Storage Allocation in Typed Languages

Butler W. Lampson

Xerox Corporation Palo Alto Research Center

Appeared in *Proc. 5th Ann. III Conf: Implementation and Design of Algorithmic Languages*, Guidel, France, 1977, pp 315-322.

This version was prepared by OCR of a draft. There may be OCR errors, and there may be other differences from the published version.

Abstract

Several interfaces between a strongly typed language and a storage allocator or deallocator are defined. Using these interfaces, it is possible to program a wide variety of allocation and automatic deallocation strategies: boundary tag allocator, buddy system, reference counting, trace-and-mark garbage collection and many others. Provided the allocator and deallocator have certain well-defined properties, the type-safeness of the language can be guaranteed.

1. Introduction

Techniques for storage allocation and deallocation have long been known and practiced in untyped languages such as assembly language or BCPL [Richards 1969]. In such languages the machine addresses in which storage allocators traffic have no special status, and hence can be subjected to the arbitrary computations which may be needed by a particular allocation method. For example:

> If blocks of a single size are being allocated, it may be convenient for the allocator to view them as elements of an array, but it must convert an array index into the machine address of that array element when allocating a block, and convert a machine address back to an array index when deallocating.
>
> If the size of each block is known (but all blocks are not of the same size), and blocks are contiguous in storage, then the size of a block (an integer) can be added to its address (a pointer) to obtain the address of the next block.
>
> In the buddy system [Knuth 1968, p 442], storage blocks are powers of two in size, and belong to a binary tree which is defined by the binary structure of their addresses: a block at level i has an address $a = 0 \bmod 2^i$, and can be decomposed into two blocks at level i-1 with addresses a and $a+2^{i-1}$. Two free blocks may thus be coalesced into a larger one if their addresses differ in the least significant one bit.

On the other hand, in an untyped language automatic deallocation of storage is not possible, since all automatic deallocation schemes depend on being able to identify the

pointers, either statically in garbage collection schemes, or when assignments are done in reference count schemes.

In a strictly typed language like Algol 68 [van Wijngaarden 1976] or Pascal [Wirth 1971], the situation is entirely different. There is no problem with garbage collection or reference counting, since the type of every datum is known and hence pointers can easily be identified, either statically or at assignment time (it is necessary, however, to initialize all pointers). On the other hand, a storage allocator cannot treat pointers as integers, bit strings or whatever, since that would violate the strict typing.

One might ask whether the type system could be extended to accommodate the kinds of manipulations required by an allocator. It seems unlikely that this will be feasible in the foreseeable future, for the following reason. The type system can be thought of as a rather simple-minded automatic program verifier, which proves a certain property of the program: namely, that a variable of type *T* always has a value of type *T*. Since this verifier is required to quickly and reliably produce a proof of correctness (with respect to this property) for *any* program which obeys the rules of the language, it cannot depend on any deep reasoning. Now the correctness of a storage allocator usually depends on arguments of some delicacy. In fact, automatic verification of the correctness of, say, a coalescing boundary tag allocator [Knuth 1968, p 442], is a nontrivial task for a state-of-the-art verifier [Wegbreit 1977]. It is certainly far beyond the capabilities of a verifier built into a compiler.

As a consequence of these facts, most strongly typed languages avoid the issue of programmed storage allocation entirely. Some fixed set of built-in facilities is provided by the language:

last-in first-out allocation in Algol 60 and most subsequent languages;

built-in New and Free procedures in Pascal, with no automatic deallocation, and chaos if a variable is referenced after being freed;

built-in allocation and garbage collection in Algol 68.

There are two difficulties with this approach. Firstly, the built-in facilities may be insufficiently complete, or they may be intolerably expensive for a particular application.

Algol 60 and Pascal illustrate the former point. In Algol 60 a variable cannot survive the lifetime of the block in which it is created. In Pascal full generality is present, but the programmer is entirely responsible for deallocation, with two unfortunate consequences: if he frees storage too slowly, space is wasted; if he frees it too fast, variables will unpredictably overlap, with uncertain but disastrous consequences for the execution of the program.

