A Customisable Memory Management Framework

Giuseppe Attardi [*]
Tito Flagella [†]

February 1994

Abstract

Memory management is a critical issue for many large object-oriented applications, but in C++ only explicit memory reclamation through the `delete` operator is generally available. We analyse different possibilities for memory management in C++ and present a dynamic memory management framework which can be customised to the need of specific applications. The framework allows full integration and coexistence of different memory management techniques. The Customisable Memory Management (CMM) is based on a *primary collector* which exploits an evolution of Bartlett's mostly copying garbage collector. Specialised collectors can be built for separate memory heaps. A `Heap` class encapsulates the allocation strategy for each heap. We show how to emulate different garbage collection styles or user-specific memory management techniques. The CMM is implemented in C++ without any special support in the language or the compiler. The techniques used in the CMM are general enough to be applicable also to other languages.

1 Introduction

As an alternative to explicit reclamation of heap memory, automatic recovery of unused memory can be performed through the technique of *garbage collection*. The garbage collector's function is to find data objects that are no longer in use and make their space available for further use by the program. An object is considered *garbage*, and therefore subject to reclamation, if it is not reachable by the program via any path of pointer traversal. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never follow a "dangling pointer" leading to a deallocated object.

There are good reasons to prefer automatic memory management: *safety*, avoiding the risk of deallocating an object too soon; *accuracy*, avoiding to forget 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; *reduced burden* on programmers who are relieved from taking care of memory management. Nevertheless garbage collection has not yet come into general use, sometimes for fears of losing efficiency but mostly for the lack of availability of the technique.

Recent research has proved that many of the limitations of traditional garbage collection techniques can be alleviated. Some experiments have even shown that explicit memory deallocation (using primitives like `free` or `delete`) is not necessarily faster than automatic reclamation of free memory [Breuel 92]. Techniques like *generational* garbage collection have been developed to minimise latency during garbage collection.

While these experiences have proved that garbage collection is a valuable technique, the variety of proposals is in itself an indication that the ideal garbage collector is impossible to achieve. A good design is one that strikes the appropriate tradeoff among many conflicting goals.

[*]Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy. Net: `attardi@di.unipi.it`.
[†]Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy. Net: `tito@di.unipi.it`. This paper appeared in the USENIX C++ Conference Proceedings 1994.
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.

We faced the task of developing memory management facilities for a large research project: the ESPRIT BRA PoSSo aims at building a sophisticate symbolic algebra system for solving polynomial systems. Researchers working on different parts of the system have different requirements on the memory management. Some users prefer a copying garbage collector in order to maintain locality of their data. Others prefer a mark-and-sweep approach because of the fixed size of their data. The core algorithms of PoSSo required a special kind of memory management due to the particular FIFO dynamics of memory usage exhibited in certain portions of the Buchberger algorithm for computing a Gröbner basis [Buchberger 85].

These requirements led us to design a framework whereby users can select among different garbage collection strategies, ranging from manual management to fully automatic garbage collection, and can also implement their own specialised memory management as appropriate for their task. Without the support provided by our framework, if memory management were left to each programmer:

1. each user would have to introduce support for memory management in his code. This means adding extra fields to data and providing code for basic memory management operations, like computing the size of objects, the address ofthe next object in the heap, etc.

2. in large applications where different memory management facilities are required, different interfaces would be present for each memory manager (MM).

3. it would be impossible to mix data under different MM's. If an object under control of one MM contains a reference to an object controlled by another MM, such reference may not be noticed by the first MM, leading to incorrect memory reclamation.

When an intensive use of such facilities is required and a variety of memory management techniques are needed, the programs become very difficult to write and maintain, and subtle "memory leak" bugs may arise which are nearly impossible to find.

The Customisable Memory Management (CMM) addresses these issues by providing a general framework within which several policies can coexist. The framework takes full advantage of the object oriented paradigm of C++, and provides a consistent and simple interface for programmers.

The CMM is a memory management facility supporting complex memory intensive applications in C++. It consists of:

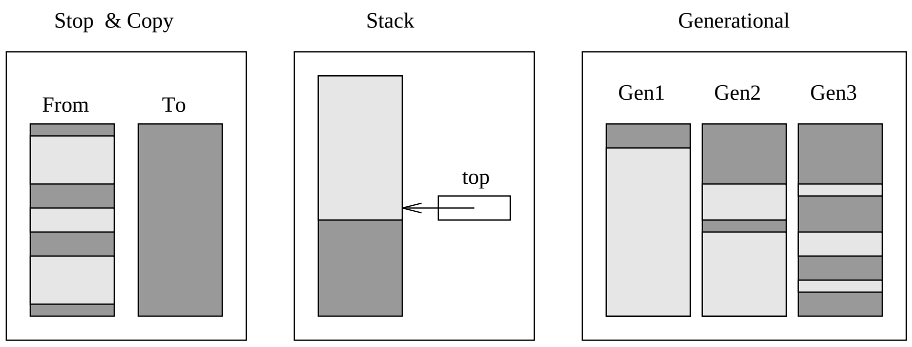
1. a general purpose garbage collector for C++; this collector is called *primary garbage collector* and is a variant of Bartlett's mostly copying collector [Bartlett 89];

2. a user interface: this is the interface used by programmers to access the CMM;

3. a programmer interface: a set of facilities used by CMM programmers to define specific memory management policies as appropriate for their applications.

