J. Symbolic Computation (1996) **21**, 1–20

Memory Management in the PoSSo Solver †

GIUSEPPE ATTARDI AND TITO FLAGELLA

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, I-56125 Pisa, Italy
email: {attardi,tito}@di.unipi.it

(Received 27 October 1994)

A uniform general purpose garbage collector may not always provide optimal performance. Sometimes an algorithm exhibits a predictable pattern of memory usage that could be exploited, delaying as much as possible the intervention of the collector. This requires a collector whose strategy can be customised to the need of an algorithm. We present a dynamic memory management framework which allows such customization, while preserving the convenience of automatic collection in the normal case. The Customisable Memory Management (CMM) organizes memory in multiple heaps, each one encapsulating a particular storage discipline. The default heap for collectable objects uses the technique of mostly copying garbage collection, providing good performance and memory compaction. Customization of the collector is achieved through object orientation by specialising the collector methods for each heap class. We describe how the CMM has been exploited in the implementation of the Buchberger algorithm, by using a special heap for temporary objects created during polynomial reduction. The solution drastically reduces the overall cost of memory allocation in the algorithm.

1. Introduction

We faced the task of developing memory management facilities for a large research project in symbolic algebra: the ESPRIT BRA PoSSo which aims at building a state of the art system for solving systems of polynomial equations. Researchers working on different parts of the system had different requirements on memory management. Someone preferred a copying garbage collector in order to achieve better data locality; others preferred a mark-and-sweep approach because of its efficiency with data of fixed size; still some others claimed they could perform explicit memory management better than with any general purpose algorithm.

In fact, one of the core algorithms of PoSSo, the Buchberger algorithm for computing Gröbner bases (Buchberger 1985), is quite memory intensive and even the best traditional garbage collection techniques (Wilson 1992, Zorn 1993) lead to thrashing where significant amounts of time are spent in garbage collection.

† The research described here has been funded in part by the ESPRIT Basic Research Action, project PoSSo.
Part of this work has been done while the first author was visiting the International Computer Science Institute, Berkeley, California.

0747–7171/96/000000 + 00 $08.00/0

However, an analysis of the algorithm shows that it exhibits a particular pattern of memory usage, which could be exploited to achieve optimal performance. There are precise points, at the end of one step in the algorithm, where all data created during the previous step become irrelevant and can be deallocated in block. By performing some kind of manual allocation within this portion of the algorithm significant improvements in performance have been reported (Faugère 1994).

Nevertheless automatic memory management through garbage collection still has several advantages over manual management since it improves: *safety*, avoiding the risk of deallocating an object too soon; *accuracy*, avoiding the risk of forgetting to deallocate unused memory; *simplicity*, assuming a computational model with unlimited memory; *modularity*, the program does not have to be interspersed with bookkeeping code not related to the application; *burden* on programmers who are relieved from taking care of memory management.

The ideal solution would be to be able to use a garbage collector under normal circumstances but to be able to deviate from its policies when an algorithm requires so.

Unfortunately, traditional collection algorithms assume total control of memory management and it is impossible to customize the collector to the particular needs of an algorithm. Even if the user wanted to manage memory by himself, some form of coordination with the general collector is still necessary. Suppose in fact that a programmer manages by himself an area of memory and that pointers are allowed from within such area to objects external to it. Such objects might still be reachable but the general collector would not be aware of them and might unduly reclaim their space.

For a similar reason, with a traditional collector it is hard to integrate code or libraries which are unaware of garbage collection and use pointers without restrictions, it is impossible to mix code from programming languages with different memory models.

The Customisable Memory Management (CMM) framework achieves the goal of allowing several collector policies to coexist. Users can choose the most appropriate one, ranging from manual management to fully automatic garbage collection, and can also implement their own specialised memory management. The extensibility of the framework is achieved exploiting the object oriented paradigm of C++, thereby maintaining a consistent and simple interface for programmers.

The CMM consists of:

1. a general purpose garbage collector for C++; this collector is called *primary garbage collector* and exploits the technique of Bartlett's mostly copying collector (Bartlett 1989);
2. a programmer interface: the interface for programs which use CMM;
3. a heap programmer interface: a set of facilities used by heap programmers to define specific memory management policies as appropriate for their applications.

Using the CMM a programmer can specify individually, for each object created, which policy to adopt for its storage. The CMM admits the presence of several collectors, each one in charge of its own heap, which coordinate with each other for proper memory management. The heap where an object resides determines the policy used for the object, but to achieve coordination it must be possible for the collector of one heap to look at objects in other heaps.

CMM users can select among a few predefined memory management disciplines, define their own, or customise those provided in the framework exploiting the mechanisms of

inheritance and specialization. The mechanism to implement these alternative policies is the *heap* abstraction. Specific algorithms are used and particular data structures are maintained by each heap to ensure its proper behaviour.

In the rest of the paper, we review the requirements for a customisable memory manager, we present the design and implementation of the CMM. Then we present the CMM interface for both Computer Algebra programmers and heap programmers. We illustrate both interfaces by showing how the CMM is exploited in the PoSSo implementation of the Buchberger algorithm. Finally we present the results of various benchmarks which show the performance of the collector and the benefits of customization.

