INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

GC^2: A Generational Conservative Garbage Collector for the ATerm Library

Pierre-Etienne Moreau — Olivier Zendra

N° 4548

Septembre 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4548--FR+ENG

INRIA
LORRAINE

GC^2: A Generational Conservative Garbage Collector for the ATerm Library

Pierre-Etienne Moreau* , Olivier Zendra†

Thème 2 — Génie logiciel
et calcul symbolique
Projets Protheo et Miró

Rapport de recherche n° 4548 — Septembre 2002 — 28 pages

Abstract: The ATerm Library is a well-designed and well-known library in the term rewriting community. In this paper, we discuss the current garbage collector provided with the library and stress the fact that some peculiarities of this functional library could be taken advantage of by the memory management system. We explain how we designed and implemented GC^2, a new mark-and-sweep generational garbage collector for the ATerm Library that builds upon these peculiarities. Experimental results on various programs validate our approach, and show that the performance of our new algorithm is very good.

Key-words: ATerm library, memory management, garbage collection, generations, mark-and-sweep

* Pierre-Etienne Moreau est membre du projet Protheo de l'INRIA - Lorraine
† Olivier Zendra est membre de l'avant-projet Miró de l'INRIA - Lorraine

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

GC^2: Un ramasse-miettes générationnel conservatif pour la bibliothèque ATerm

Résumé : La bibliothèque ATerm est une bibliothèque très bien conçue et très connue dans la communauté de la réécriture de termes. Dans ce document, nous dissertons sur le ramasse-miettes actuellement fourni avec cette bibliothèque et nous soulignons le fait que certaines particularités de cette bibliothèque fonctionnelle pourraient être avantageusement prises en considération par son système de gestion mémoire. Nous expliquons comment nous avons conçu et implanté GC^2, un nouveau ramasse-miettes générationnel à marquage-balayage pour la bibliothèque des ATerms qui se base sur ses spécificités. Les résultats expérimentaux sur divers programmes valident notre approche et montrent que les performances de notre nouvel algorithme sont très bonnes.

Mots-clés : bibliothèque ATerm, gestion mémoire, ramasse-miettes, générations, marquage-balayage

Contents

1 Introduction **4**

2 The ATerm Library **5**
2.1 The ATerm data type 5
2.2 Maximal sharing 6
2.3 Memory management: a mark-and-sweep algorithm 6
2.4 Possible improvement: a generational algorithm 8

3 Generational Garbage Collection **9**
3.1 Principles of generational garbage collectors 9
3.2 Generational garbage collection in copying collectors 9
3.3 Few generational non-moving collectors 13

4 Generational Garbage Collection in the ATerm Library **13**
4.1 Main algorithm 14
4.2 Tenuring policy 15
4.3 Memory allocation 16
4.4 Discussion 17

5 Experimental results **18**
5.1 Execution speed 19
5.2 Memory usage 23

6 Conclusion and future work **25**

1 Introduction

The W3C presented the Extensible Markup Language (XML) as the universal format for structured documents and data on the Internet. Similarly, in the Algebraic Specification community, the Term abstract data type can be seen as the universal format to represent programs, functions and data.

In the W3 community, the Document Object Model (DOM) is a platform- and language-neutral interface that allows programs to dynamically access and update the contents, structure and style of XML documents. Similarly, the Annotated Terms data type (ATerms) is a platform- and language-independent interface that allows programs to represent and manipulate tree-like data structures.

Designed and implemented at CWI (`http://www.cwi.nl`), the ATerm Library [vdBdJKO00] is a concrete implementation of the ATerm data type whose main characteristics can be introduced as follows:

- the ATerm Library is platform- and language-independent: data structures can be created and manipulated in any suitable programming language;
- the ATerm Library uses a compact representation of tree data structures: maximal sharing, also known as hash-consing, is used to share pieces of data that are structurally equivalent;
- the ATerm Library is efficient: the implementation is as efficient as the best known ad-hoc implementation of tree data structures

A consequence of this very good technical work is that the ATerm Library is widely used by several research groups working in different fields: compiler construction, software renovation, program transformation, term rewriting systems, etc. The ATerm Library is usually used to represent and transform tree-like data structures such as parse trees, abstract syntax trees, parse tables, generated code, and formatted source texts for example. The main applications involved include parsers, type-checkers, compilers, formatters, syntax-directed editors and software renovation tools.

From a user point of view, the ATerm Library has at least three major advantages: it is efficient, large terms (several millions of nodes) can be represented, memory management is transparent (a garbage collector [Wil92, Wil94] is included). Another advantage is related to the textual representation which allows exchanging data between programs written in different languages, running on different platforms. We do not insist on this aspect, simply because the focus of this paper is more the implementation of the library than its design.

From an implementation point of view, the ATerm Library consists in relying on a maximal sharing approach to represent identical terms, and using a conservative mark and sweep garbage collector to reclaim unused memory cells. In this paper, we present a new garbage collection algorithm that significantly improves the efficiency of the original mark and sweep garbage collector of the C version of the ATerm Library. The main characteristic of this new garbage collector is that it improves efficiency by using a *generational* approach. Generational garbage collection is based on the assumption that most objects only live a very short time, while a small portion live much longer. By trying to reclaim newly allocated objects more often than old objects it is expected that the collector will work more efficiently.

An essential characteristic of the ATerm Library is that it contains a garbage collector which does not need any help to find all the potential roots. Thus, only a conservative, non-moving garbage collector may be used to reclaim the memory. The main contribution of this paper is to present a conservative generational algorithm which does not introduce any overhead with respect to a classical non-generational conservative mark and sweep approach. Heap and objects are divided into two generations, and a pure functional approach is exploited to avoid inter-generation references. As a consequence, the main contribution of this work is a real improvement of the garbage collector used in the ATerm Library, and a significant overall improvement of the efficiency of the ATerm Library.

The remainder of this paper is organized as follows. First, the main characteristics of the ATerm Library and its mark-and-sweep garbage collector are presented in section 2. Then, section 3 explains the general principles of generational garbage collection. In section 4, we show how we took advantage of the characteristics of the ATerm Library to design an interesting and efficient conservative generational garbage collector. Experimental results are presented and discussed in section 5. Finally, section 6 concludes and points out a few directions where further improvements seem possible.

2 The ATerm Library

In this section presenting the ATerm Library, we focus on the representation of terms and current memory management, so as to show the situation we started from, when designing our new GC² memory management algorithm.

2.1 The ATerm data type

The ATerm data type consists of seven basic constructors:

- `INT`: an integer constant is an ATerm
- `REAL`: a real constant is an ATerm
- `APPL`: a function application consisting of a function symbol and zero or more arguments (ATerms) is an ATerm
- `LIST`: a list of zero or more ATerms is an ATerm
- `PLACEHOLDER`: a placeholder term containing an ATerm representing its type is an ATerm
- `BLOB`: a "blob" (Binary Large data OBject) containing a length indication and a byte array of arbitrary binary data is an ATerm
- an annotation may be associated to every ATerm

The operations on ATerms fall into three categories: creation and matching, reading and writing, and annotating. Only 13 functions provide enough functionality for most users to build simple applications with ATerms.

An important issue the designers and implementers of the ATerm Library had to solve was how to represent the ATerm data type in such a way that all operations could be performed efficiently, without using more memory than necessary.

Every ATerm object is stored in three or more machine words. The first word is used to store an object header and some information such as the size of a list for example. The second word is used to link cells in the collision list of the hash table. The following words are used to store references to objects or values such as integers or reals. In the original implementation of the ATerm Library, the first byte of the first word of each object is used to store the header. This header consists of three fields:

- the first bit is a mark flag used by the garbage collector
- the second bit indicates whether or not this term has an annotation
- the three following bits indicates the type of the term (`INT`, `REAL`, `APPL`, etc.)
- the last three bits represent the arity of this object (the number of references to other terms). When this field contains the maximum value of 7, the term is a function application and the actual arity can be found in a table associated to the function symbol.

The information contained in the header is used by the garbage collector to reclaim memory and to improve efficiency. For example, when an `INT` has to be marked, the two following words are not marked because it is known that they do not store references to others objects.

From an implementation point of view, several issues have to be considered. In particular, we will briefly explain how the ATerm Library designers ensure that two identical terms are represented only once in memory, and how they ensure that an unused node can be reclaimed when necessary.

Note that in the remainder of this paper, we consider the C implementation of the ATerm Library, because this the only existing implementation where a garbage collector is relevant. In the Java implementation, for example, the garbage collector used to reclaim unused memory is simply the one provided by the JVM itself.

2.2 Maximal sharing

In order to minimize memory footprint, a simple but efficient strategy is used by the ATerm Library to ensure uniqueness. Only *new* terms are created: when a term to be created already exists, that term is reused to ensure the maximal sharing. This technique is also know as "hash-consing" or "aliasing".

In addition to allowing low memory usage, maximal sharing also has very significant positive effects on speed. First, the lowered memory usage increases the program locality, decreases the potential page faults and even on-disk swapping when the memory pressure is high in the system. Second, relying on maximal sharing and ensuring it makes it possible to compare terms with mere pointer comparisons, instead of — potentially deep — structural comparisons, which can have a very significant impact on performance. As a comparison point, in [ZC01] we report a more than 10% speedup in an Eiffel compiler caused just by aliasing character strings (for a 14% memory footprint reduction).

In practice, maximal sharing of terms is ensured at creation time by checking whether a particular term already exists or not, before creating it. The technique thus consists in searching through all existing terms before building any term. For obvious efficiency reasons, the terms are stored in a hash table.

Given a function symbol and the arguments, a hash code (an integer) is computed to start the search in the hash table. To this code is associated a simple linked list which contains all the terms that have this same hash code. So, searching a term consists in scanning a simple linked list (which may be empty), and comparing the function symbol and the arguments to those of each element of this list. The comparison is done using a pointer equality check: by construction, symbols and subterms are shared and unique. If the "new" term already exists, it is returned. Otherwise, it is created by allocating a memory cell to store the function symbol and the subterms, and inserting this freshly created term into the associated list.

When a term to be constructed is not found in the hash table, a memory cell thus has to be allocated. The size of this cell depends on the type of the ATerm and, when it is an `APPL` term, on the arity of the top function symbol. In order to improve efficiency, the C implementation of the ATerm Library maintains one list of free cells for each possible size. When a cell is needed but the corresponding free list is empty, the memory manager has two possibilities: it can either allocate a new memory chunk (or *block*) and refill the corresponding free list, or decide to reclaim unused cells. In the C implementation, the ATerm Library integrates a garbage collector to automatically recycle unused space. Therefore, in addition to the common high level term manipulation interface, another advantage offered by the ATerm Library is to free the users from reclaiming unused terms.

