A STUDY IN THE INTEGRATION OF COMPUTER ALGEBRA SYSTEMS:

MEMORY MANAGEMENT IN A MAPLE–ALDOR ENVIRONMENT

STEPHEN M. WATT
ONTARIO RESEARCH CENTER FOR COMPUTER ALGEBRA
UNIVERSITY OF WESTERN ONTARIO
LONDON CANADA, N6A 5B7

1 Introduction

Many approaches have been followed to allow several computer algebra systems to work together, including MP [1], OpenMath [2], and the IAMC framework [5]. All of these propose mechanisms for loosely coupled systems sharing data by some communications protocol.

For certain types of computation, loosely coupled systems are inappropriate. Examples of this includes situations where

- the problem treated by a foreign system would have its computation costs dominated by communication costs
- the problem treated by a foreign system requires access to some *a priori* undetermined subset of a relatively large data structure or database
- common persistent data structures are maintained and manipulated by operations in each of the computer algebra systems

We have undertaken to study the problems which arise in the tight coupling of computer algebra systems. By this we mean having multiple computer algebra systems share the same address space so that objects created by one system can be passed passed by reference in calls to another system.

As an interesting practical problem, we have explored the case of using Aldor code from within the Maple system, allowing Aldor and Maple programs to share data and call each other. This allows Aldor to be used as a compiled extension language for Maple, and consequently provides Maple an efficient generic programming facility. This required creating an *Enhanced External Function* interface for Maple [7], which has been incorporated as part of the standard release of Maple since Maple 7.

In this paper we describe the low-level memory management issues arising from this integration. The natural interface between Aldor domains and Maple modules shall be reported elsewhere.

From *Proc. Int. Congress Math. Software*, World Scientific (2002)

Memory management is a well-studied subject (see, e.g. the ISMM proceedings [4] [6]), but there has been relatively little work on the integration of existing, mature memory managers. Of interest is the work on CMM [3], the "customizable memory manager," which can be useful in porting multiple C++ applications to a common setting. The work reported here focuses on the integration of mark-and-sweep collectors in a language-independent setting.

2 Systems Issues

The combined Maple-Aldor environment consists of a Maple system with dynamically loaded Aldor code. Maple and Aldor each maintain their own memory heap and use their own native formats for programs, big integers, and other data structures. Pointers to Maple objects may be passed to Aldor functions and *vice versa*. Likewise, data structures in the Maple heap may point to objects in the Aldor heap and *vice versa*.

The main problems in this integration relate to the representation of inter-heap pointers and the behaviour cooperative of garbage collection. The current implementations of both Maple and Aldor use a mark-and-sweep garbage collectors, but problems arise because of different assumptions made by the respective systems:

- Maple assumes that all data objects are self-identifying, and that there are a fixed set of possible object types. The memory manager has intimate knowledge of the fields within each of the possible object types, and has exact knowledge of which fields are pointers and which are data. The structure of the Maple memory manager, however, does not allow a constant time test to determine whether an address points to an object in the heap.

- Aldor assumes that arbitrary data structures may be required, and that data is *not* self identifying. This allows easy inclusion of foreign pointers in data objects, but does not allow the memory manager to distinguish pointer from non-pointers within these objects. The Aldor memory manager allows fast, $O(1)$ time test to determine whether a given address points to the beginning of an object in the heap.

  Because the Aldor garbage collector is more general, it does not have exact knowledge of the internal layout of allocated objects, global data or the program stack. It uses a conservative strategy where the targets of all potential pointers are regarded as live. (A potential pointer is a word that, if interpreted as a pointer, would point to a data object.)

It would be completely impractical to re-work the two systems to use a single memory manager for several reasons. First, it would be prohibitively costly. Second, the memory managers are now mature—any significant changes would risk introducing subtle bugs. Third, this would create a solution applicable only to the Maple-Aldor combination, and would not shed any light on the general problem of system integration.

The approach we have taken is to allow both the Maple and Aldor heaps to co-exist and be for the most part independently managed. The problems are how to represent pointers between heaps, and how to coordinate garbage collection.

3 Pointers between Heaps

Since the Aldor memory manager makes no assumption about the content of heap-allocated objects, pointers from Aldor objects to Maple objects are simply stored as words—*i.e.* in the same representation as pointers to Aldor or C objects.

Life is not so simple for including pointers to foreign objects in Maple data: the Maple memory manager has detailed understanding of all the types of objects in Maple so it was necessary to introduce a new type of basic Maple object. We have called these new objects *external object handles*, or *handles* for short.

Handle objects in Maple allow references to objects in external heaps to be returned from external functions to Maple. The handle objects contain three components:

- one word for data, as immediate data or an external pointer
- one word for type, as immediate data or an external pointer
- a "mark event handler" function

The mark event handler allows the referenced object in the external heap to be aware of garbage collection. If this function pointer is non-zero, it is called with data and type words as arguments when the handle object is marked by the garbage collector (see Section 5).

4 Identification of Objects

While the Aldor memory manager is able to determine whether an address points to the beginning of an Aldor object, the corresponding test is not possible with Maple's memory manager. The only way to determine whether

an address points to an object in the Maple heap is to traverse the containing heap segment from the beginning.