With CMM users can select among several predefined memory management facilities, define their own, or customise those provided in the framework.

For instance it is conceivable a situation like in the following figure, where three different memory management policies are available or even used together in the same application: a traditional stop-and-copy collector, a specialised stack allocator for portions of the algorithm with controlled behaviour or a generational collector for real-time tasks such as user interfaces. The first two mechanisms are already available in the CMM, while a generational collector has been implemented by Bartlett [Bartlett 89].



The mechanism to implement these alternative policies is the Heap abstraction which we develop in this paper. Specific algorithms are used and particular data structures are maintained by each Heap to ensure its proper behaviour. A critical question is what to do with pointers which cross the boundaries of Heaps. If no such pointers are allowed, then a Heap need only to be concerned with objects it has allocated and on which it has some control. We considered this solution too restrictive, since it would not allow portions of applications built separetely by different people to freely exchange data. We took therefore special care to design the mechanism of Heaps to ensure that different Heaps can coexist and data of different sources can be mixed. The amount of coordination necessary to achieve this goal, has been kept to a minimum, and consists of a few simple functions that each class of collectable objects must provide and which it would be possible to generate automatically. To achieve coordination in a simple and effective way, we exploit the object oriented features of C++. In practice, all the operations of the collector are performed through member functions of the class of each object. However, the action of the collector on an object may vary also depending on the Heap where the collection started, not just on the Heap to which the object belongs. For instance if the collection starts in the Stop&Copy Heap, it applies its methods to mark and traverse the object in that zone, but if a pointer leads into a StackZone, those objects are unobtrusively traversed without modifying them. Only if such traversal leads back into the original Heap, the full collector operation resumes.

2 Requirements

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

- *portability*: the CMM is simply a library of C procedures and C++ classes, which can be used with any C++ compiler. Alternative solutions rely on changes to the underlying language or compiler.
- *coexistance*: code and objects built with the CMM can be exchanged with traditional code and libraries. No restriction exist on whether a collected object can point to a non collected object and viceversa. We wanted to 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.
- *algorithm specific customisation*: the allocation policy can be customised to the particular needs of an algorithm. This is different from other solutions, where the allocation policy is associated to the type of an object. For instance, in the proposal by Ellis and Detlefs [Ellis 93], it is possible to specify whether an instance or a class is allocated in the collected heap, rather than in the non collected heap. 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 zone which can be freed quickly once a certain portion

of the simplification algorithm is complete; some other time the lifetime of the polynomial cannot be predicted, so it must be allocated in the general heap zone.

- *multiple logical heaps*: at least two heaps are necessary, one for collectable objects and one for traditional 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, called Heap.
- *usability*: only minimal burden is placed on the programmer who wants to use the collector. When collectable objects are required the programmer needs to define their class as inheriting from the base class `GcObject` and supply a method for traversing them, a task which could be automated.
- *separation of concerns*: memory management code needs not to be included within algorithms, and it is possibile to change the memory policy just by selecting which heap is employed by the algorithm.
- *efficiency*: the implementation is efficient enough to be as good and sometimes better that hand tuned allocation.

The CMM allows customisation of the collector and provides a few pre-built variants. One could argue whether a single general strategy could fit all the needs. For instance a generational garbage collector ensures that memory is reclaimed quickly. However not even a generational garbage collection is good enough for applications like PoSSo where one must prevent or delay garbage collection as much as possible, not just make its duration shorter. For the vast majority of applications a general purpose strategy is adequate, and the CMM provides a good one by default. But for research or applications that need to push the limits of technology, the CMM provides a solution with limited effort on the user.

In the rest of the paper, we recall the general principles of memory management, then present our primary collection algorithm, then discuss the CMM, its implementation and its usage. Finally we illustrate how to emulate different garbage collector styles and application specific memory management policies.

3 Dynamic Memory Management: Concepts and Terminology

A garbage collector in principle could reclaim the space occupied by all objects that the running program will no longer access. Unfortunately this is an undecidable property; therefore garbage collectors must adopt a simpler criterion based on the notion of potentially accessible or *live object*. A garbage collection mechanism basically consists of two parts [Wilson 92]:

1. distinguishing the *live objects* from the garbage in some way, or *garbage detection*;
2. reclaiming the garbage objects' storage, so that the running program can reuse it.

The formal *criterion* to identify *live* objects is expressed in terms of a *root set* and *reachability* from these roots. The *root set* consists of the global and local variables, and any registers used by active procedures. Heap objects directly reachable from any of these variables can potentially be accessed by the running program, so they are *live* objects which must be preserved. In addition, since the program might traverse pointers from these objects to reach other objects, any object reachable from a live object is also live. Thus the *live set* is the set of objects in some way reachable from the roots. Any object not in the live set is garbage and can be safely reclaimed.

Several variations are possible on this general working schema, depending on:

1. how to identify the roots (conservative, explicit registration, smart pointers, etc);
2. how to identify internal pointers pointing to other GC objects (compiler support, user support, conservative, etc);

3. how the GC distinguishes live objects from garbage (marking or copying, with their many variants).