We remind the reader that this paper is not a detailed description of the ATerm Library. In particular, we do not discuss any design or technical choice, such as "direct or indirect hashing" for example. For more details, the reader in invited to refer to the original presentation of the ATerm Library [vdBdJKO00].

2.3 Memory management: a mark-and-sweep algorithm

The most common kinds of algorithms to reclaim memory are reference counting algorithms, copying algorithms and mark-and-sweep algorithms. Each of them has its advantages and its drawbacks. See [Jon96] for a detailed comparison.

The first implementation of the ATerm Library [vdBdJKO00] is based on a mark-and-sweep algorithm because it can "easily" be implemented in C without support from the programmer or the compiler [BW88, Boe93].

The *mark-and-sweep* algorithm (also sometimes called *mark-scan*) was the first automatic memory management algorithm developed [McC60]. It is still widely used and has many variants.

It consists in traversing the set of data (objects, cells, data structures, terms in the ATerm Library, etc.) which is *live*, or active, and to mark them in order to make it possible to spot *dead*, inactive objects, whose memory can be freed or recycled. More precisely, simple mark-and-sweep is a two-phase algorithm: first, *marking*, which traverses the set of live objects (or live set), then *sweeping*, which reclaims unused ones.

A boolean flag is associated to each object that indicates whether the object is live (has just been marked) or not. Each time an object has to be allocated, the memory management system checks it can find the necessary memory in its list of free memory cells (free list). If the free list can provide a cell, the algorithm uses it to create the requested object, initializing its mark flag to false — unmarked — and the user program

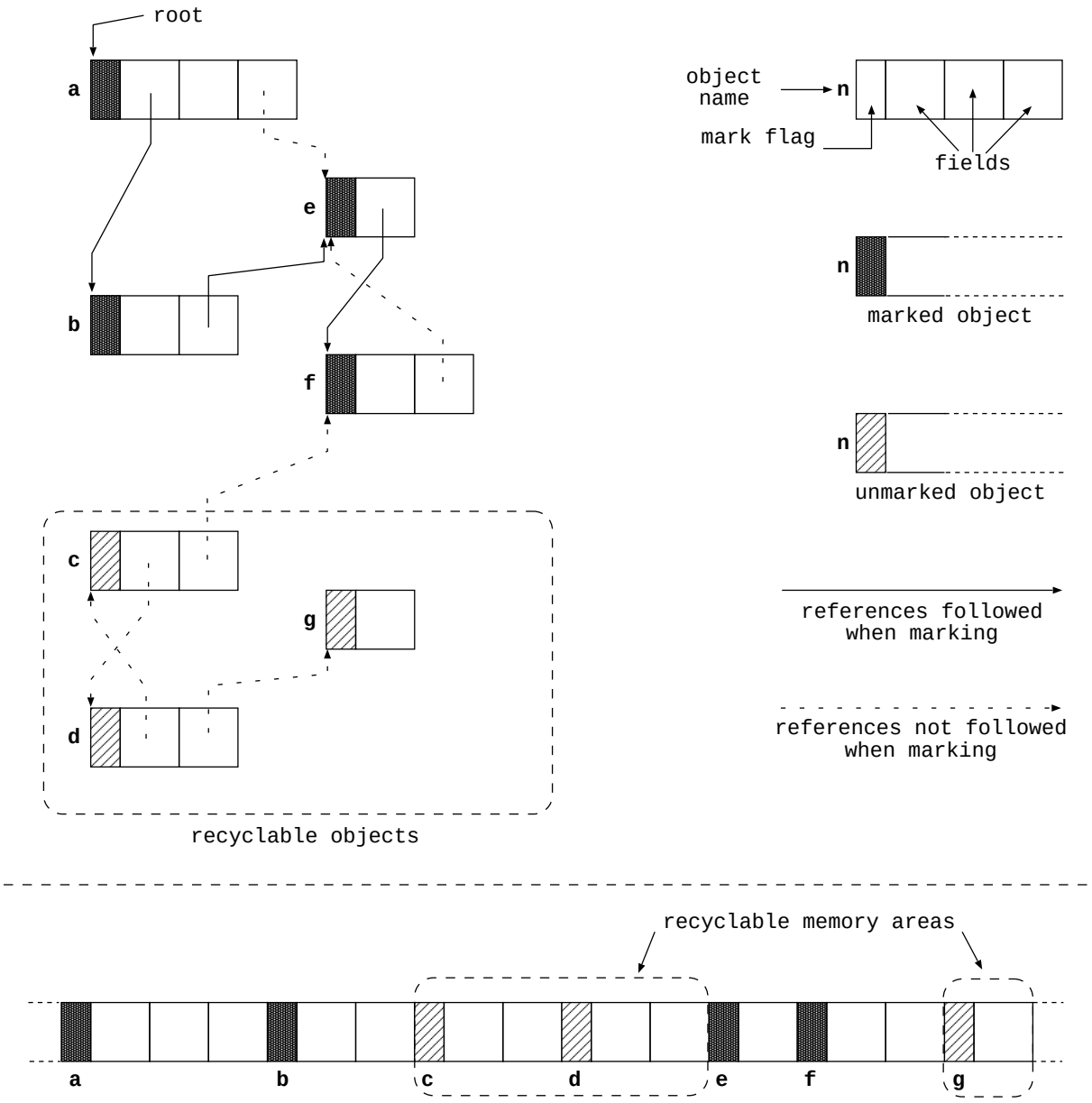

Figure 1: Mark-and-sweep.

Top: Object graph after the mark phase.
Bottom: Memory layout corresponding to the top graph.

(the *mutator*) continues it execution. When the memory management system is unable to immediately provide the memory, a garbage collection cycle may start, interrupting the mutator's execution, in order to find and reclaim the needed memory.

This cycle executes in two steps: marking and sweeping. Figure 1 page 7 describes this process.

A *root* of the graph of live objects is a reference to an object that is *directly* used by the mutator (in opposition to inter-object references). These root references may be found in processor registers, in the execution stack (C stack) that contains the local and temporary variables, as well as in global variables. In

the ATerm Library, terms can be explicitly protected by the programmer; these terms also constitute root objects.

Marking consists in computing the transitive closure of the live object graph, starting from the set of roots and following references from an object to another. Each object reached is marked *live*, and marking continues from the references found in this object in order to find new live ones. As an example, objects `a`, `b`, `e` and `f` in figure 1 are marked this way. Of course, when a reference points to an already marked object, is it useless to restart marking from this object, since that work has already been done. This is the case with the reference from `f` to `e`, the `f` object having necessarily been marked after object `e`. Once this marking phase has completed, all live objects have been marked, whereas all unmarked objects are *dead*, that is unused by the mutator program, and thus reusable. In figure 1, objects `c`, `d` and `g` are reclaimable because unmarked.

The *sweep* phase then starts. It consists in traversing sequentially the whole memory managed by the garbage collector (see bottom of figure 1) and finding which are the unmarked objects (or cells). The corresponding memory is reclaimed, that is put in a list of free cells that can be reused for new objects allocation. Marked, live objects are unchanged, except for their mark flag which is reset to false (unmarked) so as to prepare the mark phase for the next garbage collection cycle. At the end of the sweep phase, the memory management system allocates the object initially requested by the mutator, by reusing free memory from the freshly reconstructed free list. The execution of the mutator then resumes.

It is clear that this mark-and-sweep algorithm does not reclaim memory immediately, as soon as an object dies (becomes unused), but does it in an asynchronous way, that is to say later, only when a new garbage collection cycle is performed. Thus, the mutator and the collector executions are completely separate, with potentially long pauses when garbage collection occurs. Note that this a significant difference from the reference counting garbage collection algorithm[1], where the collector execution is evenly spread and interleaved with the mutator's.

This mark-and-sweep algorithm has a number of great features that lead it to being used for many diverse systems, such as various language implementations: Miranda [Tur85], Prolog [ACHS88], Lisp [Zor89], C/C++ [BW88], Eiffel [CCZ98], and Java in SUN's JDK [Sun00b].

Its first advantage — although it is not relevant in the context of the ATerm Library — lies in the fact mark-and-sweep handles cycles in the objects graph, and thus data structures with such cycles, in a very natural and unproblematic way. This simplifies the realization of a complete garbage collection system, since mark-and-sweep is capable to cope with all object graphs that occur, be they with or without cycles, whereas a reference counting algorithm requires a backup system to handle those cycles. Furthermore, and most importantly, there is no overhead for pointer manipulations in the mutator.

However, mark-and-sweep has various disadvantages. First, as we already mentioned, it completely stops the mutator during the whole garbage collection cycle (marking and sweeping). Since the duration of collection cycles increases with the mutator's live set, the larger the live set, the lower the share of the processor resources usable by the mutator. This phenomenon is known as *trashing*. It can lead to long collection pauses (time when the mutator is inactive), which may be unbearable for highly interactive or real-time programs. In addition, this algorithm tends to fragment memory, because it chops it in cells (objects) that when reused may themselves be chopped for smaller objects. This can make the mark-and-sweep algorithm unsuitable for long-running applications, for example in servers.

One particularity of the garbage collector used in the ATerm Library is to be *conservative*, in the sense that when looking for the roots of the live object graph, some word on the system-stack (an integer for example) might have the same bit pattern as an object reference. In this case, an unused object can be marked as live and some unused objects may not be reclaimed. Without any help from the compiler, which is the case when using the ATerm Library, this constrains the garbage collector to never move any object (*non-moving* garbage collector).

2.4 Possible improvement: a generational algorithm

Although the current memory management system built in the ATerm Library is quite satisfactory, we think there is still some potential to improve it. Indeed this algorithm, which we just briefly described above, is

[1] For brevity, we do not describe reference counting in this paper. The interested reader can refer to [Jon96] both for a short description or a detailed one.

a classical mark-and-sweep, integrated into the ATERM Library. We think improvements are possible by taking into account a number of peculiarities of this library, in order to design a more specialized garbage collector.

One such fundamental peculiarity is the fact that the ATERM Library does not allow so-called destructive updates: after a term has been created, it may not be updated. Thus, a term may only contain references to other terms *that existed when it was created.* This has a strong implication: a term may only refer to *older* terms. As a consequence, it seems to us that it should be possible to organize objects in the ATERM Library memory management system in a way that takes object creation time or object age into account.

Such a category of garbage collectors considering object ages exist: *generational garbage collectors.* In the following section 3, we describe in more details how such algorithms work, in order to motivate our choice for the new garbage collector we designed and implemented in the ATERM Library (which is explained in details in section 4).

