Memory Allocation in `Aldor`

A D Kennedy
Department of Physics & Astronomy
The University of Edinburgh

June 14, 2001

Abstract

We propose that the mode of memory allocation in `Aldor` should be another "orthogonal" property to be associated with objects. We discuss the reasons for this in the context of large-scale numerical computations using `Aldor` (the *Paraldor* project), and we consider what changes might be required in the compiler.

1 Memory Allocation Strategy

1.1 Introduction

We shall consider the issue of memory allocation in the context of the Paraldor project, which is an attempt to implement large-scale numerical computations for massively parallel machines in `Aldor`. Unlike more conventional applications of `Aldor` in computer algebra we have to deal with large objects whose extents are known at compile time, and performance issues are critical.

1.2 Stacks or heaps?

One of the basic precepts of `Aldor` is that the user should write code to implement an algorithm in terms of categories describing the (minimal) structure necessary for its correct behaviour. A by-product of this is that the user cannot and should not know about the implementation of the domains which implement these categories. A corollary of this is that memory management cannot be left to user control.

There might be cases where the user might give hints to the compiler in an manner analogous to the `inline` hint to the optimiser, but there is some evidence (or folklore?) that even the simple `register` memory allocation hint in `C` is rarely of any use. At some level within the hierarchy of domains implementing the categories the type of memory management should be specified, and this should be viewed as an attribute of the `Rep` of the object, and methods will be provided to create and destroy these objects. It is usually (perhaps always) obvious when an object needs to be created, but the implementation of the domain has no way of knowing when it needs to be destroyed.

This requirement for a "fully automatic" memory allocation mechanism naturally points us towards a heap allocation strategy. However, although a heap storage allocation strategy is the most general way of allocating memory for objects, it has certain disadvantages. The principal of these are *fragmentation* and *memory leaks.*

- *Fragmentation.* Heaps are wonderful when we are allocating lots of small uniform sized objects. If the objects are small then we do not have to be niggardly about having a few more of them around at any given time, and if they are uniform then new objects can immediately fill the gaps left by their fallen comrades. In our application the objects are large, to the extent that a few tens or hundreds of them will fill the whole of memory, and come in various sizes. In fact, the object do not usually come in very many different sizes, so it is possible to think in terms of allocating different sized ones in different memory pools, but a naïve heap would rapidly get indigestion when we allocate our huge vector objects and tiny scalar objects in an interwoven fashion. Heap compaction is possible, but most undesirable in view of its unnecessary cost.
- *Memory leaks.* The fact that the implementation of a domain can provide a `destroy!` method does not mean that it will ever be called, and in languages like `C++` where it is the user's responsibility to ensure that heap-allocated objects are freed programs are often replete with memory leaks when they forget to do so (or with bugs when they do so too early!). The canonical solution to this, familiar from the prehistory of `Lisp`, is garbage collection.
- *Garbage collection.* There are several varieties of garbage collectors available on the market. They can be synchronous (as in `Aldor` at present) or asynchronous (as in `Java`, where the garbage collector and finanliser run in their own threads). They

> can be scavangers or use reference counting. Reference counting is quite an attractive option for us as our objects are large, and usually only have a single reference to them. On the other hand the more popular scavenging approach is an anathema on a data parallel machine where all the processors have to wait when a single one decides it is time to go hunting. One might imagine that on a data parallel machine all the processors would garbage collect at the same time, but this behaviour is spoiled if one node uses a little extra memory (for some I/O operation perhaps), or if one node's data just happens to look more like a dirty pointer than another's.

In any case, the main objection to heaps and garbage collection is that they waste time and space and are not necessary in applications where we do not return objects of variable or unknown size. For our purposes stacks are adequate and, as a rule, they are faster and smaller.

An important distinction should be made at this point, as a general stack should not be confused with `auto`matically allocated variables in `C`, which are allocated in a routine's *frame*. Every thread has a *call frame stack*, onto which a frame is pushed whenever a routine (or more generally a block) is entered, and which is popped when it is exited. There is no reason why this particular stack has to be used for allocating our large objects, indeed we often want a routine to be able to return a new object, and this would be a futile thing to do if the object were to be allocated on the call frame.

2 Memory Spaces

We propose the following approach to memory allocation in Aldor. The user should be able to define a collection of *memory spaces*, each of which may have a maximum size, be associated with a level of the hardware memory hierarchy, have a specified latency, have different allocation strategies, etc.

Furthermore, each memory space should have an attribute which, for want of a better name, we may call `direct` or `indirect`. It is not allowed to construct a pointer to an object in a `direct` memory space, whereas this is permitted in an `indirect` memory space. A `direct` memory space may therefore contain simple values, arrays, records, unions, etc., but not linked lists or trees.

When a new object is allocated memory, this memory should be charged to a *memory space*. The compiler should be informed which memory space is to be used and be given a bound on the amount of memory required for this allocation. Perhaps `new!` and `dispose!` methods should be required which handle the construction and destruction of objects in the style of `C++`.

In general the task of calling the `dispose!` method for an `indirect` memory space is not the responsibility of the compiler: for a heap a garbage collector might be provided as part of the library at it could generate calls to `dispose!` as appropriate. The compiler might provide such a garbage collector for a default memory space as at present, but in general the implementor of the memory space would provide a custom garbage collector.

For a `direct` memory space the compiler should generate calls to `dispose!` for an object when it becomes inaccessible. This can happen either when the relevant frame is popped off the call frame stack and the object is not explicitly used as a return value, or when a compiler-generated temporary is no longer required. For variables allocated in a stack-based memory space the compiler must call `destroy!` in the reverse order to that in which the objects were created. Since there can be no aliasing in a `direct` memory space this strategy should be safe.

The key point is that we *require* the compiler to make visible (by calling `dispose!`) its temporary allocations. The user could manage his own memory allocation scheme for his own explicitly declared variables and constants, but unless he is prepared to forego the use of expressions[1] he cannot control the allocation of temporaries. Since we only require the compiler to do this for the new class of `direct` memory spaces it will not change existing semantics. The most straightforward algorithm for generating code from an expression tree generates stack-allocated temporaries automatically. If the compiler is smarter, and does fancy things with common expression elimination, lifetime analysis, etc., then more interesting space versus time tradeoffs can be considered. Even the simplest approach should be adequate for most purposes.

The use of memory spaces could be made more sophisticated if desired. For instance, the compiler could choose the memory space in which to allocate an object on the basis of its frequency of reference (using the same heuristics as for `inline`ing) and its size. For this to be possible at compile time the compiler would need to be able to estimate the object's size, and we propose that this should be done by requiring

[1]One can write assembler code in any language!

each `new!` method to declare a bound on the size of the object. If an `external` routine is called which allocates memory, then it should provide as part of its interface specification a machine-readable bound on its memory use (for each memory space). If all `builtin` and `external` memory-allocating routines do this then bounds for any `new!` method can be computed automatically.

The trick is to choose a form for the bounds which is simple enough that the compiler can generate fast run time code to estimate the object's size, but good enough that the bounds are not unrealistic. We suggest that the bounds should be piecewise linear functions of a set of scaling parameters (e.g., volume, number of widgets), which should be adequate in most practical cases.

Acknowledgements

This document is based on many discussions with Tom Ashby Peter Boyle, Martin Dunstan, Robert Edwards, Balínt Joó, and Stephen Watt.