Quite different implementations result from different combinations of the above techniques. We can characterise as follows, according to these criteria, some of the most recent implementations of garbage collectors for C++:

	Identify roots	**Identify internal pointers**	**Distinguish live objects**
Boehm	Conservative	Conservative	Mark
Edelson	Smart Pointers	User assisted	Copying
Bartlett	Conservative	User assisted	Promotion & Copying

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

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 root.* A garbage collector is *type-accurate* when it is able to distinguish which values are genuine pointers to objects. Some garbage collectors adopt a combination of these two techniques: some pointers are dealt conservatively, while others are treated in a type accurate way.

The main limitations of a purely conservative collector are memory fragmentation in applications handling objects of many different sizes, arising 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 90].

The alternative approach which is *type-accurate* in identifying objects faces some non trivial problems with hidden pointers. One such case is the `this` pointer in C++: whenever a method is invoked on an object, a pointer to that object is passed to the method via the stack as the implicit local variable `this`, but only the compiler knows where such variable is actually located. The only compiler-independent way to catch such pointers is to examine the stack conservatively. Failing to trace this pointer is dangerous: the object might be reclaimed or moved without updating the pointer. In both cases a *dangling pointer* is generated with serious consequences for the integrity of the program.

Both these limitations are avoided in the partially conservative approach proposed by Bartlett for his *mostly copying garbage collector.* We chose this technique as the basis for developing our customisable collector.

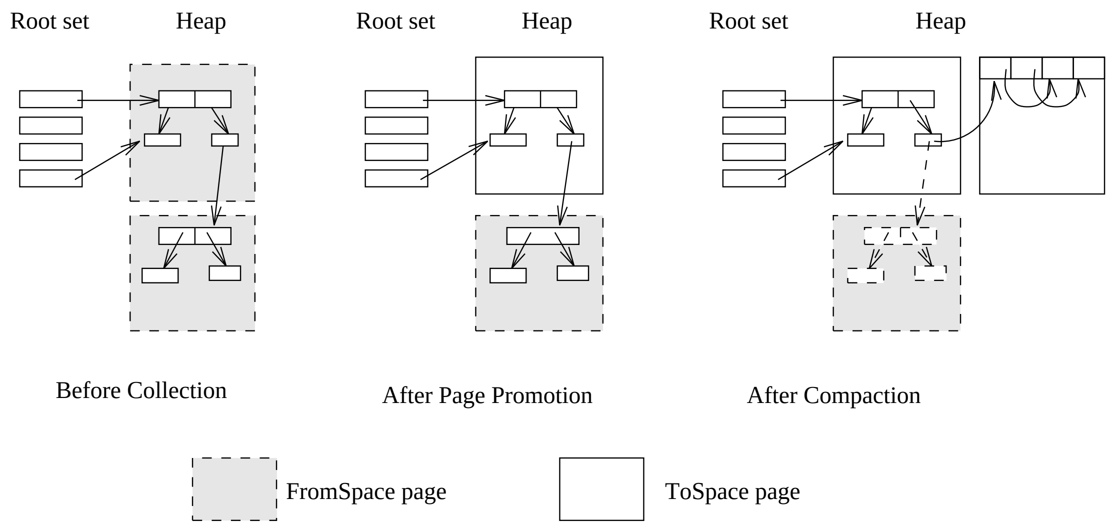
4 The Primary Collector

The Customisable Memory Management relies on an underlying general mechanisms for identifying objects, moving them and recovering memory. These mechanisms constitute the *primary collector* of the CMM and are based on Bartlett's technique. We illustrate here the technique and how we improved it for our needs.

4.1 Bartlett's mostly copying collector

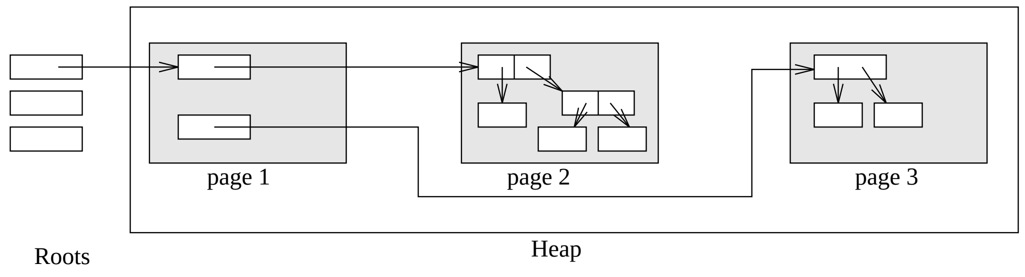
A mostly-copying garbage collector performs compacting collection in the presence of ambiguous pointers in the root set. Bartlett's implementation (BGC) is an evolution of the classical stop-and-copy collector which combines copying and *conservative* collection. BGC does not copy those objects which are referenced by ambiguous roots, while most other live objects are copied.

The heap used by BGC is a non necessarily contiguous region of storage, divided into 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. 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 is required to add a few simple declarations which enable the collector to locate the internal pointers within objects.