Algol 68 and Simula illustrate the latter point. The cost of garbage collection is uniformly imposed on all users of the heap. If the garbage collection is batched, as it normally is, any program which must meet real time demands will probably find it intolerable. If it is incremental, the cost of *every* pointer assignment will be significantly increased. There is no way for the programmer to express any special knowledge he may have about the properties of the storage he uses (e.g., some

storage will never be freed, some may be freed in large batches at the end of certain phases of computing, some is referenced only from a certain small set of places). By paying these costs, of course, the programmer gets a powerful mechanism with a time and space cost which is likely to be a reasonable percentage of his total costs, and with fairly predictable properties.

Second, the built-in facilities may be quite complex: this is certainly true for most trace-and-mark garbage collectors. If reliability of the program is important, it may be significantly affected by the reliability of the allocation mechanism. A standard method for improving the reliability of a program is to prove its correctness, preferably with the help of a program which can at least attempt to check the correctness of the proof. This method usually depends on expressing the program in a suitable language, such as Pascal or Euclid [Lampson et al. 1977], for which a formal semantics is available [Hoare and Wirth 1973, London, et al. 1977], If the storage allocation facilities are concealed in the implementation, however, and written in some lower level language, this method cannot be applied.

Needless to say, this paper does not contain any comprehensive solution to these problems. What it does contain is some ideas for programming language design (or perhaps for programming conventions) which:

allow storage allocation primitives to be implemented in a typed language, with certain carefully chosen loopholes whose properties can be formally defined;

state certain assertions about the behavior of these primitives, which can in principle be proved from the text of the implementation;

ensure that if these assertions hold, then the variables allocated by the primitives behave as variables are supposed to behave.

2. Definitions

In this section we introduce the model of storage allocation we are going to use, and define some terms. We use more or less the terminology and syntax of Pascal for describing interfaces to the language, and for examples.

We divide the text of the program into three parts:

the *allocator,* which has the interface procedures Allocate and Deallocate described in the next section;

the *client,* which is the program as written by the ordinary programmer, using the interface procedures New and Free to acquire and release storage;

the *mediator,* which talks to the client on one side and the allocator on the other, and is responsible for most of the breaches of the type system. The mediator is probably part of the implementation and very likely cannot actually be written in the typed language. Its job is to *bless* raw storage as the representation of typed variables.

Our model of storage is a linear, uniformly addressed memory consisting of a sequence of components called *storage units.* Each storage unit has an *address* which is a value in some subrange of the non-negative integers; an obvious ordering is induced on storage units by their addresses. On the IBM 370, for example, a storage unit would probably be an 8-bit byte, on the CDC 6600 a 60-bit word; the size of a storage unit and its internal structure will not concern us further, Storage is allocated in *blocks* of contiguous storage units; a block is characterized by its address, which is the address of the first storage unit in the block, and its *length,* which is the number of storage units in the block. Two blocks *overlap* if they have any storage units in common. We shall ignore the possible need for *alignment* of storage blocks so that they have addresses which are divisible by various powers of 2; alignment can be handled by supplying additional parameters to the allocator, but the details are tiresome.

Following Pascal, we assume that storage is allocated to *variables,* and for simplicity we assume that each variable is represented in a single block of storage; more complex representations are straightforward, but clutter the exposition. A variable *v* has a well-defined *size* given by the standard function size(*v*); this is the smallest number of storage units required to hold the representation of *v.* For the purpose of the exposition, we assume that v also has a definite *address* given by the standard function address(*v*); this is the address of the first of the size(*v*) storage units holding its representation. These functions are not necessarily available to the client. Note that we make no commitment to the representation of pointers.

The allocator has some region of storage which it *controls,* defined by a Boolean function AllocatorControls(*a*: address). We are committed to our storage model only for the storage controlled by the allocator. Additional storage may be used to hold code and declared variables; we are not concerned with this storage.