3 Generational Garbage Collection

3.1 Principles of generational garbage collectors

A lot of experiments focusing on the memory behavior of programs have shown that allocated data can be divided in two broad categories: temporary data, which are allocated and then die shortly thereafter, and long-lived, "permanent" data. Furthermore, numerous research works [DB76, FF81, Wil94, Zor89, SP93, Hay91, App92, BZ93] have shown that in most languages the large majority of allocated data structures or objects are temporary data. This tends to validate the intuition of the *weak generational hypothesis* [Ung84], which basically says that "most objects die young".

Consequently, it seemed interesting to separately handle in the garbage collector these two kinds of data. One way to do this is to physically segregate these two categories, putting them is two different *generation* sub-spaces. Thus, the collection algorithm can be triggered frequently on the sub-space holding young data, where the odds of reclaiming memory are the best, whereas it is triggered much more infrequently on old data, whose lifespan is potentially higher. This makes it possible to decrease the overall cost of memory management, by focusing garbage collection on areas where its effectiveness is better.

It is of course generally impossible when data is allocated to know whether it is going to live long or not, except in some special cases where the information is provided explicitly by the developer or computed by some form of flow analysis. As a consequence, a tricky issue in generational algorithms is how to decide to promote data from a young generation to an older one. Many heuristics exist, often based on the size of the object as well as its recent past (number of garbage collection cycles the data survived). Generational algorithms thus tend to be a bit complex to implement, but can lead to very good results.

3.2 Generational garbage collection in copying collectors

Quite logically, these *generational garbage collection* algorithms have been implemented in most cases in *copying garbage collectors.* A copying collector is, in a nutshell, an algorithm that manages two semi-spaces, one active, the other one inactive. Live objects in the active space are copied to the inactive space that then becomes the active space, and so on (see figure 2 page 10).

More precisely, the objects used by the mutator are allocated in the active space which is usually managed as a stack. When an allocation fails because the active space seems full (max stack height reached), a garbage collection cycle triggers, stopping the mutator's execution (see part (1) of figure 2).

This cycle consists in tracing, like in the mark-and-sweep algorithm (see section 2.3 page 6), objects that are still live in the active space (source space) and to copy them — either while marking, or after — in the second, inactive semi-space (destination space), which is itself also managed as a stack (see parts (1) and (4) of figure 2). Since the destination space did not contain any object allocated to the mutator, objects from the source space can harmlessly be copied to the "bottom" of that destination space (also managed stack-like), then upwards, contiguously.

The first live object found, whatever its position in the source space, is thus copied to the bottom of the destination space. The second live object, wherever it is found, is copied into the destination space just

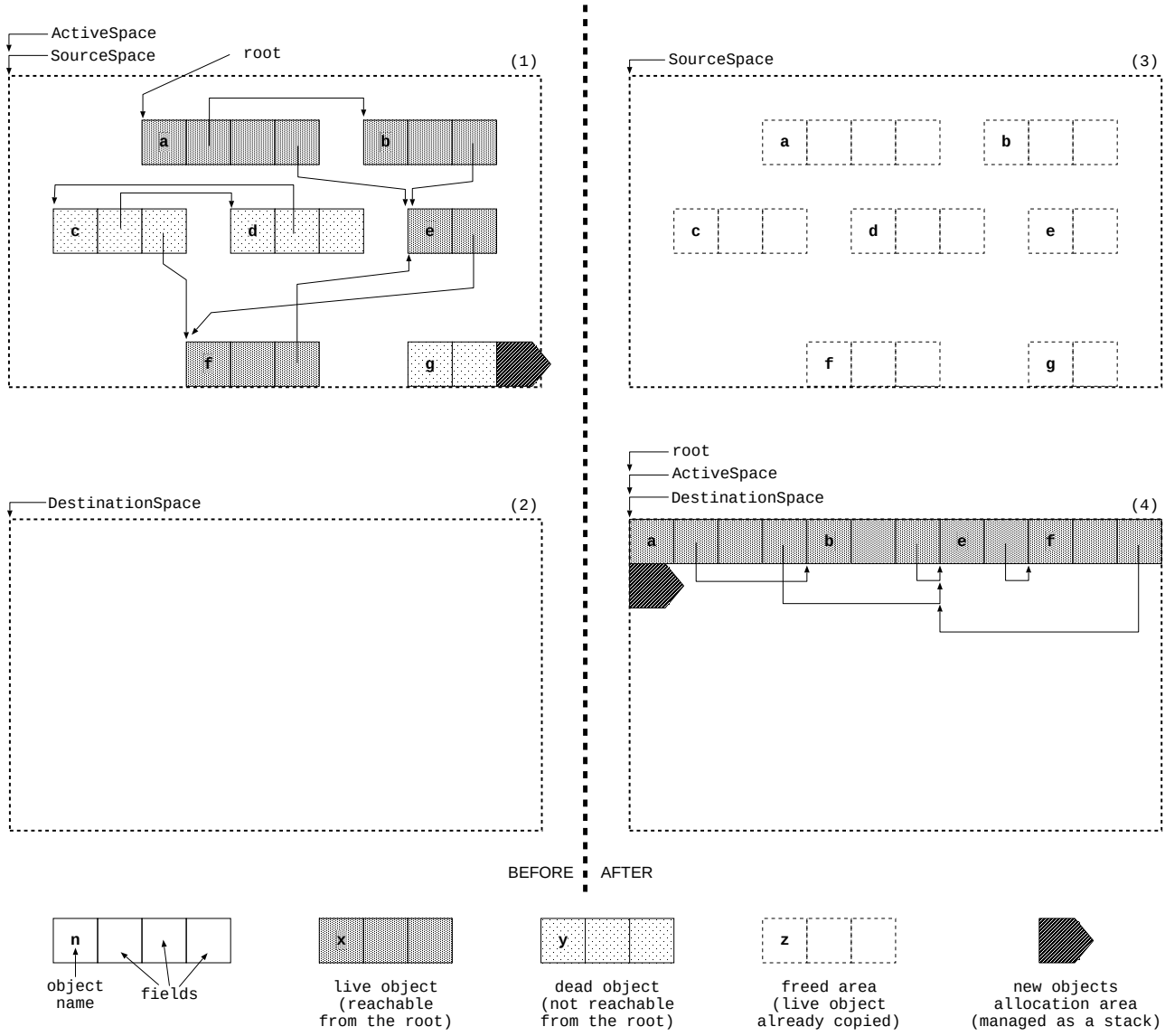

Figure 2: Copying garbage collection.

Left: Memory state just before copy.
Right: Memory state just after copy (pointers `SourceSpace` and `DestinationSpace` not updated yet).

above the previous object, and so on. Therefore, even though live objects are not contiguous in the source space, being interlaced with dead objects, they are organized in a contiguous layout, hence without any waste of space, in the destination space (see part (4) of figure 2).

The latter contains, at the end of the live objects traversal, a copy of the mutator's live objects graph. Obviously, dead objects are not copied. The whole memory corresponding to the space formerly used by these dead objects is thus found above the live objects in the destination space, whereas it was scattered amongst live objects when in the source space. This memory is now immediately reusable, by swapping the roles of the destination and source spaces (i.e. swapping `SourceSpace` and `DestinationSpace` in parts (3) and (4) of figure 2) and restarting memory allocation in a stack-like way from the top of the new active space. It is of course crucial, when copying live objects from the source space to the destination space, to maintain coherent references between objects. The pointers they may hold thus have to be updated so as not to point anymore to objects in the source space but instead to their copy in the destination space. Details of this tricky handling of references for garbage collectors that move objects being of not particular interest in this paper, we shall not present them in depth. Interested readers can find all wanted details in the bibliography, especially [Jon96].

It is quite obvious that these *copying* garbage collectors are very appropriate to implement *generational* garbage collectors, since the concept of semi-spaces matches quite closely the notion of generations.

Figure 3 page 12 illustrates how a simple, two-generation system works. In that figure, references to objects have been omitted for clarity; they are the same as in figure 2. The left part of figure 3 shows the memory state *before* a collection is triggered on the young generation, whereas the right part shows the memory state *after* that collection is finished. Objects in the young generation have ages that are the number of collections cycles the object survived (the ages of objects in the old generation are irrelevant to our explanation). Thus, newly allocated objects are 0 cycle old, those that survived one collection 1 cycle old, etc. Let's assume objects whose age is greater than or equal to 2 are considered as old objects and are thus copied to the old generation, whereas objects younger than this remain in the young generation. In this case, objects `b`, `e` and `f` are copied to the destination space of the young generation, like they were in the figure for a non-generational copying collector (figure 2), whereas object `a` is copied to the old generation since it reaches the old age.

Note that only one semi-space is shown for the old generation in figure 3, because this figure aims at describing a garbage collection cycle on the young generation only (*minor* collection). Of course, when the old generation is garbage collected (*major* collection), it too requires a second semi-space.

The most obvious advantage — which is the *raison d'être* — of copying garbage collectors, hence of generational copying collectors, is the absence of fragmentation, whatever long the execution of the mutator. Indeed, since copying objects implies compacting memory, the fragmentation that potentially appeared when executing the mutator completely disappears during the following collection cycle. Copying collectors are thus considered very reliable, especially for server-like programs. Another significant advantage is the fact that a successful allocation has a very low cost: a simple pointer comparison in order to know whether available memory (between current top of active space and maximum top) is large enough for the to-be-allocated object, followed by an incrementation of the current top by the object size. Furthermore, this cost remains constant even when allocated objects have different sizes or are very large. Finally, since the cost of a copy is proportional to the live set size, this kind of algorithms is very adequate for the numerous systems with a low object survival rate. These advantages of copying garbage collectors makes them very popular for various languages and systems, such as Lisp [Min63], ML [App92], Eiffel [ISE99], Java (HotSpot virtual machine from SUN 1.3 JDK) [Sun00a], etc.