4.2 Revised Algorithm

Experimenting with the original implementation of Bartlett's mostly copying algorithm, we noticed that for some of our programs the amount of garbage not reclaimed was too high. The main reason for this was that a whole page was promoted when it contained just a single object reachable from a root: all objects in that same page will be preserved as well as their descendants, thereby missing to reclaim significant amounts of memory. This is illustrated in the following figure, where the object in the rightmost heap page is retained since it is pointed from within the leftmost page which has been promoted.



To improve the ability to reclaim storage of Bartlett's algorithm we keep a record of those objects actually reachable within a page being promoted during the first phase. This allows us to identify reachable objects in promoted pages. This information is contained in a bit table called `LiveMap`.

Here is our revised version of Bartlett's algorithm, which in most cases is still iterative:

1. Clear the `LiveMap` bitmap

2. Scan the root set to determine objects which cannot be moved. Any directly reachable object is marked as *live* setting a bit in the `LiveMap` bitmap and the page to which it belongs is promoted.

3. Scan each promoted page linearly, looking for live objects. Traverse each live object by applying the following procedure to each pointer it contains:

   (a) if the pointer lays outside the heap do nothing;

   (b) if it points to an object not yet reached: scavenge the object if it belongs to a non promoted page, i.e. copy it, mark the copy as *live*, set a forwarding pointer within the object to the copy. Otherwise mark the object *live* and, in case it is past the current scanning position, recursively traverse it.

   (c) if it points to a *live* object in a non promoted page update the pointer to the forward position.

All new pages allocated for copying reachable objects belong to *ToSpace*, therefore the algorithm does not need to traverse copied objects. A copied object is traversed when the collector examines its page, so traversal is rarely recursive.

This algorithm does not require a forward bit as used in Bartlett's implementation: we can determine that an object has been forwarded if it is marked as live and contained in a non-promoted page. We also do not need to store in each object its size which Bartlett requires in order to scan through the objects in a promoted page. And finally, since we can determine the heap to which an object belong from its address, we can completely get rid of the one word of header required in Bartlett's algorithm, therefore eliminating any space overhead for collected objects.

Our experiments with the new algorithm show improvements up to 50% in the amount of space reclaimed with the new algorithm.

5 Multiple Heaps

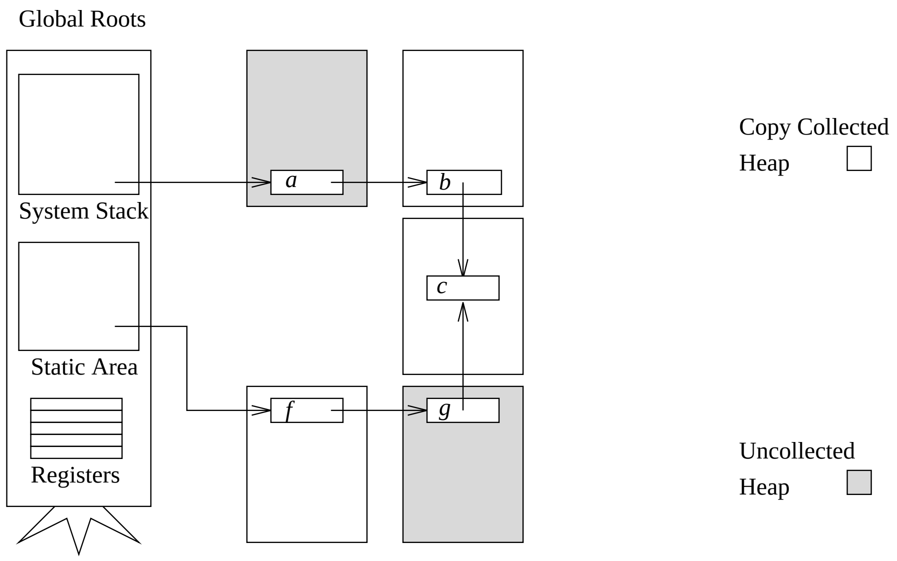
Bartlett's algorithm creates and manages a heap of objects which are collected by copying. The traditional uncollected heap is still available through the primitives `malloc` or `new` on uncollected classes. The uncollected heap cannot be eliminated since there are programs and libraries which may use uncollected object in an unsafe way for the collector [Ellis 93], and there are objects that can't be relocated. It must be possible however that objects in the uncollected heap point to objects in the collected heap and viceversa.

Pointers across heaps must be dealt carefully. The original Bartlett's implementation requires that pointers to collected objects be registered as roots. This is not practical, since it would entail registering as root any collected object which is passed to a library which might store it internally. This can be cumbersome to do and may be accidentally forgotten.

Therefore we need to extend the collector algorithm so that it is capable of discovering such pointers. The solution will later be generalized to deal with other logical heaps, created and maintained by users.

The uncollected heap should be considered as part of the root set. An obvious solution would then be to scan conservatively the entire uncollected heap searching for pointers to collected objects. This would be too expensive and would posit as live also objects pointed from unreachable locations in the heap. Alternatively one could perform a first complete scan from the root set to identify cross-pointers from uncollected to collected objects, in order to promote the pages where the latter reside, and then the mostly-copying algorithm would be applied. This is also a costly alternative, since it requires traversing twice the objects.