The essential property of variables from our viewpoint is *non-overlap:* if *x* and *y* are different variables, then an assignment to *x* cannot change the value of *y.* It is this property which a storage allocator must help to preserve. We are only interested by *dynamic* variables in the sense of Pascal (i.e., variables created by New, rather than by a declaration), and dynamic variables can be accessed only through pointers. Two dynamic variables $p\uparrow$ and $q\uparrow$ are *different* if $p \neq q$. The treatment of overlap is complicated by the fact that a variable may have components which themselves are variables, but this detail is inessential for our purposes and will be ignored. For a more careful treatment, see [Lampson et al. 1977, section 7]; also, note that in Pascal or Euclid $q\uparrow$ can never be a component of $p\uparrow$ although *x* can be a component of $p\uparrow$ if *x* is a formal parameter.

3. Interfaces

Following the ideas behind the *zone* construction in Euclid, we will define the relationship between an allocator and its client (which is the rest of the program) as follows. The client performs two basic operations which result in the creation and destruction of (dynamic) variables:

New(**var** *p:* $\uparrow T$), which creates a new variable of type *T* and stores a pointer to it in *p.*

Free(**var** *p:* ↑*T*), which destroys the variable pointed to by *p* and sets *p* to nil.

A variable *p*↑ is said to *exist* in the interval between the call of New which creates it, and the call of Free which destroys it. Existence of the variable is independent of the value of *p* or any other pointer (but see the discussion of automatic deallocation below).

These operations are calls on the mediator, which in turn implements them by calling two procedures

Allocate(*n:* **integer**, **var** *a:* **address**)

Deallocate(*n:* **integer**, **var** *a:* **address**)

which are part of the allocator. Allocate is supposed to return in *a* the machine address of a block of at least *n* storage units, within which the new variable will presumably be represented. Deallocate is supposed to receive the information that the block of *n* storage units at *a* is available for further use. A block of storage, and each of the storage units it contains, is said to be *allocated* in the interval between the call of Allocate which supplies it, and the call of Deallocate by which it is returned to the allocator. The allocator's responsibilities are as follows:

Any allocated block must be entirely contained in storage controlled by the allocator.

Two allocated blocks must not overlap.

The mediator has responsibilities to both the allocator and the client. Its responsibility to the client has already been stated: two dynamic variables which exist at the same time must not overlap. Its responsibilities to the allocator may be stated as follows:

The mediator and its client must not read or write any storage unit which is not allocated.

For every call of Deallocate(*n*, *a*) there must be a matching previous call of Allocate(*n′*, *a′*) such that $n=n'$ and $a=a'_{\text{just after Allocate returns}}$ (i.e., only storage which has been allocated can be freed, and only once).

The client in turn has responsibilities to the mediator:

A storage unit must be read or written only if it is part of the representation of an existing variable. Except for dangling references, this property is presumably guaranteed by the language, by virtue of its enforcement of the type system.

For every call of Free(*p*) there must be a previous call of New(*p′*) such that $p=p'_{\text{just after New returns}}$ (i.e., only variables which have been created can be destroyed, and only once).

These properties of the mediator's interfaces to the client and the allocator can be ensured by the following assumptions about how the mediator works:

A call of New(*p*) gives rise to a call of Allocate(*n*, *a*) with $n \geq \text{size}(p\uparrow)$. The address of the newly created variable *p*↑ will be *a.* The storage block returned by this call of

Allocate will not be used for any other purpose. Obviously more flexibility is possible, but would complicate the exposition.

Deallocate is called only as a result of a call of Free(*p*), and with the parameters *n*=size(*p*↑) and *a*=address(*p*↑).

Provided these assumptions are satisfied, and the client and the allocator meet their responsibilities, it is safe for the mediator to take the storage blocks supplied by the allocator and bless them as variables. We do not attempt a proof of this statement, but content ourselves with a few observations.

