Department of Defense, 1980.

[2] D.M. Berry and A. Sorkin, *Time required for garbage collection in retention block-structures languages*, Internat. J. Comput. Inform. Sci. 7 (4) (1978).

[3] J. Cohen, *Garbage collection of linked data structures*, ACM Comput. Surv. 13 (3) (1981).

[4] P.J. Courtois, F. Heynemans and D.L. Parnas, *Concurrent control with 'readers' and 'writers'*, Comm. ACM 14 (10) (1971).

[5] E.F. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten and E.F.M. Steffens, *On-the-f1y garbage collection: An exercise in cooperation*, in: Lecture Notes in Comput. Sci. 46 (Springer, New York, 1976).

[6] M.L. Fredman, J. Komlos and E. Szemeredi, *Storing a sparse table with 0(1) worst case access time*, Proc. Foundations of Computer Science Conf., 1982.

[7] P. Brinch Hansen, *EDIsoN - a muItiprocessor language*, Software Practice and Experience 11 (N4) (1982).

[8] D.E. Knuth, *The Art of Computer Programming* Vol. I: Fundamental AIgorithms (Addison-Wesley, Reading, MA, 1973).

[9] G. Lindstrom and M.L. Soffa, *Referencing and retention in block-structured coroutines*, ACM TOPLAS 3 (3) (1981).

[10] B. Liskov et al., *CLU Reference Manual*, Lecture Notes on Comput. Sci. 114 (Springer, Berlin, 1981).

[11] *LOGLAN 82: Programming Language*, Inst. of Informatics, Warsaw University, 1982.

[12] A. Newell, I*Information Processing Language V Manual* (Prentice-Hall, Englewood Cliffs, NJ, 1961).

[13] J.L. Steele, *MuItiprocessing compactifying garbage collection*, Comm. ACM 18 (9) (1975).

[14] N. Wirth, *The programming language PASCAL*, Acta Infor-matica N1 (1971).