However, copying memory management systems have at least one important and obvious drawback. The memory required by such a system is double the memory needed for the mutator, since it is necessary when collecting garbage to have not only the active space, but also a destination space of same size. Another disadvantage is that keeping pointers up-to-date when objects are moved tend to be complicated and costly, especially for inter-generation references [Jon96]. Also, when considering two garbage collection cycles, the copying algorithm touches *all* the memory areas of the two spaces (active and inactive), that is twice as many areas as in a non copying algorithm. This implies higher risks of delays because of memory page faults. An additional execution time overhead stems from the fact that the locality of the objects touched by a copying

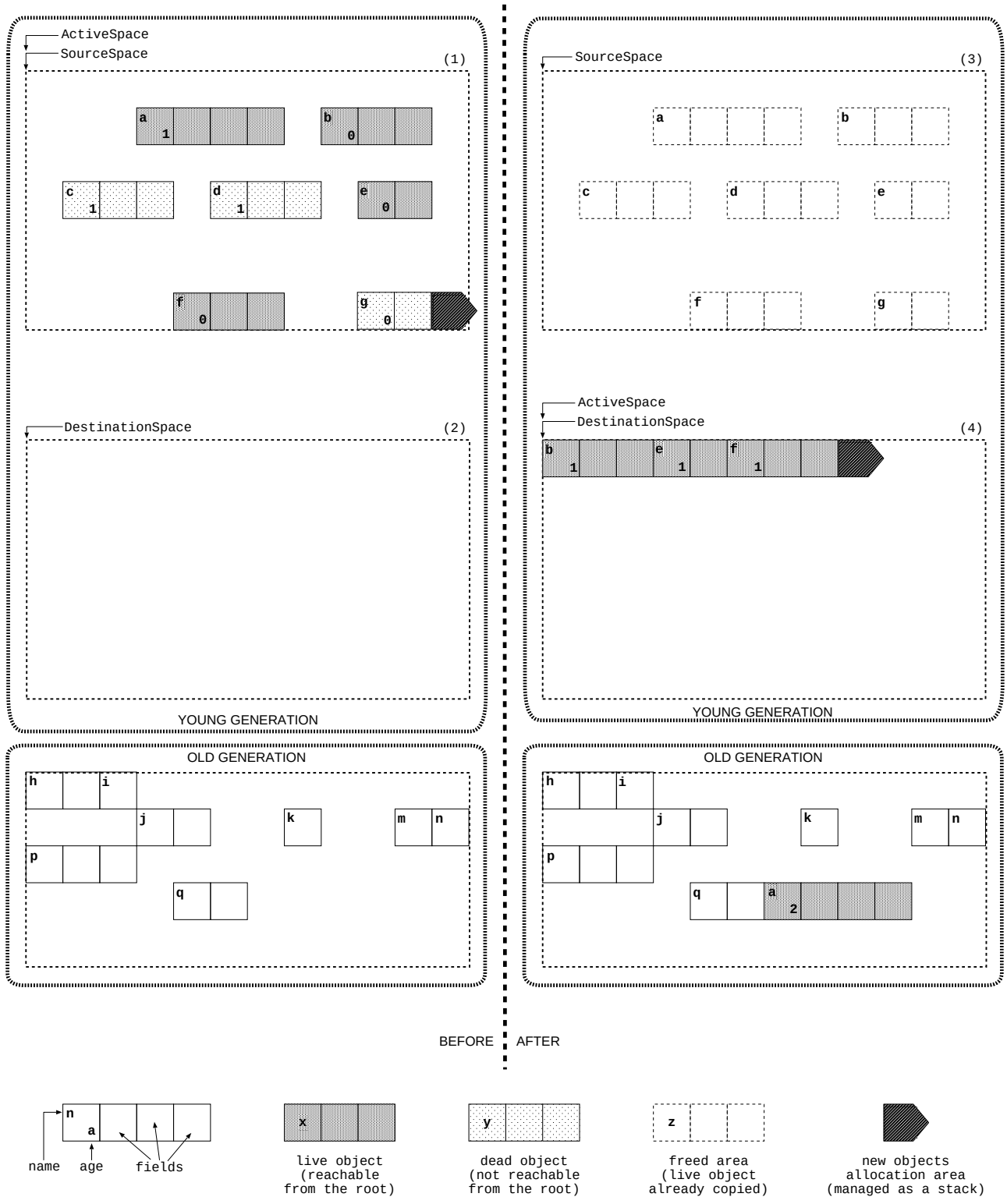

Figure 3: Generational garbage collection (on the young generation)

Left: Memory state just before copy.
Right: Memory state just after copy (pointers `SourceSpace` and `DestinationSpace` not updated yet).

garbage collector is also significantly worse than that of a non-moving collector. These performance problems related to cache misses and page faults are studied in [Zor91], that indicates that copying collectors are in this area significantly less efficient than classical, non-moving, mark-and-sweep collectors. Note that variants of copying collectors, generally called *compacting garbage collectors*, avoid using two spaces by copying live objects from the active space to the beginning of itself. This is of course better with respect to total memory footprint, but can be tricky and costly since yet unprocessed objects may be present where live objects should be copied. We shall not detail compacting collectors here, since they fall beyond the scope of this paper.

3.3 Few generational non-moving collectors

To avoid these big drawbacks of copying generational collectors, it seems interesting to try and design a *non-moving*, generational collector. However, very few of these systems exist, although they can offer pretty good performance, especially memory-wise [Zor89].

The reason for this are a bit unclear to us. We might surmise that the fact mark-and-sweep matches less closely the generational concepts than copying collectors, as indicated in [DWH+90], explains their lower use for generational systems. It is also possible that mark-and-sweep algorithms tend to be viewed as a bit "old-fashioned" garbage collectors, with too many fragmentation problems.

Nonetheless, the relative simplicity of mark-and-sweep collectors, coupled with the fact that the current garbage collector in the ATerm Library is a mark-and-sweep one — and has to be a non-moving one – lead us to decide to go on with GC^2, a new mark-and-sweep, generational collector for the ATerm Library. This decision also made it easier to compare our new collector with the current one, to more clearly evaluate the overall impact on performance, especially because of the generational aspect.

4 Generational Garbage Collection in the ATerm Library

In this section we describe how the characteristics of the ATerm Library can be exploited to design a new and efficient *conservative generational garbage collector*, which we called GC^2.

As mentioned in section 3.2, a difficulty introduced by generational collectors is that inter-generational references can appear. They can be created in two ways: either when storing a pointer into an object or when an object containing references to other objects of the same generation is promoted to an older generation. It is of course essential to keep track of these inter-generational references. The ones created by promotion can be relatively easily tracked. But for those created by assignment, a *write-barrier* is needed to trap and record these pointers. In both cases, this imposes an overhead on the mutator. The originality of the algorithm we designed and present in this paper is to avoid this overhead, by exploiting some of the ATerm Library peculiarities.

More precisely, we take advantage of two fundamental aspects of the ATerm Library :

- A characteristics of the ATerm Library is to guarantee a pure functional style: a term is never modified. Therefore, objects are built in a bottom-to-top way and a newly created object only contains references to older terms.
- If we guarantee the collection of all younger generations whenever we collect an old generation, we only need to record old-to-young pointers.

A consequence of these remarks is that old-to-young pointers can only be created by promoted objects. Therefore, in principle, we only need to take care of these.

However, a finer analysis of the algorithm showed us that it was also possible to cleverly define the promotion policy, in such a way that an object only contains references to older objects. This is crucial, because it allows us to design a specialized generational garbage collection algorithm for the ATerm Library which does not introduce any overhead, unlike other *write-barrier*-based generational garbage collectors.

The next sections describe and discuss the design and implementation of GC^2, our specialized garbage collection algorithm.

4.1 Main algorithm

In order to simplify the presentation and the implementation, only two generations of objects are considered in what follows. Thus, depending on the generation it belongs to, an object can be either *young* or *old*. As explained in section 3.2, generational garbage collection algorithms are usually implemented in copying garbage collectors, but this approach is not possible because the ATerm Library should provide a non-moving garbage collector. Therefore, the presented algorithm is based on a conservative mark-and-sweep collector (like the original implementation of the ATerm Library).

We distinguish two main phases, the mark and the sweep phase, as well as two main generations, the *young* and the *old* one. This leads us to decompose our algorithm into 4 main different steps:

- Minor Mark Phase: all young live objects are marked.
- Minor Sweep Phase: the part of the heap that corresponds to the young generation is swept.
- Major Mark Phase: all live objects are marked, be they young or old.
- Major Sweep Phase: the whole heap (young and old generations) is swept.

According to this general scheme, it appears that we have to be able to distinguish young objects from old ones, as well as the part of the heap that corresponds to the young generation.

The first requirement can be solved by storing the *age* of each object in its *header* (as already explained in section 2.1 page 5, in the ATerm Library, each object has a header that stores some information, for example the mark flag). In order to avoid any misunderstanding, in what follows we will make a distinction between the notion of *generation* of objects and the notion of object *age*. On the one hand, an object has an *age* that corresponds to the "elapsed time" since it was created. In practice, this age is approximated by the number of collection phases the object survived. On the other hand, the heap is divided into a young and an old *generation*: this corresponds to an organizational view of the heap. In theory, there is no strong relation between the notion of age and the notion of generation: a generation can contain young objects as well as old ones. But in practice it is better to limit as much as possible the young generation to the set of young objects, and the old generation to the set of old objects. Indeed, the age is used by the minor mark phase to mark only live young objects, whereas the generation information is used by the minor sweep phase to scan only a subset of the heap.

To fulfill the second requirement — identifying the young generation from the old one — a naive solution could consist in maintaining a list of young objects and a list of old objects (by using the ATerm hash-table for example), and thus mapping the notion of generation to the notion of age. But this is generally not a good idea: for efficiency and pointer identification reasons, it is better to divide memory into fixed size blocks and link these into a list of young blocks and a list of old blocks. The list of young blocks corresponds to the young generation, and the list of old blocks to the old generation

This notion of block is also used to improve the main allocator. Each time a new object has to be created, it is more efficient to give it a part of a pre-allocated block than to call the `malloc` system function. In practice, a block usually corresponds to several kilobytes of memory. Empty blocks are stored in a list of free blocks. When a new block is needed, the collector first tries to allocate a block from this free block list. In case it is not possible, a new block is allocated via a `malloc`.

As in the original implementation of the ATerm Library, our GC^2 memory allocator maintains several lists of free objects, one per size. When a new object has to be created, the allocator first checks whether a corresponding free object is available. If not, some memory has to be allocated or reclaimed. Depending on a heuristics, the system decides either to allocate a new block or to start a mark and sweep cycle. In the latter case, the system also has to choose between a minor and a major collection. During a major collection, all the live objects are marked, and the whole heap is swept, exactly like in the non-generational approach. During a minor collection however, the set of all roots is scanned, but only young objects are traversed. Thus, objects only referenced by old objects are not marked. Then the list of young blocks — the young generation — is swept in order to reclaim unreachable young objects. Note that the promotion of blocks is performed during this minor sweep phase: according to a heuristics, the system may decide to promote to the old generation a block and all the objects stored in it (this promotion policy is also known as "*en masse* promotion").

4.2 Tenuring policy

As mentioned at the beginning of this section, the originality of the GC² algorithm we present is to provide an organization of objects such that no inter-generational old-to-young pointer can appear.

When considering the "real age" of an object (the actual time elapsed since the object creation), the pure functional approach of the ATerm Library guarantees that an object can only contain references to *older* objects. In this context, it becomes clear that no inter-generational old-to-young pointer could ever appear.

However, in practice, for space and efficiency reasons, the "real age" can unfortunately not be used as a reference to divide the heap into an old and a young generation. Several approximations have to be used. When performing "*en masse* promotion", the generation an object belongs to does not necessary reflect the "real age" of this object. This first approximation has to be taken care of, because it can introduce inter-generational pointers, for example when a young object `o1` is promoted to the old generation and contains a reference to another young object `o2` (which remains in the young generation): this creates an old-to-young pointer.