2. Design Goals and Constraints

In designing the CMM we tried to achieve the following goals:

1 *Algorithm specific customization*: the allocation policy can be customised to suit the particular needs of an algorithm. This is different from other solutions, where the allocation policy is associated to the type of an object (Ellis and Detlefs 1993). For the purpose of our applications, it is necessary to allocate the same type of object sometimes with one policy and sometimes with another. For example, in PoSSo there is only one class of polynomials, but sometimes a polynomial is allocated in a special heap which can be freed quickly once a certain portion of the simplification algorithm is complete; in other cases the lifetime of the polynomial cannot be predicted, so it must be allocated in the general heap.
2 *Multiple logical heaps*: At least two heaps are necessary, one for collectable objects and one for uncollectable objects. However two is not enough: for instance collectable objects containing data which cannot be relocated for some reasons must be handled differently from other objects which are copied by the collector. For this reason the CMM provides multiple logical heaps.
3 *Usability*: Only a minimal burden should be placed on the programmer who uses the collector. The CMM currently required the programmer to supply a traversal method for each class of collectable objects them, a task which might, however, be automated.
4 *Separation of concerns*: Memory management code needs not to be included within algorithms, and it is possible to change the memory policy just by selecting which heap is employed by the algorithm.
5 *Efficiency*: The implementation should be efficient enough to be as good as and possibly better than hand tuned allocation.

We had, moreover, to satisfy these constraints, for the practical applicability of the solution:

1 *Portability*: The solution could not rely on changes to the underlying language or compiler. Therefore the CMM is built as a program library, which can be used with any C++ compiler.
2 *Coexistence*: Code and objects built with the CMM can be exchanged with traditional code and libraries. No restrictions should exist on whether a collected object can point to a non-collected object and viceversa. We must be able to pass collected

objects to programs unaware of garbage collection, allowing them to store such objects in data structures, without special burden on the programmer or risk that the object would be garbage collected. Alternative solutions require the programmer to put an object in an "escape list" before passing it to an external procedure.

The CMM allows customization of the collector and provides a few pre-built variants. One could argue whether a single general collection strategy could fit all the needs. For instance an ephemeral garbage collector ensures that memory is reclaimed quickly. However even an ephemeral garbage collection is not good enough for applications like PoSSo where one must prevent or delay garbage collection as much as possible. An ephemeral garbage collector is useful to reduce the latency of collection, which is essential in interactive applications, but its overall performance is worse than that of other techniques, as we verified experimenting with the ephemeral version of (Boehm and Weiser 1988).

For the vast majority of applications a general purpose strategy is adequate, and the CMM provides a good one by default. Exploiting customization one can use the CMM also in applications that have special or high performance demands.

3. Design

The task of a garbage collector (Wilson 1992) is to distinguish *live objects* from garbage, for instance by tracing them through memory starting from a *root set* (local and global variables, machine registers).

Depending on the kind of information available during the traversal of objects from the root set, a tracing collector can be *conservative* or *type-accurate*.

A *conservative* garbage collector does not require cooperation from the compiler and assumes that anything that *might* be a pointer actually *is* a pointer. In this case an integer (or any other value) is assumed to be a pointer by the collector if it corresponds to an address inside the current heap range: any such value is called an *ambiguous pointer*. A root containing an ambiguous pointer is called an *ambiguous root*. A garbage collector is *type-accurate* when it is able to distinguish which values are genuine pointers to objects.

The main limitations of a purely conservative collector are memory fragmentation in applications dealing with objects of various sizes, which arises from the inability to move objects, and the risk that a significant amount of memory might not be reclaimed in applications with densely populated address spaces of strongly connected objects (Wentworth 1990).

These limitations are avoided in the partially conservative approach proposed by Bartlett (1988) for his *mostly-copying garbage collector*.

The CMM allows customization through the *heap* abstraction. Each heap class implements a different allocation discipline. Here we present the built-in heaps in CMM and the discipline they implement.

3.1. The default heap

The default heap of CMM uses the technique of mostly-copying garbage collection (Bartlett 1988).

The default heap consists of a number of equal size pages, each with its own *space-identifier* (either *From* or *To* in the simplest non-generational version). The *FromSpace* consists of all pages whose identifier is *From*, and similarly for *ToSpace*. The collector

conservatively scans the stack and global variables looking for potential pointers. Objects referenced by ambiguous roots are not copied, while most other live objects are copied. The process is illustrated in Figure 1.

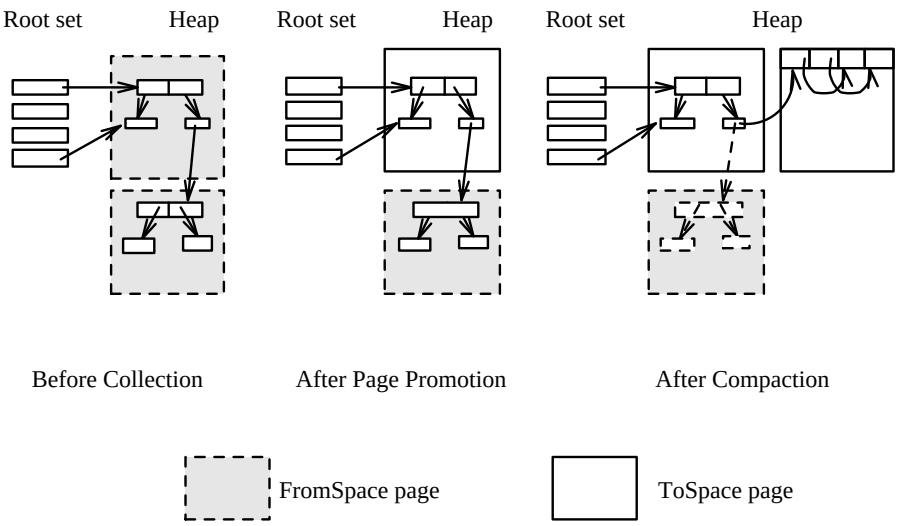

Figure 1. Mostly-copying collector.