Such a proof could not be given without a more formal model of the activities of the client. This model would naturally be stated by reducing the client's use of dynamic variables to reads and writes of storage units. The issues raised in constructing such a model would be similar to the issues which arise in attempting to prove the correctness of a compiler or interpreter.

The desired behavior of the allocator can be formally specified with the help of an auxiliary Boolean array *allocated,* which parallels the storage controlled by the allocator: *allocated*(*i*) is True iff storage unit *i* is allocated.

Nothing we have said is affected by the existence of more than one allocator. It is entirely a matter of performance which allocator is called to supply a particular piece of storage; as long as all the allocators satisfy the requirements stated above, and storage returned to a Deallocate procedure is supplied by the corresponding Allocate procedure, the correctness of the program cannot be affected.

4. Dependency

Automatic deallocation of storage has much more far-reaching implications for the interaction between the language and the allocator than does automatic allocation: it is better to give than to receive (or easier anyway). The reason is simple: automatic deallocation is a great deal more automatic. Storage is allocated only on demand, but when we speak of automatic deallocation we usually mean that the storage should be freed when it is no longer needed. Determining when this is the case requires some kind of global knowledge about the use being made of the storage by the program. In the case of variables which are associated with procedure instantiations this is straightforward, and gives rise to the familiar stack allocation of Algol 60 and its derivatives. In the fully general setting which is the usual generalization, however, the assumption is that storage is needed until it can no longer be *reached* by starting from a pointer in some distinguished set of pointers, and following pointers to an arbitrary depth. This definition is usually combined with an absence of restrictions on copying of pointers, so that a pointer to any piece of storage can exist more or less anywhere in the data structures of the program. In this situation only examination of all the pointers will suffice to determine whether a variable is no longer needed.

Other definitions of “no longer needed” are of course possible. For example, if an explicit Free operation is available, a variable is assumed to be no longer needed when it is explicitly freed. Usually there is no way to check that the variable will not be referenced

again, however, and we get the familiar dangling reference problem. This provides a second motivation for more automatic deallocation: in addition to affording a significant programming convenience, it also removes a significant source of obscure errors.

There are two basic approaches for keeping track of all the pointers: batch and incremental. The batch approach is trace-and-mark garbage collection [Fenichel 1971], and the incremental approach is reference counting. Each has fairly obvious advantages and disadvantages, of which we remark on only two: reference counting cannot free circular or self-referencing data structures, and its effectiveness depends on the ratio of pointer assignments to storage deallocation. Both schemes require proper initialization of all pointers.

Mixed approaches are also possible, in which certain pointers are kept track of with reference counts, and others by tracing [Deutsch and Bobrow 1976]. Presumably the hope is to reference count the pointers which are infrequently assigned to, and trace those which can be easily found; In Lisp, for example, it is attractive to reference count pointers stored in list structure, and trace pointers stored in the local variables of functions and in global variables. When these two categories do not exhaust all the pointers, difficult decisions must be made. Deutsch and Bobrow also point out that a variety of techniques can be used to implement reference counting.

In this section we will study some facilities which permit programming of fairly general automatic deallocation strategies within a typed language. The idea is the same as before: to reduce the built-in facilities to a minimum, and carefully characterize the responsibilities of the language and the deallocator to each other. The greater difficulty of the problem, however, results in considerably more machinery in the language.

We introduce a partially ordered set of *dependency* values, and allow one of these values to be attached to a type. Thus, if δ and ε are dependency values and T is a type, $\delta\ T$ and $\varepsilon\ T$ are also types. We speak of a variable of type $\delta\ T$ as a δ variable, and of a pointer to a δ variable as a δ pointer. The idea of dependency is that a δ variable is "dependent on" or "reachable from" an ε variable if and only if $\delta \leq \varepsilon$. From the language's point of view, of course, the meaning of dependency is up to the deallocator; the language simply guarantees certain properties of pointers to dependent variables. From the deallocator's point of view, the properties guaranteed by the language must be strong enough to permit useful conclusions to be drawn about the behavior of the pointers.