If we examine where Bartlett's algorithm fails, we can figure out an alternative solution. In the following figure, object c is pointed both from b, in the copy collected heap, and from g, in the uncolleted heap.



If we apply the mostly-copying algorithm, the pages where b and f will be promoted since they are pointed from roots. In the copy phase object c would be copied to a new page and the pointer in b will be updated. However, when later we reach c from g, we discover that its page should have been promoted. We could in fact promote it now, if only b had not been updated. This in fact suggests a solution: we do not update pointers when an object is copied, but we just record the location to be updated, using a temporary bitmap. If we discover that the object should not have been moved, we restore all the objects in its page from their copies. The updates to pointers are performed only at the end of the algorithm, using the bitmap and the forwarding pointer stored in the objects. This technique is similar to the one suggested by Detlefs [Detlefs 92] to handle C/C++ unions of pointers and non-pointers.

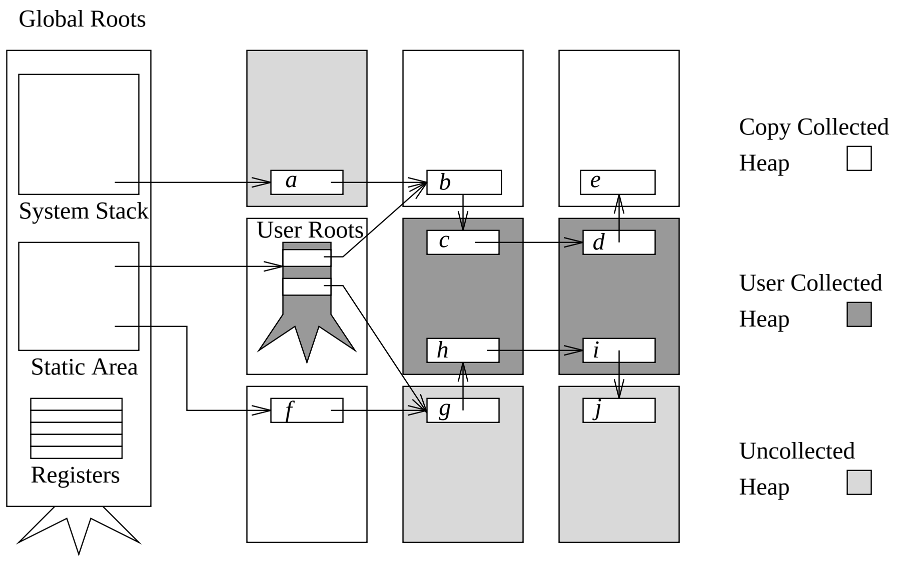
5.1 User Collected Heaps

With the algorithm described so far, two heaps are available: an uncollected heap for non garbage collected objects and a collected heap.

Our goal is to allow users to build their own heaps with specific allocations 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 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 such heap is known.
- coordination among heaps: a simple set of conventions is established to ensure that pointers across heaps can be properly traversed.

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



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 relative to 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 requires to cut across several heaps.

5.2 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 object, the `scavenge` procedure is used to copy an object or perform whatever action is needed to preserve it.

Supporting multiple heaps requires to specialise these operations along two dimensions: according to the type of the object for traversal; and according to the heap where the object is located for scavenging.

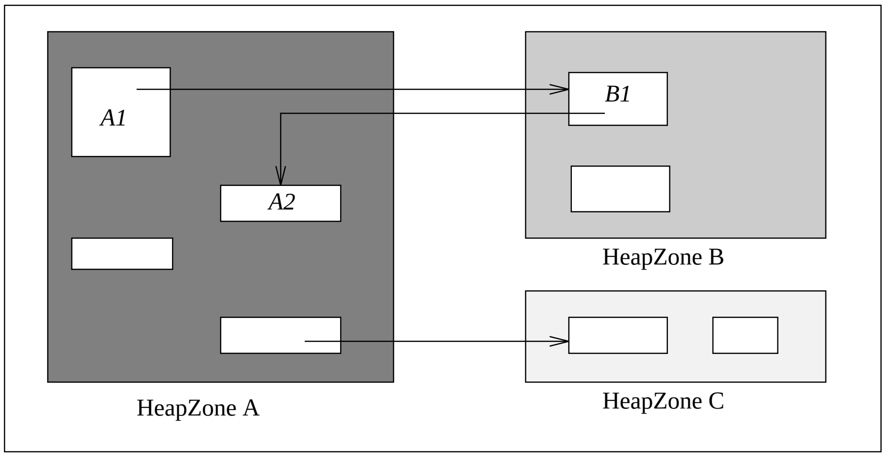
One way to customise these operations is to use the mechanism of *callbacks*, used for instance in programming window based user interfaces. With this schema, a user would register a specific callback routine with the general garbage collector, for use on specific type of objects. So when the garbage 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 preferred 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 registration has to be added in the application programs. A version of our algorithm for C would still exploit callbacks.

5.3 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 recognized pointers to an object in his heap, when the object is moved.

So it is important that the traversal mechanism is uniform but capable to distinguish by whom it was initiated. This is achieved supplying a Heap parameter to `traverse` and making `scavenge` a member function of a `Heap`. Consider for instance the following situation:



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 zone, 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 $A2$.