If an object is referenced from a root, it must be scavenged to survive collection. Since the object cannot be moved, the whole page to which it belongs is saved. This is done by *promoting* the page into *ToSpace* by simply changing its page space-identifier to *To*. At the end of this promoting phase, all objects belonging to pages in *FromSpace* can be copied and compacted into new pages belonging to *ToSpace*. Root reachable objects are traversed with the help of information provided by the application programmer: the programmer must supply the definition for a member function for each class of objects which traces the internal pointers within objects of that class. Further details on the implementation can be found in Attardi and Flagella (1994b).

3.2. The Uncollected Heap

Besides the copy-collected heap, also the traditional uncollected heap is supported by providing the primitives `malloc` or `new` on uncollected classes. The uncollected heap cannot be eliminated since there are programs and libraries which may use uncollected objects in an unsafe way for the collector (Ellis and Detlefs 1993), and there are objects that can't be relocated. However, we allow objects in the uncollected heap to point to objects in the collected heap and viceversa.

3.3. User Collected Heaps

Our goal is to allow users to build their own heaps with specific allocation strategies for their applications.

We must however fulfill some essential requirements for the solution to be consistent and practical:

Allow pointers across heaps: Restricting the range of pointers is difficult and inconvenient.

Transitivity of liveness: If an object is pointed to by a live object it is live as well. We must ensure that a pointer crossing heap boundaries does not go unnoticed by the collector.

Independence of collectors: It must be possible to write a collector for a particular heap, without relying on the collectors for other heaps, provided the root set for this heap is known.

Coordination among heaps: A simple set of conventions is established to ensure that pointers across heaps can be properly traversed.

In Figure 2 three heaps are present: the uncollected, the copy collected, and one user collected heap.

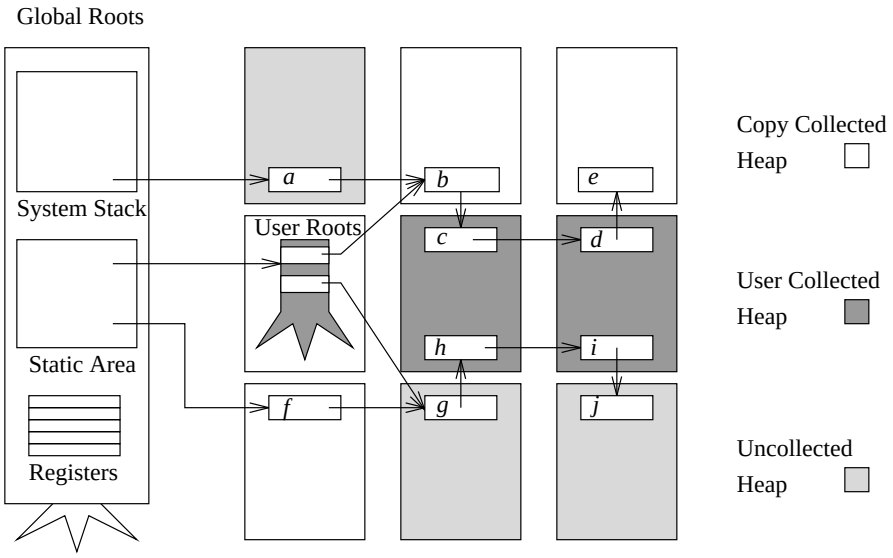

Figure 2. Multiple heaps.

All six possible cross-heap pointers are shown. The user heap is maintained by the user, who keeps a record of the roots into his heap, so that he can perform a collection of that heap when appropriate, without involving the general collector. However the general collector must be capable of identifying for instance object e as live, even though this involves passing through several heaps.

3.4. Customising the GC

The basic operations of a copying tracing collector are traversal and scavenging. The `traverse` procedure is used in the first phase of the collector to identify live objects, the `scavenge` procedure is used to copy an object or perform whatever action is needed to preserve it.

One way to customise these operations is to use the mechanism of *callbacks*, used for instance in programming window based user interfaces. With this schema, a user would register a specific callback routine with the general garbage collector, for use on specific type of objects. So when the garbage collector recognises one of these objects during traversal, it applies the appropriate callback to collect the object.

Callbacks can be different for each individual object, but this is not necessary for our purposes, so we prefer to replace callbacks with member functions. This makes these functions more convenient to define and to retrieve by the collector through the standard mechanism of C++.

Moreover the `traverse` function could actually be generated automatically and no instructions for registration have to be included in the application programs.

3.5. Coordination

To achieve coordination among collectors for the various heaps, one has to agree to a mechanism that allows traversing objects in different heaps on behalf of the collector for another heap. While traversing a foreign heap, a collector should not be allowed to make changes to the objects it visits, except to update recognised pointers to an object in its own heap, after the object has been moved.

This means that one must perform scavenging only for objects in the heap being collected. In other words the scavenge procedure must remain the same throughout a collection, but the scavenge for one heap must not operate on objects in other heaps. `scavenge` is then implemented as member function of each heap class.

`traverse` instead must be specialised according to the type of the object, so we implement it as a member function of each class of objects.