The second approximation is introduced by the age representation. In our current implementation of GC², the age of each object is encoded with 2 bits, with values ranging from 0 to 3. This age corresponds to the number of collections the object has survived since it was created: an age of "2" means that the object has survived 2 collections cycles. In our algorithm, we decided to considered an object as old at age 3. The age of an object is incremented during each collection cycle. In practice, this increment is performed during the mark phase. However, during a minor mark phase, only young objects are traversed, so the age of an old object is not incremented and remains at 3 (old objects do not age anymore). Thus, not incrementing the age of an old object is not an issue.

In the "real age" model, we use the $<_{real}$ order to compare the ages of two objects. This order is the classical precedence defined on natural integers (assuming that a "creation time", a Unix time for example, is encoded by an integer). In our approximation model, many objects may have the same age. So we do no longer compare objects, but rather equivalence classes, with the $<_{approx}$ order. This leads us to make the two following observations:

- If we consider two objects, x and y, such that y has survived to strictly more collections than x ($x <_{approx} y$), we also know that y has been created before x. So, we have $x <_{real} y$.
- Let us now consider two objects x and y, such that y has been created before x ($x <_{real} y$). Let n be the number of collections performed since the creation of y. We can distinguish two cases. First, if $n < 3$, it is clear that x cannot have survived to more than n collections. Second, if $n = 3$ (y is an old object), we have $x \leq_{approx} y$, since the age of y is the maximal element in the $\leq_{approx}$ ordering.

As a consequence, we have the following properties ($<_{approx}$ is *compatible* with $<_{real}$):

$$\begin{array}{lcl} \forall x, y, x <_{real} y & \Longrightarrow & x \leq_{approx} y \\ \forall x, y, x <_{approx} y & \Longrightarrow & x <_{real} y \end{array}$$

Using the above approximation, it becomes clear that no inter-generational old-to-young pointers may occur. To illustrate such cases, let us try to find a counter example: let us consider that two objects o_1 and o_2 exist such that $o_1 <_{approx} o_2$ (o_1 younger than o_2) and such that o_1 is a subterm of o_2. Because $<_{approx}$ is compatible with $<_{real}$, we know that $o_1 <_{real} o_2$. We also know that the pure functional characteristics of the ATerm Library are such that an old object (in the actual elapsed time reference) cannot contains a reference to a younger object. As a consequence, o_1 cannot be a subterm of o_2 and no old-to-young pointer may occur. This first property (no old-to-young pointer) guarantees that the proposed algorithm is correct for the minor mark phase: it is not necessary to traverse an old object to mark live young objects, since the old object cannot reference younger ones.

The second important point to verify is that the proposed algorithm is correct for the minor sweep phase. Since a young block may contain objects of any age, either young or old, and because an old object may be reachable (from other old objects) but not have been marked, we have to ensure that no old object at all is deleted during this minor sweep phase.

To do this, it suffices to modify the minor sweep phase in such a way that old objects belonging to a young block are never deleted. This modification allows us to mix old and young objects in a young block.

A potential drawback of this strategy is that some unused memory (namely, the unreachable old objects) may not be reclaimed during the minor sweep phase. This is not an issue, because they will be collected when the following major collection occurs.

Another interesting point would be the possibility of mixing old and young objects in an old block. However, a particularity of the ATerm Library makes this impossible. Let us for instance imagine two *young objects*, x and y, such that x is a subterm of y, y belongs to an *old block* and x belongs to a *young block*. Let us also imagine that y is no longer reachable and that a minor mark and sweep collection has to be performed. Since x and y are not marked, the minor sweep phase would reclaim object x, but not y because the latter belongs to the old generation (remember that old blocks are not scanned during a minor collection). The dramatic result would be that y would contain an invalid reference (dangling pointer), and that y would still be considered as a live object by the ATerm Library. Indeed, y is still referenced by the ATerm hash-table (which is of course not a root for the garbage collector). It could thus happen that the reference is used internally by the library, for example when the hash-table has to be resized, which would lead to unpredictable but surely incorrect behavior. This counter example shows us that is not possible to allow a young object to belongs to an old block.

To avoid this problem, a solution consists in promoting a block to the old generation only if it exclusively contains old objects; a block containing a young object cannot be promoted. This promotion is performed during the sweep phase, because when sweeping the list of young blocks, it becomes possible to detect and promote blocks which only contain old objects. However, it is not a good idea to promote all the blocks that do not contain any young object: that would allow promoting empty blocks to the old generation, which would probably cause a significant waste of memory. Similarly, the opposite policy, promoting only blocks which are completely full, may be too restrictive, because most of the blocks could remain forever in the young generation.

An intermediate solution seems to be the most interesting in practice. A block can be promoted only if it contains only old objects *and* is "reasonably full". The amount of free space tolerated in a block for promotion to the old generation thus becomes one of the parameters of the algorithm. For example, free space representing less than 25% of the block can be an acceptable ratio. Also note that a block in the old generation may *never* be used to allocate a new object. In one sense, this block memory is frozen until the block becomes completely empty. This empty block is then reclaimed by the collector and put in the list of free blocks, where it can be reused (as a new young block). This strengthens the fact that a block should be promoted only if it is sufficiently full and the objects within it are not supposed to die soon.

4.3 Memory allocation

With respect to the original implementation of the ATerm Library, we have in GC^2 completely re-designed the memory allocation algorithm.

In the original implementation, the principle is quite simple and efficient: when an object has to be allocated and no more memory is available, a block is allocated (using the `malloc` function). According to the size of the object to be allocated, this block is divided into elementary cells which are put into a corresponding free-list. Then, the first free cell is removed from the list and is used to store the object. Similarly, during the sweep phase, each time an object can be garbaged it is removed from the ATerm hash-table and the corresponding memory cell is inserted into the list of free cells.

This algorithm is interesting for its simplicity but it is not optimal with respect to memory fragmentation. Indeed, given a block which contains some free memory, the corresponding empty cells may not be contiguous in the free-list. A consequence of this phenomenon is that the traversal of a term may produce a lot of cache misses or system page loads. Another potential bottleneck of this algorithm may be related to the allocator: each time a new block is allocated, it has to be completely initialized. Furthermore, each time an object is allocated, a cell has to be removed from the free list. The total unitary cost for each cell allocation is 15 assembly instructions[2]. In the remainder of this section, we present our new memory allocation scheme which reduces the memory fragmentation and decreases this unitary cost to 9 assembly instructions only in GC^2.

[2] When compiled with `gcc 2.95.3` with the `-O` optimization option on a Pentium III machine.

The principle of our new memory allocator is quite simple. Each time a new block is needed, it is allocated, either from a recycled free block or with a brand new one created with a `malloc`. The objects are then allocated as in copying collectors, in a stack-like way, by incrementing a pointer.

An advantage of this approach is that it is no longer necessary to initialize a block before using it. Another advantage relates to the fragmentation issue, which can be reduced during the sweep phase. Indeed, with the original algorithm, each time an object can be garbaged it is removed from the ATerm hash-table and the corresponding memory cell is inserted into the associated free list of cells (remember that for each size a free list is defined).

In our new collector, we have a more radical approach: we rebuild all the free lists entirely during the sweep phase. More precisely, at the beginning of the sweep phase, all free lists (one free list per size) are set to empty. Then, when sweeping a block, its free cells are added at the beginning of the corresponding free list. This means that for this block both the previously free cells and those which can be newly reclaimed in this cycle are inserted into the free list. At first sight, this could be seen as an overhead with respect to the previous implementation, because the whole free list for each size is rebuilt at each collection. But in fact this reduces fragmentation, because the free lists are now ordered, with all the free cells belonging to one block being contiguous in their free list. This tends to concentrate new object allocations in the same memory block, instead of scattering them everywhere. This also greatly improves locality within the allocator and within the mutator.

Our approach also makes block promotion very simple. As seen in the previous section, when a block contains only old objects and is reasonably full, it is promoted to the old generation. This promotion is itself a very cheap operation, since it consists simply in removing the block from the list of young blocks and putting it in the list of old blocks. After that, it is no longer allowed to allocated new objects in this block. As a consequence, to make the implementation correct, it is necessary to remove the empty cells that belong to the block from the corresponding free list. In principle, this operation is quite expensive since a quadratic search could be necessary to find all empty cells. With our approach, this search is no longer necessary since all free cells are contiguous. The promotion of a block is performed just after the sweep of that block, and to reclaim free cells we only have to remove the latest inserted cells corresponding to this block. This is done very easily, in little, constant time, by just resetting the free list (head) pointer to the value it had just before sweeping the considered block.

Similarly, when a block is detected to be completely empty, its free cells are removed from the free list and, depending on a heuristics, the block is either returned to the system or inserted into the list of free blocks.

4.4 Discussion

As described in this section 4, the extension of a garbage collection algorithm to a generational one may have a deep impact on the internal data structures of the application.

In the considered case, namely the ATerm Library, the main impact concerns how object headers are encoded. As presented in section 2 page 5, the ATerm data type consists of seven basic constructors. Each of them is represented by an object which has a special header. In the original implementation of the ATerm Library, this header is an 8-bits word which consists of three fields: the first bit is a mark flag used by the garbage collector; the second bit indicates whether or not this term has an annotation; the three following bits indicates the type of the term (`INT`, `REAL`, `APPL`, etc.); the last three bits represents the arity of this objects (the number of references to other terms).

In our new GC² implementation, two bits have been added to the header to store the age of each object (from 0 to 3). Thus, the header is now encoded on 10 bits instead of 8. This modification is not without impact, because on 32-bit machines the 24 remaining bits of the word (the complement from 8 to 32) were used to encode function symbols. With 22 bits instead of 24, the maximum number of possible symbols is reduced to 2^{22}. This restriction also concerns the maximal length of a list which is now restricted to 2^{22} elements. However, this should not be a strong limitation in practice since $2^{22} = 4,194,304$. Furthermore, it will always be possible to allocate an extra 32-bits word when an application needs to manipulate very big lists or a tremendous number of symbols. On 64-bit machines, the impact of our 2 extra bits is just a

non-issue. In the following, we consider that these limitations are not too restrictive and we try to identify what is the impact of the proposed algorithm.

Because of the fundamental properties of the ATerm Library, it was clear that the benefit offered by a generational garbage collector could be considerable. By exploiting the *weak generational hypothesis* [Ung84], we aim at dramatically reducing the time spent marking and collecting objects. However, as reported in the literature [AG93, DWH+90], the generational garbage collection approach is a very nice idea but does not necessary improve the efficiency of the resulting application. Depending on the application behavior, the speedup due the generational approach may be partially canceled by an introduced overhead. In practice, this is mainly due to the need of recording old-to-young pointers when they are created.