6 The CMM Run Time

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.

The CMM provides a `malloc` routine which uses such pages to allocate objects, implementing the traditional uncollected heap. `malloc` actually creates an instance of class `CmmObject`, which contains an array of the required size, and returns a pointer to such array, as expected by calling programs. This is in fact an *interior pointer* inside an object, and we exploit the ability of the CMM to map such pointers to their base. This allows us to traverse also `CmmObject`s by means of its member function `traverse`, defined as follows:

```
void CmmObject::traverse(Heap* heap) {
   for (int i = 0; i < this->size(); i++)
      promote_page(block[i]);
}
```

so that it promotes pages which are pointed from within the block. The only essential information that `CmmObject` must provide is the size of the block.

In all other collected heaps, the objects allocated are instances of class `GcObject` or its derivatives, which have their specialised version of `traverse`. No space overhead is present in `GcObject` except for what C++ must supply 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.

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*.

6.1 The GcObject class

The run time support required for collectable objects is provided by the class `GcObject`. Every class of collectable objects is derived from `GcObject`.

Users access the services of the CMM mainly by using `GcObject` member functions. The most notable function of `GcObject` is the overloaded `new` operator which takes care of allocating the object in a specific heap. The other functions are used by the primary collector or by user defined collectors.

Here is the public interface for this class.

```
class GcObject
{
public:

  void* operator new(size_t, Heap* = (Heap *)heap);
  virtual void traverse(Heap* zone);

  GcObject *next();             // returns the next adjacent object
  int forwarded();              // tells whether the object has been forwarded
  void SetForward(GcObject *ptr); // sets the forwarding pointer
  GcObject *GetForward();       // returns the forward location of the object
  Heap *zone();                 // returns the zone to which the object belongs
  void mark();                  // marking primitives
  bool IsMarked();
  void SetLiveMap();
};
```

7 CMM User Interface

A collected class must derived from the class `GcObject` provided by the CMM. 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, because it implements the interface between the CMM and 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 generalisation of those in [Bartlett 89]. Suppose the following collected classes were defined:

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

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

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

```
void BigNum::traverse(Heap *zone)
{
  zone->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(Heap *zone)
{
  BigNum::traverse(zone);               // Appling rule (c)
  pp.traverse(zone);                    // 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. For instance, its definition for the GNU C++ compiler is:

```
  #define VirtualBase(A)  & (_vb$ # A)
```

In summary the rules are:

(a) for a class containing a pointer, say `class C` { *type* `*x;` }, the method `C::traverse` must contain `zone->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(zone)`

(c) for a class derived from another collected class, say `class C:`*GcClass* {`...`}, the method `C::traverse` must contain *GcClass*`::traverse(zone)`.

(d) for a class deriving from a virtual base class, say class `C: virtual` *GcClass* {`...`}, the method `C::traverse` must contain `zone->scavenge(VirtualBase(`*GcClass*`));`

Preprocessing [Edelson 92] or compiler support [Samples 92] 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.

7.1 Object Creation

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

```
p = new(zone) 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(zone)`

where we call collected those classes which inherit from `GcObject` 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. For instance, in the PoSSo solver, one sets the variable `heap` to the heap implementing the stack policy before starting the simplification. Throughout the simplification, all objects (monomial, polynomial, large precision integers, lists and so on) are allocated in this heap and freed in a single step at the end of the simplification. After simplification, one reverts to the normal heap. It is essential that this can be done without changing a single line in the user code.

8 Heap Classes

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.

8.1 The Heap 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 Heap;
{
 public:
  int Index();                     // identifies the Heap

  Heap();                          // initializer

  virtual GcObject* allocate(int ObjSize) = 0;
  virtual void reclaim(GcObject* ObjPtr) = 0;
  virtual void scavenge(GcObject **ptr) = 0;
  virtual void collect() = 0;

  // Operations on the Root Set:
  void register(GcObject *);       // add an element
  void register(GcObject **);
  void deregister(GcObject *);     // remove an element
  void deregister(GcObject **);
  void ScanRoots(Heap *zone);      // scan the roots

  bool outside(GcObject *ptr);     // checks if pointer is outside this Heap
  void visit(GcObject *ptr) {
     if (! ptr->IsMarked()) {
        ptr->mark();
        ptr->traverse(this);
     }
  }

 private:
  int index;
  RootSet *roots;
};
```

roots is a pointer to an instance of class RootSet, used for registering potential roots. Depending on the particular type of RootSet used, the collector can be conservative, type-accurate or both. The simplest RootSet considers as possible roots only the objects explicitly registered by the user. The derived class ConservativeRootSet scans also the system stack, the process static data area, and registers for possible roots.

8.2 The Bartlett Heap

The Heap Bartlett encapsulates the primary collector of the CMM. The function gcalloc, gcmove and gccollect are the primitive functions provided by Bartlett's implementation of the collector.