The properties guaranteed by the language are:

A δ pointer may not be a component of an ε variable unless $\delta \leq \varepsilon$.

Dependent pointers are always initialized to nil when they are created.

This scheme has been abstracted from a number of specific examples of dependency relations. Here are some of them:

The dependency values may be lifetimes of the variables: $\delta < \varepsilon$ implies that δ has a *longer* lifetime than ε. This may seem backwards at first, but it is correct. In the sense we are interested in, longer-lived things are more dependent, since pointers from short-lived to long-lived objects cause no trouble; it is pointers in the reverse direction which may

result in dangling references. Note also that the return links holding a stack of procedure activations together point from the called procedures to the calling ones, so that the long-lived activations are dependent on the short-lived ones for their chance to regain control

The dependency values might be drawn from the set {not traceable, traceable}, or {**ntr**, **tr**} for short, with **ntr** < **tr**. This captures the idea that a pointer to a traceable variable (i.e., one which can be reached in the tracing phase of a trace-and-mark garbage collector) must come from another traceable variable, and not from untraceable variables.

A generalization of the previous example is to have several kinds of traceability, corresponding perhaps to different classes of variables which have significantly different properties. For two kinds the dependency values would be {**ntr**, $\mathbf{tr_1}$, $\mathbf{tr_2}$, $\mathbf{tr_1} \oplus \mathbf{tr_2}$} , with **ntr** < $\mathbf{tr_1}$ < $\mathbf{tr_1} \oplus \mathbf{tr_2}$ and **ntr** < $\mathbf{tr_2}$ < $\mathbf{tr_1} \oplus \mathbf{tr_2}$

As a consequence of the guaranteed properties, if we want to find all the pointers to δ variables, it suffices to examine all ε variables for $\varepsilon \geq \delta$. From this observation we can draw several conclusions of value to a deallocator:

> If all such ε variables are gone, no pointers to δ variables exist. This is the situation we would like to have for references to local variables in any language which stores local variables on the stack. It may also be of value if a number of variables need to be created within a particular scope (e.g., one phase of a compiler) and they can all be destroyed on exit from that scope.
>
> Define base(δ) as the smallest value greater than or equal to δ (of course it might not be unique, in which case an obvious generalization of what follows is required). Then if we start at the base(δ) variables and recursively trace all the pointers $\geq \delta$, all the δ variables will be found. To make this useful, of course, the language must also provide a way to enumerate all the pointers in a variable; we omit the possibly messy details of this enumerator. Note that procedure activations must also be regarded as variables for this purpose.
>
> In a more general setting, **tr** might be adjoined to a set of lifetimes, so that some pointers are managed by tracing and others by lifetime. For example, suppose we start with dependency values $\delta < \varepsilon < \varphi$. We add four new values, $\delta_{\mathbf{tr}} < \varepsilon_{\mathbf{tr}} < < \varphi_{\mathbf{tr}} <$ **tr**, and the relations $\delta_{\mathbf{tr}} < \delta$ etc, which gives us the following picture:
>
> $$\begin{array}{ccccc} \delta & < & \varepsilon & < & \varphi \\ \vee & & \vee & & \vee \\ \delta_{\mathbf{tr}} & < & \varepsilon_{\mathbf{tr}} & < & \varphi_{\mathbf{tr}} < \mathbf{tr} \end{array}$$
>
> Now pointers to $\varepsilon_{\mathbf{tr}}$ variables can be stored in $\varepsilon_{\mathbf{tr}}$, ε and **tr** variables (as well as in the φ variables). Provided deallocation is not done during the lifetime of ε variables, they don't have to be traced.