As already mentioned, the algorithm presented in this section has a main advantage: compared to a non-generational approach, it does not introduce any overhead. Even with low level considerations (memory fragmentation, system page default, cache memory access, etc.), there is no reason to think that the new algorithm can be slower. The only introduced bottleneck could be related to the more complex heuristics: before each collection cycle the algorithm has to decide whether a minor or a major collection has to be performed. As we will show, the absence of overhead is confirmed by the experimental results of section 5. This can be explained by the simple fact that in the worst case (when the *weak generational hypothesis* [Ung84] is not verified) the generation algorithm has exactly the same behavior as the non-generational one: only major collections occur.

5 Experimental results

In this section, we evaluate the impact of the approach we proposed and show the improvements due to our GC^2 generational algorithm for the ATerm Library. We base our measurements on a collection of representative examples that correspond to typical uses, from very focused examples to large applications.

As explained in [vdBdJKO00], the ATerm Library has already been used in various applications ranging from development tools for domain-specific languages to factories for the renovation of COBOL programs. The ATerm data type is also the basic data type used by different rewrite engines such as ASF+SDF [vdBKO99], ELAN [BKK+96, KM01], TOM [MRV01] and Stratego [JV00] to represent the terms manipulated.

In such systems, the efficiency of the ATerm Library is crucial: every piece of data is represented by an ATerm, and all computations are performed using the ATerm Library. We used two of these systems (ASF+SDF and ELAN) to generate applications that intensively use the functionality of the ATerm Library. This way, we do not restrict ourself to toy examples and we show the impact on real uses of the ATerm Library. These benchmarked applications are:

Knuth-Bendix completion: an implementation of the Knuth-Bendix completion written in ELAN [KM95]. This benchmark consists in completing the p8 problem: a modified version of the Group theory, with 8 identity elements and 8 inverse elements, together with the corresponding axioms. These theories are often used as benchmarks for theorem provers.

N-queens problem: an ELAN implementation of the N-queens problem that searches for a solution with a chessboard of size $n = 12$. This is also a typical benchmark problem in logic programming.

Rewriting: a restricted ELAN interpreter written in ELAN itself. It executes an ELAN program composed of pure conditional rules on an input term, and outputs the result together with a proof term that represents the derivation from the input term.

Primes: an ELAN implementation that computes the list of prime numbers up to 50,000.

Fibonacci: a C implementation of the Fibonacci function, using Peano integers. This benchmark computes the 32^{th} Fibonacci number.

ASF+SDF Compiler: the ASF+SDF to C compiler is written in ASF+SDF itself. This benchmark consists in compiling the ASF+SDF compiler specification using the compiler itself (*bootstrap* process).

All the figures we present in the following pages were obtained by running the benchmarks on a dual Pentium III 800 MHz with 512 MB of RAM under FreeBSD 4.6 and with gcc version 2.95.3.

5.1 Execution speed

Figures 4, 5, 6, 7, 8 and 9 illustrate the efficiency of the ATerm Library on the various examples we just introduced. Each figure contains two graphs that show execution times respectively with maximal sharing (sub-graph "a") and without maximal sharing (sub-graph "b"). Each (sub-)graph compares our new implementation of the ATerm Library ("new" bar) with the current official implementation (version 1.6.6, "old" bar). For each benchmark program, execution times (in seconds) are normalized to the slowest one. Hence lower is better. For each execution, the time spent in the garbage collector (represented by a dark color) is included in the total execution time. The right part of each (sub-)graph represents a zoom on this "dark" area, in order to better detail the behavior of the implemented garbage collector and illustrate the impact of the GC^2 algorithm we propose. This right part shows the times spent respectively in the mark phase (white color) and in the sweep phase (grey).

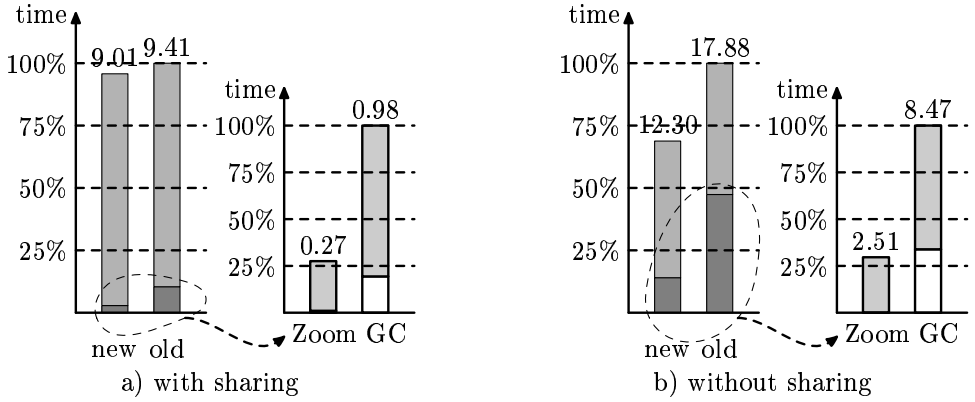

Figure 4: Knuth-Bendix completion

Figure 4 illustrates execution times on the Knuth-Bendix completion example. When using our new GC^2 garbage collector, total execution time *with* maximal sharing (a) decreases from 9.41 seconds to 9.01 seconds. This 4% improvement on time may seem relatively modest. However, less than 10% of the total execution time is spent in the garbage collector. When looking at the "zoomed" part of figure 4a, it is clear that the time spent in sole garbage collector decreases from 0.98 second to 0.27 second. This impressive 72% decrease in time shows that, on the very part of the library — the garbage collector — we improved, our new GC^2 algorithm is 3.63 times as fast as the "old one".

The second part of figure 4, figure 4b, illustrates the execution time of the Knuth-Bendix completion example *without* maximal sharing. In this case, total execution times are, as expected, much worse than with maximal sharing (12.30 second versus 9.01, and 17.88 second versus 9.41). The improvement offered by our garbage collection algorithm, 12.3 seconds instead of 17.88 (31% less time), is larger than before (4%), because the time spent in the garbage collector is proportionally more important. But the "zoomed" part of figure 4b shows that, as before, our new algorithm is 3.37 times as fast as the "old" one, since it improves the time efficiency of the garbage collector by 70% (2.51 seconds instead of 8.47).

Note that the sweep phase (grey part of the bars in the zoomed graphs) is important in this Knuth-Bendix completion benchmark: almost 100% of the garbage collection times in our GC^2 generational algorithm, roughly 75% in the old one. Indeed, the death rate in very high in this benchmark, which causes the live set

(to be marked) to remain relatively small whereas in the sweep phase a lot of work is done to reclaim the dead objects.

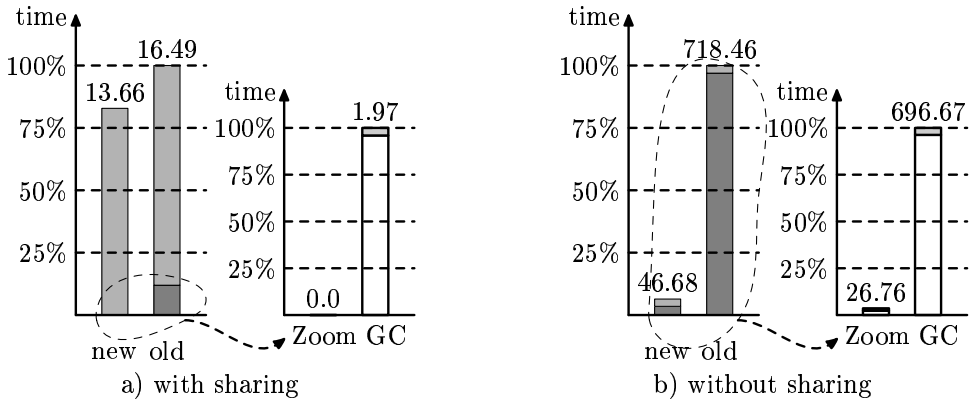

Figure 5: N-queens

The N-queens (figure 5) example is particular in the sense that a lot of backtracking occurs during the computation. In addition to the C system stack, the mark phase also has to scan the backtracking stack to determine the object graph roots. Figure 5 is interesting because it shows that our new GC^2 implementation is faster than the old algorithm.

For the maximal sharing version (figure 5a), our new GC^2 makes the whole program 1.2 times as fast, which corresponds to an execution time improvement of 17%. Unfortunately, this is not because the garbage collector execution is more efficient, but because it is not used at all. Indeed, in this benchmark, internal garbage collector thresholds are such that no garbage collection is done in our implementation[3].

For a reason yet unknown to us, the old implementation of the ATerm Library of the no-sharing version of N-queens (figure 5b) is very slow on this example. Our new algorithm decreases total execution time by 93% (46.68 seconds instead of 718.46), which makes it 15.3 times as fast as the old one. This is mainly because the garbage collection time represents, on this version without sharing, a large part of the total execution time. When only garbage collection time is considered, our implementation appears 26 times as fast as the old one, with our execution time representing only about 4% of the old execution time.

As the zoomed graphs show, almost all the time taken by the old garbage collector on the N-queens benchmark is spent marking (white part of the bars). Since this benchmark makes very heavy use of backtracking, we suppose that the old algorithm does too many misidentifications in the backtracking stack and thus marks a lot of objects that are actually dead. Conversely, since our algorithm has a stricter filtering of addresses, it probably does not suffer from this problem. Indeed, we kept the original filtering of candidate roots from the ATerm Library, with one small but important improvement: we added a min-max barrier at the beginning of the filtering, that rejects all adresses not included between the lowest and the highest valid addresses in the blocks managed by GC^2.

Figure 6 shows the results for the rewriting benchmark. Figure 6a, that corresponds to maximal sharing, shows that both the old and the new garbage collectors represent a negligible part of the total execution time[4].

However, in the no-sharing case of figure 6b, where the garbage collection times are significant, useful conclusions can be drawn. Our new implementation makes the whole program 1.7 times as fast as the old one (18.71 seconds versus 31.73), which corresponds to a 41% decrease in time. When focusing on garbage

[3]One might argue that a non-triggered garbage collector is the best possible one... However thresholds are implementation-dependent and benchmarks execution very sensitive to these in some edge cases — like the N-queens benchmark with maximal sharing. We thus won't claim that our garbage collector is better than the old one on this borderline case benchmark.

[4]Actually no time at all for our new implementation, which indicates the garbage collector did not even start, like in the N-queens benchmark.