```
class Bartlett: public Heap
{
 public:
  Bartlett() {
    roots = new ConservativeRootSet();
  }

  GcObject* allocate(int ObjSize) {
    return (GcObject *)gcalloc(GCBYTEStoWORDS(ObjSize));
  }

  void scavenge(GcObject **ptr) {
    if (OutsideHeap((int *)*ptr))
       return;
    GcObject *p = GetBeginning((int *)*ptr);
    if (outside(p))
       visit(p);
    else {
       *p->SetForward(gcmove(p));
       ToBeForwarded(ptr);
    }
  }

  void reclaim(GcObject* ObjPtr) {}; // delete does nothing

  void collect() {
    gccollect();                     // the actual Bartlett's collector
  }
}
```

Bartlett's collector starts scanning the set of possible roots to identify live objects. Because it is a conservative collector, `roots` is an instance of `ConservativeRootSet`. The objects it contains are traversed to identify other live objects. Objects are traversed in a type-accurate way by applying the user supplied function `traverse`. `traverse` applies in turn the Heap member function `scavenge` to each reachable object. For each object in the Bartlett Heap, Bartlett's original `gcmove` primitive is used to copy it and compact memory; otherwise the object is visited using the function `visit`, which marks the object if necessary and then traverses it.

8.3 The Uncollected Heap

The uncollected heap is available through the default `new` operator or the functions of the `malloc` library. Objects not inheriting from `GcObject` are allocated in this heap.

8.4 The root set

Many heap zones require the user to explicitly register the possible roots. To support that, the class `Heap` contains an instance of the class `RootSet` supporting the following operations:

```
void set(GcObject *);
void unset(GcObject *);
void setp(GcObject **);
void unsetp(GcObject **);
```

`setp` and `unsetp` are used to (un)register pointers to GC objects as roots. `set` and `unset` are used to (un)register GC objects as roots. Consider the following example:

```
cell GlobalRoot;                              // Define a cell variable

main()
{
  cell *LocalRoot = new cell;                 // Define a cell pointer
  HeapStack *MyHeap = new HeapStack(10000);   // Create a new heap zone
  MyHeap->roots.setp((GcObject **)&LocalRoot); // Register the pointer as a root
  MyHeap->roots.set(&GlobalRoot);             // Register the cell as a root
  LocalRoot->next = new(MyHeap) cell;         // Allocates some new cells
  GlobalRoot.next = new(MyHeap) cell;
  MyHeap->collect();                          // The collector will identify
                                              // any allocated cell, starting
                                              // traversing from cell LocalRoot
                                              // and GlobalRoot
  MyHeap->roots.unsetp((GcObject **)&LocalRoot); // Deregister the local root.
}
```

9 Implementing Heaps

This section illustrates the CMM programmer interface for implementing new Heaps. We describe the mechanism through an example, which is a simplified version of the actual Heap used in PoSSo.

9.1 The HeapStack

A foremost algorithm in the PoSSo algebra system is the one for computing of 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 use of a garbage collector was essential. The main step of the Buchberger algorithm [Buchberger 85] consists in the simplification of a polynomial which involves many operations creating a lot of 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 try out 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 called synchronously after each step.

We present a simplified solution in which the size of the stack is fixed, and a copying collector which uses two areas. The real solution we adopted for the problem is more complex and uses a list of areas, and a copying collector.

9.2 The HeapStack

First we define the `HeapStack` class as a `Heap` 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 Heap
{
  public:
   void scavenge(GcObject **ptr);
   GcObject* allocate(int words);
   void reclaim(GcObject* ObjPtr) {};
   void collect();
   HeapStack(int size = 100000);

  private:
   pages FromSpace, ToSpace;
   int FromTop, ToTop;
};

HeapStack::HeapStack(int StackSize)
{
   FromSpace = allocate_pages(StackSize, index);
   ToSpace = allocate_pages(StackSize, index);
}

inline GcObject* HeapStack::allocate(int size)
{
   int words = BYTEStoWORDS(size);
   int *object = FromSpace + FromTop;
   if (words <= (FromSize - FromTop)) {
      FromTop += words;
      return (GcObject *)object;
   }
   else return (GcObject *)NULL;
}
```

The collector uses the root set to traverse the roots using its traversing strategy. After having moved to `ToSpace` all the objects reachable from the roots, it traverses those objects in order to move all further reachable objects. The specific action required for scavenging objects is as follows:

```
void HeapStack::scavenge(GcObject **ptr)
{
    GcObject **OldPtr = ptr;

    if (OutsideHeap((int *)*ptr))
       return;
    GcObject *p = GetBeginning((int *)*ptr) ;
    if (outside(p))
       visit(p);
    else if (*ptr->forwarded())
       ToBeForwarded(ptr);
    else {
       *ptr = moveTo(ToSpace, *ptr);
       OldPtr->SetForward(*ptr);
    }
}
```

This code relies on support provided by classes `GcObject` and `HeapStack`. As the final step the collector exchanges the roles of `FromSpace` and `ToSpace`.

```
void HeapStack::collect()
{
   pages *TmpSpace;
   GcObject *ObjPtr;
   // First traverse the objects registered as roots, applying our scavenge
   ScanRoots(this);
   // Now traverse the objects already moved into ToSpace
   ObjPtr = ToSpace;
   while (ObjPtr < ToSpaceEnd) {
      ObjPtr->traverse(this);
      ObjPtr = ObjPtr->next();
   }
   // swap FromSpace and ToSpace
   TmpSpace = FromSpace; FromSpace = ToSpace; ToSpace = TmpSpace;
   FromTop = ToTop; ToTop = 0;
}
```