The interplay between `scavenge` and `traverse` is explained considering the situation in Figure 3:

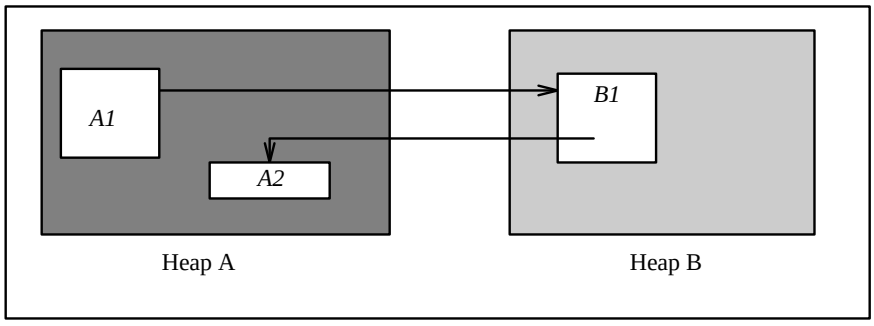

Figure 3. Pointers across heaps.

Suppose a garbage collection is started in heap A which uses a copy collector. While traversing object $A1$, the garbage collector identifies a pointer to the object $B1$, belonging to heap B. Object $B1$ is scavenged by the `scavenge` function of the heap A. This function recognizes object $B1$ as external to heap A, so it does not copy the object, as it would if it were internal to the heap, but only traverses the object to determine whether further objects in heap A can be reached from it. The behaviour of `scavenge` changes again when object $A2$ is reached which belongs to heap A. Applying the scavenge function of heap A has the effect of copying object $A2$.

4. Implementation

Heap memory is divided into pages of equal size. The allocator for each `Heap` requests pages from the low level page allocator, where to allocate its objects. Each page is tagged with the heap to which it belongs.

Collected objects are instances of class `CmmObject` or its derivatives, which have their specialised version of `traverse`. No space overhead is present in `CmmObject` except for what required by C++ for the support of virtual functions.

A bitmap is used to deal with internal pointers to objects. Whenever a CMM object is created, the bit corresponding to its first word is set. Using this information, a pointer inside that object can be normalized to the beginning of the object, simply scanning the bitmap backward until the first set bit is found.

When an object has been moved, its first word is replaced by a forwarding pointer to the new object. As already mentioned, this happens only during garbage collection and the collector can determine this situation from the fact that the object is marked *live* and it is in a page in *FromSpace*.

Overall, one can estimate the space overhead required for using CMM as one word per object (C++ vtable pointer) and 2 bits for each word in the heap. No extra pointer indirection is required, which might introduce execution overhead.

4.1. The CmmObject class

The run time support required for collectable objects is provided by the class `CmmObject` from which every class of collectable objects must be derived.

The creation of collectable objects is performed by the overloaded `new` operator which takes care of allocating the object in a specific heap. Other member functions of class `CmmObject` are used by the primary collector or by user defined collectors.

Here is the public interface for this class.

```
class CmmObject
{
public:
  void* operator new(size_t, CmmHeap*);
  virtual void traverse();
  int size();                       //  returns the size of the object
  CmmObject *next();                //  returns the next adjacent object
  bool forwarded();                 //  tells whether the object has
                                    //     been forwarded
  void SetForward(CmmObject *);     //  sets the forwarding pointer
```

```
  CmmObject *GetForward();        //  returns the forward location
  CmmHeap *heap();                //  returns the heap to which the
                                  //    object belongs
  void mark();                    //  marking primitives
  bool Marked();
}
```

5. Interface for Computer Algebra programmers

A Computer Algebra programmer who wants to use CMM for allocating objects, must define them by means of a collected class, i.e. a class derived from class `CmmObject`. The default collector calls the method `traverse` on collected objects to identify their internal pointers to other objects. Users have to provide `traverse` methods for each class whose data members contain pointers. `traverse` must be defined according to well defined rules presented below, so that the CMM can identify the structure of user defined collected objects.

These rules ensure that superclasses or class objects contained in the class are correctly handled. The following example illustrates the rules, which are a generalization of those in (Bartlett 1989). Suppose the following collected classes were defined:

```
class BigNum: public CmmObject
{
  long data;
  BigNum *next;                   //  Rule (a) applies here
  void traverse();
}

class monomial: private BigNum    //  Rule (c) applies here
{
  PowerProduct pp;                //  Rule (b) applies here
  void traverse();
}
```

A `BigNum` stores in `next` a pointer to a collected object which needs to be scavenged, so `traverse` becomes:

```
void BigNum::traverse()
{
  scavenge(&next);                //  Applying rule (a)
}
```

Because `monomial` inherits from `BigNum`, the method `traverse` for this base class must be invoked; finally, since a `monomial` contains a `BigNum` in `pp`, this object must be traversed as well:

```
void monomial::traverse()
{
  BigNum::traverse();             //  Appling rule (c)
```

```
  pp.traverse();                    //  Applying rule (b)
}
```

Finally, to deal with multiple base classes, we must identify the hidden pointer to the base class present inside an object. This cannot be done in a compiler independent way, so the CMM provides a macro `VirtualBase` which is compiler specific.

In summary the rules are:

(a) for a class containing a pointer, say `class C { `*type* `*x; }`, the method `C::traverse` must contain `scavenge(&x)`
(b) for a class containing an instance of a collected object, say `class C { `*GcClass* `x; }`, the method `C::traverse` must contain `x.traverse()`
(c) for a class derived from another collected class, say `class C:`*GcClass* `{...}`, the method `C::traverse` must contain *GcClass*`::traverse()`.
(d) for a class deriving from a virtual base class, say class `C: virtual` *GcClass* `{...}`, the method `C::traverse` must contain `scavenge(VirtualBase(`*GcClass*`))`;

Preprocessing (Edelson 1992) or compiler support (Samples 1992) could be adopted to avoid hand coding of these functions and risks of subtle errors in programs. We plan to address this issue in the future.