5. Incremental deallocation

To write an incremental deallocator, a different, indeed an almost orthogonal, language feature is needed. This is the ability to specify an *assignment procedure* for a type (or at least for a pointer type) which is invoked whenever any assignment to a variable of that type is clone, including the implicit assignment of nil which we assume is done just

before the pointer is destroyed. If AP_T is the assignment procedure for type *T,* then the sequence

$p : T$;
$p := e$;

results in a call of $AP_T(p, e)$ immediately before the assignment is done. We avoid the view that the assignment itself should be done by AP_T, since it is irrelevant to our point.

For pure reference counting deallocation, this is all that is required, at least in principle. There is no problem in finding a place to keep the reference count, since a different assignment procedure is called for each type, and hence a different reference count field can be used for each type. In practice this might not be very satisfactory if a large number of different types are reference-counted. In addition, there must be a guarantee that only the assignment procedure modifies the reference count. Both of these problems can perhaps be solved by some generalization of the Simula sub-class idea. Further discussion of this problem falls outside the scope of this paper; for examples, see [Mitchell and Wegbreit 1977]. Alternatively, the transaction scheme of Deutsch and Bobrow could be used.

The major drawback of reference counting (aside from its inability to free self-referential structures, which is usually a minor problem in practice) is that it may cause a large amount of unnecessary work. A procedure which runs down a list looking for the biggest element, for example, will increment and decrement the reference count for each element of the list as a result of the assignments to the single pointer variable being used for the traversal (a good optimizing compiler which expanded the assignment procedure in-line might be able to improve this situation, but such compilers are more often encountered in papers than in real life). It was this observation that motivated the mixed strategy described in [Deutsch and Bobrow 1976]. A slight extension to our dependency scheme can accommodate such a mixed strategy: namely, to allow a *set* of assignment procedures to be attached to a type, with the choice of a procedure from the set depending on the dependency value of the variable which contains the target variable of the assignment. Consider the dependency set (**ntr**, **tr**}. We reference-count **ntr** types, but only if their pointers are stored in **ntr** variables; a **ntr** type stored in a **tr** variable will have a null assignment procedure. When a reference count goes to zero, and no pointers to the variable can be found in **tr** variables, we know that no pointers exist.

Disclaimer

None of the schemes proposed in this paper has been implemented.

Acknowledgements

I am indebted to Jim Horning and Jim Mitchell for enlightening discussions on interfaces, to Peter Deutsch for the same help with deallocation methods, and to Ben Wegbreit for a better understanding of the verification issues.

References

[Deutsch and Bobrow 1976] L. P. Deutsch and D. G. Bobrow, "An Efficient, Incremental, Automatic Garbage Collector," *Comm. ACM* **19**, 9.

[Fenichel 1971] R. R. Fenichel, "List Tracing in Systems Allowing Multiple Cell-Types," *Comm. ACM* **14**, 8.

[Hoare and Wirth 1973] C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal," *Acta Informatica* **2***,* pp 335-355.

[Knuth 1968] D. E. Knuth, *Fundamental Algorithms,* Addison-Wesley, Reading, Mass.

[Lampson, et al. 1977] B. W. Lampson, et al., "Report on the Programming Language Euclid," *SIGPLAN Notices* **12**, 2.

[London, et al. 1977] R. L. London, et al., "Proof Rules for the Programming Language Euclid," in preparation.

[Mitchell and Wegbreit 1977] J. G. Mitchell and B. Wegbreit, "Schemes: A High-Level Data Structuring Concept," *Current Trends in Programming Methodology,* vol. 4, Prentice-Hall, New York.

[Richards 1969] M. Richards, "BCPL: A Tool for Compiler Writing and Structured Programming," *AFIPS Conf. Proc.* **34**.

[van Wijngaarden, et al. 1976] A. van Wijngaarden (ed.), et al., *Revised Report on the Algorithmic Language Algol 68,* Springer-Verlag, Berlin, New York.

[Wegbreit 1977] Personal communication.

[Wirth 1971] N. Wirth, "The Programming Language Pascal," *Acta Informatica* **1**, pp. 35-63.