The `roots` for `HeapStack` can be set or deleted using the Heap member function register and unregister. In the case of the Buchberger algorithm we register two global variables containing the `Base` of polynomials and the list of polynomial pairs which are the only objects which need to be preserved after each simplification step:

```
  HeapStack BBStack;
  Base b;
  Pairs p;
  ...
  main() {
    BBStack.register(b);
    BBStack.register(p);
    ...
       BBStack.collect()
    ...
    BBStack.deregister(b);
    BBStack.deregister(p);
  }
```

10 Related Work

The Boehm-Weiser collector [Boehm 88] is a well known collector for C++ which is convenient to use since it is totally conservative. However is not customisable and is subject to unduly retention of space and memory fragmentation since it cannot compact memory. Our copying collector has some advantage in performance not having to reconstruct a free list after collection and being more accurate in tracing live objects.

Work on adding garbage collection to C++ has been done by Dain Samples and Daniel Edelson. Samples [Samples 92] 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 as soon as possible. Our feeling is that this work demonstrates how the flexibility of object oriented languages can be used to implement a very complex environment, like CMM, without requiring modifications to the language.

Edelson [Edelson 92] has been experimenting with the coexistance of different garbage collection techniques. The flexibility of the solutions he adopts in his approach allows the coexistance 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 [Ellis 93] 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. They also propose to change the operator `new T` to call the collector allocator when `T` is a `gc` type, and as a consequence of this, the overloading of `new` and `delete` operators for `gc` classes is forbidden. While the `gc` keyword is compatible with our solution of inheriting from the base class `GcObject`, the constraint on `new` needs to be relaxed to allow overloading of `new` when additional arguments are present. Otherwise this constraint will block the possibility of using different zones for the same kind of objects in different portions of a program. Other suggestions from the Ellis-Detlefs proposals are quite valuable, 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.

11 Conclusion

The CMM offers to programmers garbage collection facilities without significant compromises. They can use a generic collector, a specific collector or no collector at all, according to the need of each algorithm. The algorithm can be in control when necessary of its memory requirement and does not have to adapt to a fixed memory management policy.

The CMM is implemented as a C++ library, produced with extensive revisions from the original Bartlett's 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.

The next challenge would be to incorporate in the compiler the minimal facilities required for CMM support: the addition of the `gc` keyword, proposed by Ellis and Detlefs, could facilitate this.

12 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`.

13 Acknowledgements

Carlo Traverso and John Abbott participated in several meetings and provided valuable feedback on the design. Joachim Hollman provided useful comments on the first implementation. J.C. Faugere provided the idea for this work, adopting a specific memory management in his implementation of the Buchberger algorithm. Discussions with J. Ellis where useful to ensure compatibilty of his proposal with our framework.

References

[Bartlett 88] Joel F. Bartlett "Compacting garbage collection with ambiguous roots" Tech. Rep. 88/2, DEC Western Research Laboratory, Palo Alto, California, February 1988.

[Bartlett 89] Joel F. Bartlett "Mostly-copying collection picks up generations and C++", Tech. Rep. TN-12, DEC Western Research Laboratory, Palo Alto, California, October 1989.

[Boehm 88] H.-J. Boehm and M. Weiser "Garbage collection in an uncooperative environment", Software Practice and Experience, 18(9), 1988, 807-820.

[Breuel 92] Thomas M. Breuel "Personal communication", October 1992.

[Buchberger 85] B. Buchberger, "Gröbner bases: an algorithmic method in polynomial ideal theory", *Recent trends in multidimensional systems theory*, N. K. Bose, ed., D. Reidel Publ. Comp. 1985, 184–232.

[Detlefs 92] D. L. Detlefs, "Concurrent garbage collection for C++", CMU-CS-90-119, School of Computer Science, Carnegie Mellon University, 1990.

[Edelson 92] D.R. Edelson "Precompiling C++ for garbage collection", in *Memory Management*, Y. Bekkers and J. Cohen (Eds.), Lecture Notes in Computer Science, n. 637, Springer-Verlag, 1992, 299-314.

[Edelson 92b] D.R. Edelson "A mark-and-sweep collector for C++", Proc. of ACM Conference on Principle of Programming Languages, 1992.

[Ellis 93] J.R. Ellis and D.L. Detlefs "Safe, efficient garbage collection for C++", Xerox PARC report CSL-93-4, 1993.

[Samples 92] A.D. Samples "GC-cooperative C++", Lecture Notes in Computer Science, n. 637, Springer-Verlag, 1992, 315-329.

[Wentworth 90] E. P. Wentworth "Pitfalls of conservative garbage collection", Software Practice and Experience, 20(7), 1990, 719-727.

[Wilson 92] P.R. Wilson "Uniprocessor garbage collection techniques", in *Memory Management*, Y. Bekkers and J. Cohen (Eds.), Lecture Notes in Computer Science, n. 637, Springer-Verlag, 1992, 1-42.