5.1. Object Creation

When creating a collected object one can specify in which heap to allocate it. The parameter `heap` can be supplied in the standard C++ placement syntax for the `new` operator:

```
p = new(heap) Person(name, age);
```

If the user does not specify any heap, the default heap `heap` is used:

```
p = new Person(name, age);
```

which is equivalent to:

```
p = new(heap) Person(name, age);
```

where `heap` is a global variable initialised to the system heap.

When creating collected objects, the programmer can decide case by case where to allocate them. In summary, the following are the alternatives for object allocation:

Heap	**Classes**	**Creation**
uncollected	uncollected	`new/malloc`
copy collected	collected	`new`
user collected	collected	`new(heap)`

where we call collected those classes which inherit from `CmmObject` and uncollected all others.

With the CMM, object allocation is not tied to the type of an object as in other proposals, so a programmer can design his classes without committing to a particular memory policy. The policy can be decided later, or even be different in different portions of an application. It is essential that this can be done without changing a single line in user code implementing operations on the objects. The following example illustrates this point.

6. Example

A foremost algorithm in the PoSSo algebra system (Attardi and Traverso 1994b) is the Buchberger algorithm (Buchberger 1985) for computing the Gröbner basis of a set of polynomials. Dependencies between temporaries and persistent data make the use of explicit memory allocation/deallocation nearly impossible, so the use of a garbage collector was essential. The main step of the algorithm consists of the simplification of a polynomial and involves operations which create many intermediate polynomials of which only the last one is relevant and is inserted into the basis. Once this polynomial has been computed, all the temporary structures allocated can be removed.

The peculiar dynamics of the problem offers an opportunity to exploit the CMM facilities to implement a specific memory management. We created a heap in which the allocation is stack-like (and thus fast), and the garbage collector is called synchronously after each step. Its actual implementation is described in the next section.

The reduction of the S-polynomial `p` of the critical pair `pair`, with respect to a list of polynomials `simplifiers` produces a new polynomial to be inserted in the basis. Here is a fragment from the actual code in the PoSSo library performing this step:

```
Poly *p = pair->SPolynomial();

if (p != NULL) {
  p = simplify(p, simplifiers);
  if (p != NULL)
    p = normalize(p);
}
```

The relevant aspects of the algorithm with respect to memory management are:

1. Large amounts of memory are allocated during `simplify` and most of this memory can be freed at the end of this step. The only data to be preserved is the simplified polynomial which must be inserted into the final basis.
2. In many cases `simplify` returns a zero polynomial. In these cases no memory must be preserved.
3. Since the complexity of the algorithm is exponential, the amount of memory allocated by `simplify` also grows exponentially with the size of the ideal.

We can tune the memory management for this algorithm by means of the CMM, adopting two different heaps: the default one (`CmmHeap::heap`) and a special one for this algorithm (`tempHeap`), an instance of the `HeapStack` class sketched below.

Memory is usually allocated inside the default heap, but before calling `simplify` the heap is switched to the `tempHeap`. All the memory allocated during `simplify` is therefore obtained from the `tempHeap` heap.

Notice that this does not require any changes to any of the remaining functions in the PoSSo library: the algebraic operations on polynomials, coefficients etc. are unmodified and use the standard `new` operator to allocate objects.

After returning from `simplify` we switch back to the default heap, and the polynomial returned by `simplify` is copied into the default heap. At this point the `tempHeap` contains no live data and can be freed with a single operation without involving a garbage collection.

Here again is the code augmented with instructions for CMM memory management.

```
CmmHeap *previousHeap = CmmHeap::heap;  // Save the current heap
CmmHeap::heap = tempHeap;       // Set the current heap to tempHeap

Poly *p = pair->SPolynomial();

if (p != NULL) {
  p = simplify(p, simplifiers);
  if (p != NULL)
    p = normalize(p);
}

CmmHeap::heap = previousHeap;   // Restore the previous heap
p = new Poly (*p);              // Copy p out of the tempHeap
tempHeap->clear();              // empty the tempHeap
```

The last operation on the `tempHeap` is very fast: it involves just resetting a few internal variables to empty the heap.

This solution is simple and works effectively for small problems but has a drawback due to the fact that simplification requires exponential amounts of memory. Therefore a heap of a fixed size will be quickly exhausted with larger problems before the end of one simplification cycle is reached, when it could be recovered. Even if we make the heap of variable size, as in the actual implementation, its size grows so quickly that it will exhaust all available memory.

Therefore we need to reclaim memory from the `tempHeap` earlier, during `simplify`. In this case we cannot just empty the `tempHeap`, because simplification is still in progress and some of its data are in the `tempHeap`. A real garbage collection is required, but it can be a very efficient one because:

> which objects are still in use by `simplify` is known;
> no pointer to objects in the `tempHeap` has been handed out to procedures which might store them elsewhere.

Given these assumptions, before starting the simplification we register as roots for the `tempHeap` the two variables which refer to objects used throughout `simplify`: the variable containing the current polynomial and the one containing the current monomial.

Since the current monomial is part of the current polynomial, the collector will reach it and copy it when traversing such polynomial. However in the code for `simplify` there are references to such monomial directly through the variable `CurrentMonomial`. In order for this reference to be automatically updated to the copy made by the collector, `CurrentMonomial` must be also designated as a root.