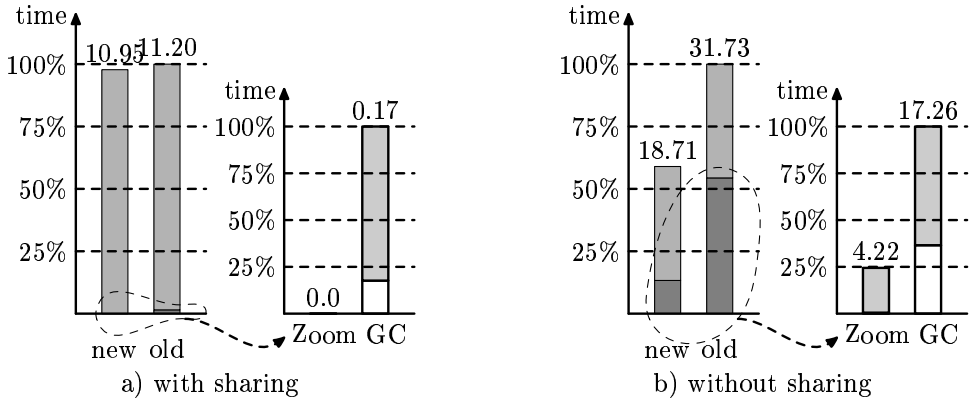

Figure 6: Rewriting

collector only, it becomes clear that our new GC² algorithm is on this benchmark 4.1 times as fast as the old one, allowing a reduction of 75% in time (4.22 seconds instead of 17.26).

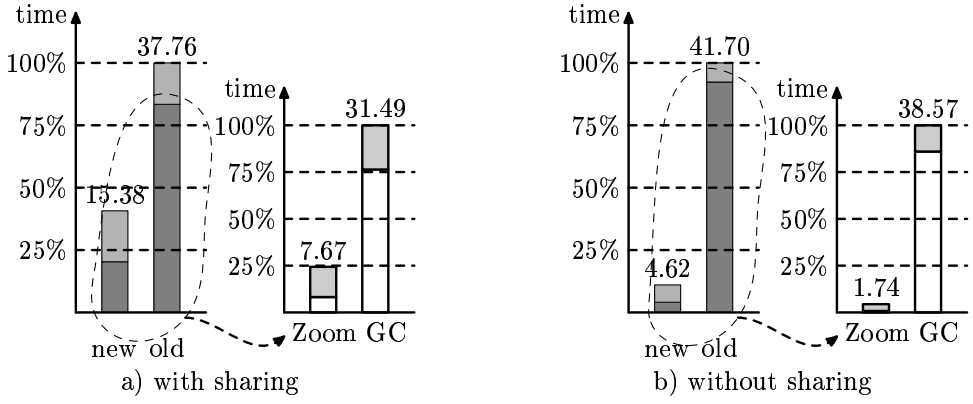

Figure 7: Primes

The primes example (figure 7) computes a list of prime numbers: each new prime number found is added to the end of the current list. In case of maximal sharing (figure 7a), this implies that the list has to be completely rebuild each time. This explains why the garbage collector represents a very large part of the total execution time: 50% in our new implementation (7.67 seconds of 15.38) and 83% in the old one (31.49 seconds of 37.76). This makes this benchmark very different from the three previous ones, where the garbage collection times represented — on the maximal sharing versions — a much more reduced part of the total execution times. The primes benchmark thus puts heavy pressure on memory, even on the generally "reasonable" maximal sharing version. In such cases, our GC² algorithm performs quite well. Indeed, whole program execution with our garbage collector is 2.4 times as fast as the old one (15.38 seconds versus 37.76), which corresponds to a 59% decrease in time. The garbage collection part only is with our algorithm 4.1 times as fast as the old one, with a 76% reduction in execution time (7.67 instead of 31.49).

The large pressure this benchmark puts on the garbage collector is of course even truer in the no-sharing version, in figure 7b. There, garbage collection times represent 38% of the total execution time for our algorithm (1.74 seconds of 4.62) and 92% for the old one (38.57 seconds of 41.70). In this case, it can clearly be said that the efficiency of GC² shines through. Indeed, overall speedup is excellent: the program with our new algorithm is 9 times as fast as the old one, which translates into a 89% time reduction (4.62 seconds compared to 41.70). Speedup when considering only the garbage collection algorithms is even more obvious:

ours takes only 4% of the time of the old one, which corresponds to an execution speed multiplied by 22 (1.74 seconds instead of 38.57).

The zoomed part of the graphs indicates that, for the old garbage collector implementation, at least 75% of the time is spent marking (white part of the bars). We think the explanation for this is the same as for the N-queens benchmark. Primes is a highly recursive benchmark, with a high death rate. Since the old algorithm is not perfect on address filtering, a lot of misidentifications probably occur, causing memory retention and a lot of objects to be marked whereas they are actually dead. This hypothesis is reinforced by the fact the graphs show that, for GC^2, marking represents less than half the time spent in the garbage collector, which seems logical considering our stricter filtering.

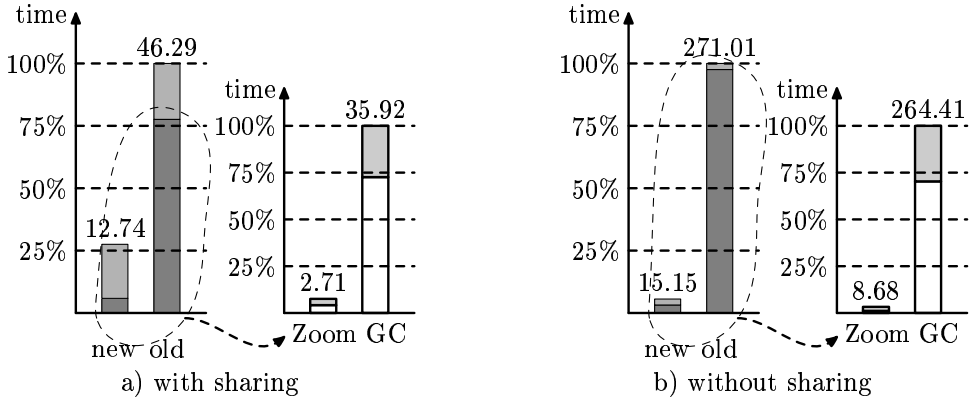

Figure 8: 32^{th} Fibonacci number

Figure 8 illustrates the computation of the 32^{th} Fibonacci number, using a Peano representation of integers. Like the primes benchmark, Fibonacci is very memory intensive, with the garbage collector taking a large part of the execution times, both in the sharing and non-sharing versions. This benchmark thus provides results very similar to those of the primes benchmark (figure 7). Once again, our new generational garbage collector proves efficient, being 13 times as fast as the old one — 2.71 seconds versus 35.92, a 92% decrease in time — on the maximal sharing version (figure 8a) and 30 times as fast — 8.68 seconds instead of 264.41, a 97% time reduction — on the version without sharing (figure 8b).

The zoomed graphs clearly show that marking represents most of the time spent (about 75%) in the old garbage collector. Indeed, Fibonacci is a program with a large live set. The fact that our new generational collector costs much less time, especially marking time, indicates that a lot of the live objects are probably long-lived ones. In such situations, our generational GC^2 algorithm performs of course quite well, compared to a non-generational one, because it can avoid making a lot of the old objects. This hypothesis of a large generation of old objects in Fibonacci has to be confirmed by a more in-depth look at memory. Such a detailed examination is done in the following 5.2 section on memory.

Having presented the very good performance of our new generational garbage collector for the ATerm Library on small benchmarks, we now go on with one last test program. This one is indeed in a way much more significant than the previous ones, because it consists of a very large real world application.

This benchmark, whose results are presented in figure 9, is the ASF+SDF Compiler, a large application that does an intensive use of the ATerm Library. In this application, garbage collection times take a more "representative" part of the total execution time. In the maximal-sharing version (figure 9a), our new algorithm represents 8% and the old one 20% of the total time, while these figures raise respectively to 44% and 88% in the no-sharing version (figure 9b).

When considering only garbage collection, in the maximal sharing version, our generational algorithm improves performance by a factor of 2.8; time decreases by 64%, with 2.56 seconds versus 7.20. In the no-sharing version, this factor of improvement is 3.2 and the execution time reduction 69%, with 37.59 seconds versus 121.16. On the whole program execution, our generational algorithm also has a significant impact,

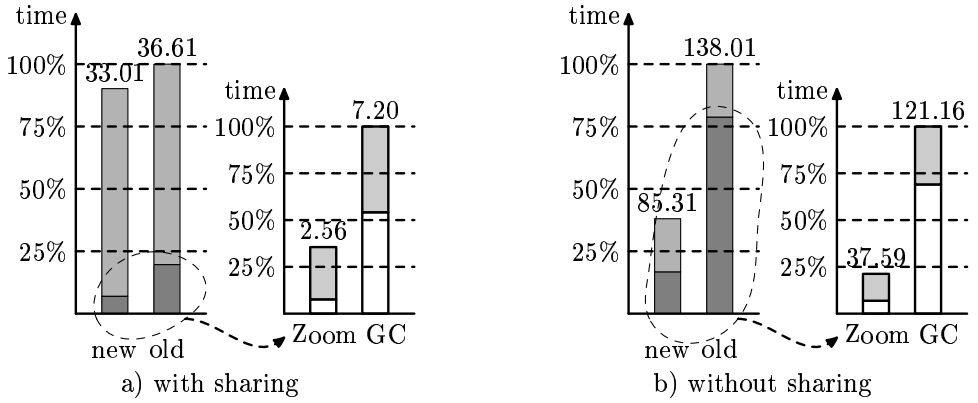

Figure 9: ASF+SDF Compiler

allowing the ASF+SDF Compiler with maximal sharing to take 10% less time, being 1.1 times as fast as the version with the old garbage collector. Without sharing, overall time decreases by 38% with our generational algorithm, which means the compiler has become 1.6 times as fast as the non-generational version.

The zoomed graphs also allow us to better compare the two algorithms and the relative impact of marking (white) and sweeping (grey). In the old garbage collector, marking represents roughly 52% to 70% of the time, whereas sweeping accounts for most of the time spent in our new GC^2 generational collector. This is because the ASF+SDF Compiler handles big terms and features a large set of live objects. It seems safe to assume that this is because of the long-lived objects in the live set. Since the old garbage collector is a non-generational one, it has to mark the whole of this large live-set for all the collection cycles — which are probably numerous because little memory can be reclaimed. On the contrary, our new algorithm, being a generational one, is at its best in this case. Indeed, it marks only the young objects, which are probably not numerous, and does not mark (or even sweep) most of the — many — old objects, hence its large speed advantage. The following 5.2 section offers a deeper look at memory behavior and allows us to check the validity of the hypothesis that old objects are responsible for the performance advantage of our generational garbage collector for this benchmark.