To allow garbage collection in the presence of inter-heap pointers, it is necessary to have certain knowledge whether a pointer into the Maple heap is actually pointing at a Maple object. Operations to traverse the Maple heap have therefore been included in the Enhanced External Function interface for this purpose. These operations allow a program to traverse the Maple heap and see the addresses of all objects.

5 Garbage Collection Events

When a reference to a Maple object is passed to an external function, or a Maple object is created by an external function, it is possible that there may be no references to that object from other Maple structures. We call these "captured Maple objects." When such an object is not referenced by other Maple structure, it is susceptible to be garbage collected. Since Aldor functions can create captured Maple objects, it is necessary for a Maple–Aldor environment to ensure that any captured Maple objects are protected from garbage collection. Analogously, captured Aldor objects may arise from calling Maple functions on Aldor objects, and these must also be handled by a Maple–Aldor environment.

Protecting captured objects can be done in any of three ways: through explicit object management, through reference counting, or through proper garbage collection. When there are many objects flowing back and forth across an interface, and user-level library code operates on these objects, explicit management is error-prone. Reference counting does not allow the creation of cyclic data objects, which are not uncommon in a computer algebra setting. We have therefore developed an automatic method to protect captured objects under garbage collection. This method relies on a protocol of "garbage collection events."

There are four garbage collection events which are made known via Maple's new enhanced foreign function interface. These are

1. the start of a garbage collection,
2. the start of the mark phase,
3. the marking of handles to external objects,
4. the end of the garbage collection.

Notification of the start, the mark phase, and end of garbage collection may be requested by registering event listeners. These take functions which should

be called when the appropriate event occurs.

All installed start listeners are called at the beginning of a Maple garbage collection. The mark phase listeners are called when the garbage collector is willing to receive mark requests for Maple objects. The end listener is called at the end of a garbage collection.

Notification of the marking of external object handles is handled on a per object basis, and is achieved by placing a mark event handler in the object itself as described in Section 3. An additional Maple kernel operation is made available to allow the external application to request that its captured Maple objects be retained as live during a garbage collection.

6 Garbage Collection Protocol

The garbage collection events are sufficient to allow the Maple garbage collector and the Aldor garbage collector to work together. By registering appropriate Aldor memory management functions for the the Maple GC events, we are able to establish the following protocol:

1. Whichever heap initiates a garbage collection, the first thing it does is to initiate a garbage collection in the other, so that both heaps garbage collect synchronously.
2. Traverse the Maple heap building information allowing exact constant-time identification of Maple objects.
3. The Maple heap is marked starting from the Maple root set.

   Whenever a handle to an Aldor object is encountered, its mark event handler function is called. This will initiate marking within the Aldor heap.
4. The Aldor heap is marked starting from the Aldor root set.

   The targets of all potential pointers are marked. (A potential pointer is a word which, if interpreted as a pointer, would point to an object.) Potential pointers to Maple objects are identified using the procedures describe in Section 4.
5. Discard the information allowing exact constant-time identification of Maple objects.
6. The Maple and Aldor heaps are swept independently.

This protocol ensures that no live object is collected and permits object cycles to traverse heap boundaries.

7 Conclusions

We see that the successful tight coupling of computer algebra systems involves careful attention to interactions between memory managers. To allow the mark-and-sweep garbage collectors of Aldor and Maple to interact properly, it was necessary to

- introduce the concept of garbage collection events
- introduce the concept of a "handle" object in Maple
- expose a limited set of representation-independent operations on the Maple heap

These provide sufficient functionality to allow the Aldor memory manager to work cooperatively with Maple's memory manager. The additional cost associated with this implementation is an additional pass over the Maple heap to gain exact pointer knowledge. This is comparable in cost to the sweep phase of Maple's garbage collector.

References

1. *MP: A protocol for the efficient exchange of mathematical expressions*, S. Gray, N. Kajler and P. Wang, Proc. ISSAC 94 International Symposium on Symbolic and Algebraic Computation, pp. 330-335, ACM Press 1994.
2. *An OpenMath 1.0 Implementation*, Stéphane Dalmas, Marc Gaëtano, Stephen Watt, Proc. ISSAC 97 International Symposium on Symbolic and Algebraic Computation, pp. 241-248, ACM Press 1997.
3. *A Customisable Memory Management Framework for C++*, G. Attardi, T. Flagella and P. Iglio, Software Practice and Experience, 28(11), 1143-1183, 1998.
4. Proc. ISMM'98 International Symposium on Memory Management, ACM Sigplan Notices 34(3), ACM Press 1999.
5. *Design and Protocol for Internet Accessible Mathematical Computation*, Paul S. Wang, Proc. ISSAC 99 International Symposium on Symbolic and Algebraic Computation, pp. 291-298, ACM Press 1999.
6. Proc. ISMM 2000 International Symposium on Memory Management, ACM Press 2000.
7. *An Enhanced External Function Interface for Maple*, S. Watt, Waterloo Maple internal report, 2001.