After each reduction step, garbage collection on the `tempHeap` is invoked. The garbage collector visits the two registered roots and copies all objects reachable from them. In practice the current polynomial and the current monomial are copied into `ToSpace`. At the start of the next reduction cycle a whole emispace is emptied and available for further allocation.

Two remarks: a collection is not actually performed after each step, but only when the percentage of space left in the heap is below a certain threshold. Secondly, it may happen that the heap fills up before the end of a reduction: in such case the heap is expanded as necessary.

Here is a sketch of the code, where `SL->first` accesses the first element of the list of polynomials `SL`, `SL->next` accesses the rest of such list and `simplifier->head->powerp` selects the power product of the head monomial of polynomial `simplifier`:

```
Poly *simplify(Poly *p, PolyList &simplifiers)
{
  if (simplifiers == NULL)
    return p;

  CurrentPolynomial = p;
  CurrentMonomial = *p;

  tempHeap->roots.setp(&CurrentPolynomial);
  tempHeap->roots.set(&CurrentMonomial);

  while (CurrentMonomial != NULL) {
      bool reduced = false;
      //  iterate through the list of simplifiers
      PolyList SL;
      for (SL = simplifiers; SL != NULL; SL = SL->next) {
          Poly *simplifier = &SL->first;

          if (divisible(CurrentMonomial, simplifier->head->powerp)) {
              CurrentMonomial = reduce(simplifier);

              tempHeap->collect();
              reduced = true;
              break;   //  restart reductions
            }
        }
      if (!reduced)
         CurrentMonomial = &CurrentMonomial->next;
    }
    tempHeap->roots.unsetp(&CurrentPolynomial);
```

```
    tempHeap->roots.unset(&CurrentMonomial);
  return CurrentPolynomial;
}
```

7. Interface for heap programmers

To manage a heap one normally has to maintain the set of roots for the objects in the heap, manage the pages where objects are allocated and implement the memory allocation and recovery primitives. A suitable encapsulation for these functionalities is provided by the `Heap` class.

7.1. The CmmHeap Class

A class implementing a heap must supply definitions for the following pure virtual functions: `allocate` and `reclaim`, implementing the memory allocation strategy, `collect` to perform collection, and `scavenge`, the action required to preserve live objects encountered during traversal. Heap classes are derived from the abstract class `Heap`, defined as follows:

```
class CmmHeap
{
 public:
  CmmHeap();                          //  initialiser
  virtual CmmObject* alloc(int bytes) = 0;
  virtual void scavenge(CmmObject **ptr) = 0;
  virtual void collect() = 0;
  bool inside(CmmObject *ptr);     //  checks if ptr is within this heap
  RootSet *roots;
}
```

`roots` is a pointer to an instance of class `RootSet`, used for registering potential roots. The CMM provides two predefined heap classes:

`DefaultHeap`: encapsulates the primary collector of the CMM which implements Bartlett's mostly-copying discipline;
`UncollectedHeap`: it provides the standard manual allocation discipline. It is available through the default `new` operator or the functions of the `malloc` library. Objects not inheriting from `CmmObject` are allocated in this heap.

7.2. The root set

Some heaps may require the user to register the possible roots explicitly. The class `RootSet` is designed to support managing roots. It provides the following primitives:

```
void set(CmmObject *);
void set(CmmObject **);
void unset(CmmObject *);
void unset(CmmObject **);
void scan();
```

`set` and `unset` are used to (un)register (pointers to) GC objects as roots. `scan` is invoked to traverse objects reachable from the root set.

7.3. Example

We illustrate the interface for heap programmer by showing how to build the `HeapStack` used in the previous section, which is simplified version of the actual heap used in PoSSo.

In this version the size of the heap is fixed, and two spaces are used to perform a copying collection. The real solution adopted in the PoSSo library is more complex and uses multiple spaces.

First we define the `HeapStack` class as a `CmmHeap` consisting of two areas which implement the `FromSpace` and the `ToSpace` of the collector, and a `RootSet` to register the roots to use for the collection:

```
class HeapStack: public CmmHeap
{
  public:
   CmmObject* alloc(int);
   void reclaim(CmmObject*) {};
   void scavenge(CmmObject **);
   void collect();
   HeapStack(int);

  private:
   char* FromSpace, ToSpace;
   char* FromTop, ToTop;
   int size;
   CmmObject* copy(CmmObject *ObjPtr);
}
```

The creation of a `HeapStack` involves requesting two groups of pages for the two spaces:

```
HeapStack::HeapStack(int bytes)
{
   size = bytes;
   FromSpace = allocate_pages(bytes / BYTESxPAGE, this);
   ToSpace = allocate_pages(bytes / BYTESxPAGE, this);
}
```

Allocating memory for an object consists just in advancing the index `FromSpace`:

```
CmmObject* HeapStack::alloc(int size)
{
   int words = BYTEStoWORDS(size);
   if (words <= size - FromTop) {
      FromTop += words;
      return (CmmObject *)(FromSpace + FromTop);
   }
```

```
   else return (CmmObject *)NULL;
}
```

The collector uses the root set to traverse the roots. After having moved to `ToSpace` all the objects reachable from the roots, it traverses those objects in order to move all further reachable objects. As the final step the collector exchanges the roles of `FromSpace` and `ToSpace`.

```
void HeapStack::collect()
{
   char *tmpSpace;
   CmmObject *objPtr;

   //  swap fromSpace and toSpace
   tmpSpace = fromSpace; fromSpace = toSpace; toSpace = tmpSpace;
   fromTop = toTop; toTop = 0;
   //  First traverse the objects registered as roots
   roots.scan();
   //  Now traverse the objects just moved
   objPtr = fromSpace;
   while (inside(objPtr)) {
      objPtr->traverse();
      objPtr = ObjPtr->next();
   }
}
```

This code relies on support provided by classes `CmmObject` and `HeapStack`.
The specific action required for scavenging objects is as follows:

```
void HeapStack::scavenge(CmmObject **ptr)
{
  CmmObject *p = basePointer((GCP)*ptr);    //  find the start of the object
  int offset = (char *)*ptr - (char *)p;
  if (!inside(p))
    visit(p);
  else if (FORWARDED(p))
    *ptr = (CmmObject *)((char *)p->GetForward() + offset);
  else {
    CmmObject *newObj = copy(p);
    p->SetForward(newObj);
    *ptr = (CmmObject *)((char *)newObj + offset);
  }
}
```

8. Performance

To compare the performance of the CMM and the original Bartlett's implementation, we run several well-known test cases for the Buchberger algorithm on a SparcStation 10

Table 1. Benchmarks

	Bartlett	CMM default	CMM TempHeap	Improv. %
katsura5	3.59	3.79	3.17	17
cohn1	12.45	8.68	6.85	22
cyclic6	37.58	28.78	19.77	32
valla	56.96	46.43	34.3	27
katsura6	356.41	258.45	211.58	18

with 32 Mbytes of physical memory. The timings in seconds achieved on these benchmarks are summarised in Table 1.

The improvement appears to be significant across a variety of benchmarks, ranging from 17 to 32%. It is also interesting to note that the CMM default algorithm has quite better performance to Bartlett's original, despite the overhead due to its use of C++ and member functions rather than straight C.

To study in detail how much the garbage collector influences the overall performance, we analysed the various versions by means of a program profiler.

In Table 2 we report the results of running the benchmark katsura6 (Katsura *et al.* 1987), providing details on the timings of memory operations: `alloc`, the primitive allocator; `gc`, overall time spent in garbage collection; `pure alloc`, allocation time less collection time; `gc calls`, the number of calls to the collector; `gc average`, average time of a collection. In the last column we show two figures for each operation, one for the default heap and the second for the TempHeap, since both heaps are used.

Table 2. Analysis

	Bartlett	CMM default	CMM TempHeap
Katsura6 (profiled)	452.38	275.86	213.49
`alloc`	223.68	43.96	3.19+0.04
`collect`	215.07	37.06	2.47+0.03
`pure alloc`	8.61	6.90	0.72+0.01
`gc calls`	931	450	16584+2
`gc average`	0.23	0.08	0.00+0.01

The use of TempHeap produces striking results: the garbage collection time becomes negligible and accordingly allocation time is also drastically reduced. The total allocation cost using the default CMM heap is 44 sec which is slightly less than the gain from using the TempHeap. Therefore the 18% improvement in the overall execution time achieved by means of the TempHeap is quite close to using an ideal allocator with zero cost and so this represents the maximum increase in performance one can expect to obtain by improving memory management.

As Buchberger and Jebelean (1993) have noted, the cost of arithmetic computations may become dominant in the Buchberger algorithm. This is the case with the katsura6 benchmark, where significant amounts of time are spent in the arithmetic of arbitrary pre-

cision integers (42.5% in `_mpn_addmul_1`, 10.2% in `_mpn_mul_1`, 4.3% in `_mpz_mul`, 3.3% in `_mpn_mul`) which grow respectively to 50.4%, 12.9%, 4.8% and 4.1% when using the TempHeap. These are routines from the GNU Multiple Precision library (GMP) (FSF 1994) that are used in the PoSSo library: `mpn_addmul` is an assembly code routine which multiplies a limb vector with a limb and adds the result to a second limb vector, `_mpn_mul_1` multiplies a limb vector with a limb and stores the result in a second limb vector, `_mpn_mul` multiplies two natural numbers and `_mpz_mul` multiplies two integers.

We have also received satisfactory reports on the performance of CMM by the partners in the PoSSo project who used it in particular for implementing a linear algebra package (Rouillier 1994).

9. Related and Future Work

The Böhm-Weiser collector (Boehm and Weiser 1988) is another well known collector for C++ which is totally conservative and therefore quite convenient to use. However it is not customisable and is subject to unduly retention of space and memory fragmentation since it cannot compact memory.

Work on adding garbage collection to C++ has been done by D. Samples and Edelson (1992). Samples (Samples 1992) proposes modifying C++, to include a garbage collection environment as part of the language. This may be a good long term approach for garbage collection in C++ but is not suitable for a project like PoSSo which needs portable garbage collection facilities immediately. On the other hand our work demonstrates that the flexibility of object oriented languages allows us to implement a complex environment, like CMM, without requiring modifications to the language.

Edelson (1992) has been experimenting with the coexistence of different garbage collection techniques. The flexibility of the solutions he adopts in his approach allows the coexistence of different garbage collectors, but he does not provide any interface to the user to customise and/or define his own memory management facilities.

Ellis and Detlefs (1993) propose some extensions to the C++ language to allow for collectable object. The major change is the addition of the type specifier `gc` to specify which heap to use in allocating the object or a class. With some minor modifications discussed by Attardi and Flagella (1994), this proposal is compatible with the CMM. The Ellis-Detlefs proposal contains other valuable suggestions, for instance making the compiler aware of the garbage collection presence and avoid producing code where a pointer to an object (which may be the last one) is overwritten. This can happen for instance in optimizing code for accessing structure members.

We are investigating the possibility of incorporating the CMM in the run-time support used by FOAM (Watt *et al.* 1994b), the intermediate language of A# (Watt *et al.* 1994), a language for symbolic algebra. FOAM itself is implemented by translation into C code which uses a run-time support which includes a totally conservative garbage collector.

Building a common run-time for A# and PoSSo would be a significant result, since it will enable sharing of libraries and access to the facilities of both systems.

Another challenge would be to incorporate into a C++ compiler the minimal facilities required for CMM support: the addition of the `gc` keyword, proposed by Ellis and Detlefs (1993), could facilitate this.

10. Conclusion

The CMM offers garbage collection facilities which are both flexible and efficient.

Programmers can select the collector which is most suitable to the need of each algorithm: either the default collector, or a specific collector or no collector at all. The algorithm can be in control when necessary of its memory requirements and does not have to adapt to a fixed memory management policy.

The CMM is implemented as a C++ library which can be linked with the application code. It is being heavily used in the implementation of high demanding computer algebra algorithms in the PoSSo project. The CMM provides the required flexibility without degradation in performance as compared to versions of the same algorithms performing manual allocation.

11. Availability

The sources for CMM are available for anonymous ftp from site `ftp.di.unipi.it` in the directory `/pub/project/posso`. Please address comments, suggestions, bug reports to `cmm@di.unipi.it`.

Acknowledgements

Carlo Traverso and John Abbott participated in the design. J.C. Faugère provided the idea for this work. Joachim Hollman and Fabrice Rouillier helped in testing the first prototype implementation. Discussions with J. Ellis were useful to ensure compatibility of his proposal with our framework. Comments from L. Semenzato helped to improve the presentation.

References

Attardi, G., Flagella, T. (1994). A customisable memory management framework. *Proceedings of USENIX C++ Conference 1994*, Cambridge, Massachusetts, April 1994.

Attardi, G., Flagella, T. (1994b). Customising object allocation. in M. Tokoro and R. Pareschi (eds.) *Object-Oriented Programming, Proceedings of the 8th ECOOP, Lecture Notes in Computer Science* **821**. Berlin:Springer-Verlag, 320–343.

Attardi, G., Traverso, C. (1995). The PoSSo Library for Polynomial System Solving. *Proc. of AIHENP95*, World Scientific Publishing Company.

Bartlett, J.F. (1988). Compacting garbage collection with ambiguous roots. Tech. Rep. 88/2, DEC Western Research Laboratory, Palo Alto, California.

Bartlett, J.F. (1989). Mostly-copying collection picks up generations and C++. Tech. Rep. TN-12, DEC Western Research Laboratory, Palo Alto, California.

Boehm, H.J., Weiser, M. (1988). Garbage collection in an uncooperative environment. *Software Practice and Experience* **18**(9), 807–820.

Buchberger, B. (1985). Gröbner bases: an algorithmic method in polynomial ideal theory. *Recent trends in multidimensional systems theory*, N. K. Bose (Ed.), D. Reidel Publ. Comp., 184–232.

Buchberger, B., Jebelean, T. (1993). Parallel rational arithmetic for Computer Algebra Systems: motivating experiments. *3rd Scientific Workshop of the Austrian Center for Parallel Computation*, Report ACPC/TR 93-3.

Edelson, D.R. (1992). Precompiling C++ for garbage collection. *Memory Management*, Y. Bekkers and J. Cohen (Eds.), *Lecture Notes in Computer Science* **637**, Berlin:Springer-Verlag, 299–314.

Edelson, D.R. (1992). A mark-and-sweep collector for C++. *Proc. of ACM Conference on Principle of Programming Languages*.

Ellis, J.R., Detlefs, D.L. (1993). Safe, efficient garbage collection for C++. Xerox PARC report CSL-93-4.

Faugère, J.C. (1994). Résolution des systèmes d'équations algébriques. PhD thesis, Université Paris 6.

The GNU MP library, Free Software Foundation, Version 1.99.8. (1994).

Katsura, S., Fukuda, W., Inawashiro, S., Fujiki, N.M., Gebauer, R. (1987). *Cell Biophysics* **11**, 309–319.

Rouillier, F. (1994). Personal communication.

Samples, A.D. (1992). GC-cooperative C++. *Lecture Notes in Computer Science* **637**. Berlin:Springer-Verlag, 315–329.

Watt, S.M., Broadbery, P.A., Dooley, S.S., Iglio, P., Morrison, S.C., Steinbach, J.M., Sutor, R.S. (1994). A first report on the A# compiler. To appear in *Proc. 1994 International Symposium on Symbolic and Algebraic Computation (ISSAC 94)*. IBM Research Report RC 19529.

Watt, S.M., Broadbery, P.A., Iglio, P., Morrison, S.C., Steinbach, J.M., Sutor, R.S. (1994b). FOAM: A First Order Abstract Machine, V 0.35. IBM Research Report RC 19528.

Wentworth, E.P. (1990). Pitfalls of conservative garbage collection. *Software Practice and Experience* **20**(7), 719–727.

Wilson, P.R. (1992). Uniprocessor garbage collection techniques. *Memory Management*, Y. Bekkers and J. Cohen (Eds.), *Lecture Notes in Computer Science* **637**, Springer-Verlag, 1–42.

Zorn, B. (1993). The measured cost of conservative garbage collection, *Software Practice and Experience*, **23**, 733-756 (1993).