5.2 Memory usage

In this section we focus our attention on memory for two typical examples: the Fibonacci function and the larger ASF+SDF Compiler. We use these examples to better illustrate the behavior of the generational approach we chose for our new GC^2 garbage collector for the ATerm Library.

Figures 10 and 11 show how the memory is managed when running the Fibonacci example with and without maximal sharing. As previously explained, this example computes a big integer represented using "successors". The x axis for these figures indicates the number of allocated objects, and thus kind of represents time. The y axis keeps track of heap size, that is the size of all the objects managed by the garbage collector. For each graph, one curve represents the old generation, that is memory held by live objects in old blocks, whereas the other curve shows the size of the young generation (by cumulating all live objects).

In figure 10, the effect of sharing can been seen clearly: only a few objects die after each garbage collection cycle, and overall memory footprint (live set) increases in a relatively monotonous way. The old generation appears to account for most of the objects in the live set, although the young one also tend to grow. On this example, the generational approach is convincing since most of the objects becomes "old" and a very few of them die: the old generation never decreases. The young generation also grows steadily, although a few objects are reclaimed at each collection cycle. Thus, overall, very few objects are reclaimed and most of them become very old. This confirms the hypothesis we made in the previous section that old objects were responsible for the much better performance of our new GC^2 generational algorithm, since it does not mark (or sweep) this old generation, unlike the old non-generational algorithm.

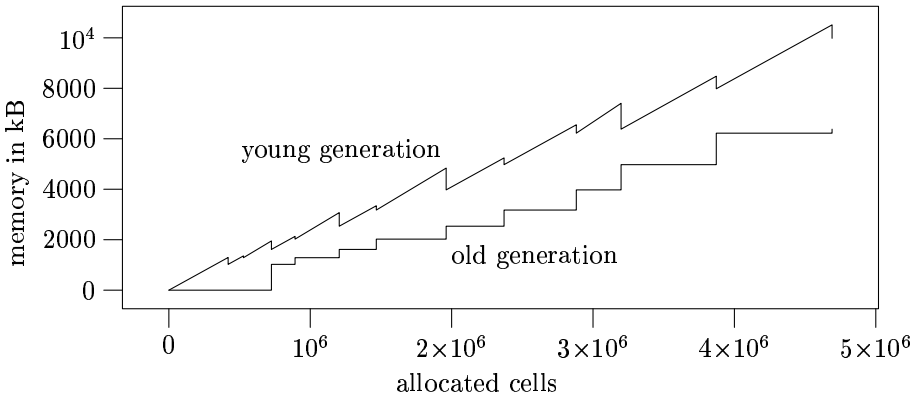

Figure 10: 32^{th} Fibonacci number (with sharing)

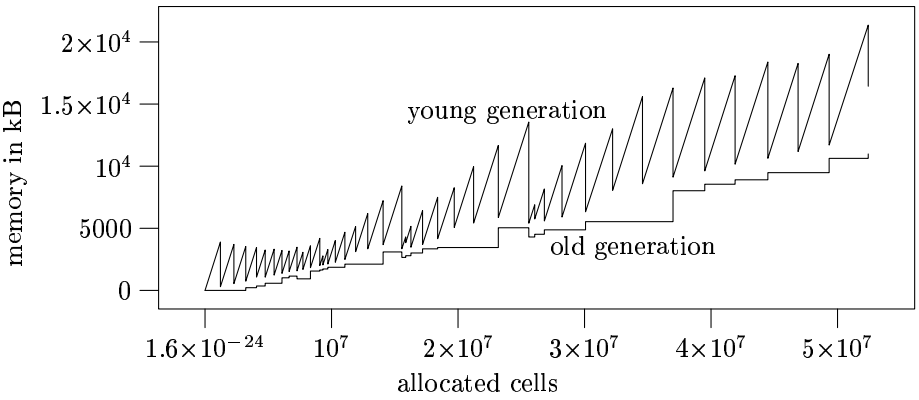

Figure 11: 32^{th} Fibonacci number (without sharing)

Figure 11 also shows an interesting behavior of our generational approach, on the no-sharing version of Fibonacci. There, most of the newly allocated objects die, so the young generation size decreases a lot after each garbage collection cycle. However, some objects do not die and and become old. Once an object is old, the probability of it dying is very low, since the old generation curve rarely decreases. Again, the fact that our algorithm does not spend time on the old generation explains the much better performance it has when compared to the old non-generational garbage collector. The performance gap in even greater in this no-sharing case because, the young generation being efficiently collected, it remains small, and the old generation thus represents an even larger part of the total heap.

Figures 12 and 13 now detail memory behavior for the ASF+SDF Compiler, respectively with and without sharing.

This benchmark is also very interesting. The ASF+SDF Compiler with the version with maximal sharing (figure 12) behaves very much like the Fibonacci benchmark *without* sharing (figure 11). Most of the newly allocated objects in the ASF+SDF Compiler are reclaimed during the following garbage collection cycle, which maintains the young generation rather small, while the old generation grows slowly but steadily. Since the part of the old generation in the total heap grows slowly, the advantage for our generational algorithm, although quite real, remains limited, which corresponds to the good but not exceptional performance speedup presented in figure 9a page 23.

The no-sharing version of the ASF+SDF Compiler benchmark (figure 13) behaves similarly to the maximal sharing version, with one big difference: total memory increases much faster and higher, mostly because of the old generation. Since the young generation remains small, most objects in memory are old ones, which

is a situation very favorable to generational garbage collectors, like the GC^2 algorithm we described in this paper. This translates logically into the very large performance advantage for GC^2 that was shown by figure 9b page 23.

On these two examples, we can notice that, without any knowledge of the mutator (the application being managed memory-wise), the heuristics we used to promote objects to the old generation seems to be validated. Indeed, the old generation never decreases, which shows objects that are promoted actually are old, long-lived objects. Similarly, since the young collection is almost entirely collected and does not grow indefinitely (except, to a limited extend, for Fibonacci with sharing, in figure 10), we can conclude that overall, old objects are all promoted and do not remain forever in the young generation.

On a side note, these examples also illustrate the effect of the maximal sharing offered by the ATerm Library, because the total amount of needed memory is reduced by approximately a factor of 2 in Fibonacci and a factor of 16 (5,000 instead of 80,000) in the ASF+SDF Compiler.

6 Conclusion and future work

In this paper, we presented GC^2, a new algorithm for a generational mark and sweep garbage collector.

The main originality and contribution of this paper are to present a conservative generational algorithm that does not introduce any overhead compared to a non-generational approach. To achieve this result, the algorithm exploits the "pure functional" characteristics of the ATerm Library: a term can be built, but

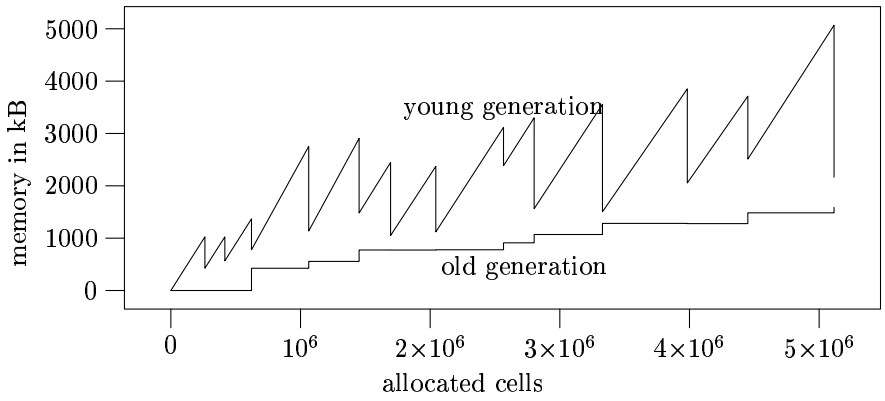

Figure 12: ASF+SDF Compiler (with sharing)

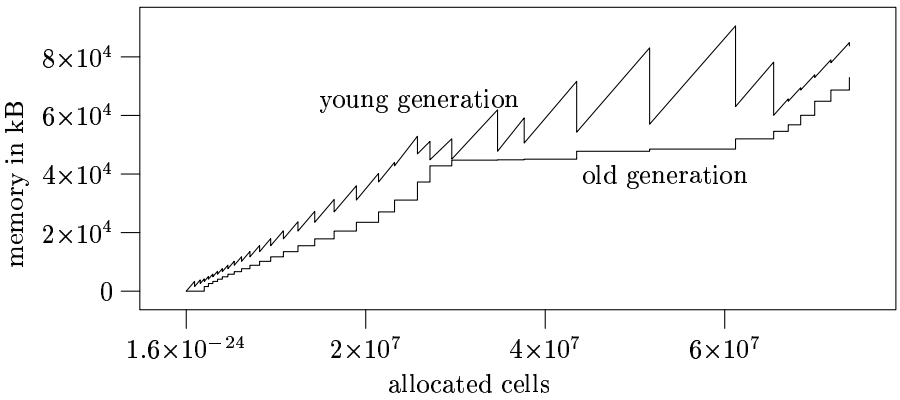

Figure 13: ASF+SDF Compiler (without sharing)

never modified. This particularity allows us to define a generational algorithm that does not create any inter-generational old-to-young pointer, and thus prevents the introduction of any overhead by recording such pointers.

In section 5, we detailed experimental results that clearly show the interest of our approach. Indeed, time spent in our version of the garbage collector is roughly reduced by a factor of 3, when compared to its current non-generational version.

In future work, we plan to study how the proposed approach can be generalized: which are the peculiarities of the ATerm Library that make this algorithm difficult to use in another context? Conversely, can this algorithm be generalized if a particular constraint is relaxed? Answering these two questions is not so easy. A first analysis tends to show that the "pure functional" approach is the key characteristic that make our approach possible. Nonetheless, a generalized version, with a *write barrier* mechanism, may also be interesting, because such term destructive updates are quite rare in term-based applications: the introduced overhead could be negligible in practice.

Acknowledgments

We are grateful to Mark van den Brand for inspiring us the idea of improving memory management in the ATerm Library and for numerous invaluable discussions about the ATerm Library and its use in term-based applications. We are also deeply in debt to Pieter Olivier who supported us by quickly providing accurate technical details about the current, non-generational version of the ATerm Library garbage collector.

References

[ACHS88] Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, 1988.

[AG93] Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Department of Computer Science, Princeton University, February 1993.

[App92] Andrew W. Appel. *Compiling with Continuations*, chapter 16, pages 205–214. Cambridge University Press, 1992.

[BKK+96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, WRLA'96, (Asilomar, Pacific Grove, CA, USA)*, volume 4. Electronic Notes in Theoretical Computer Science, September 1996.

[Boe93] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 197–206, Albuquerque, NM, June 1993. ACM Press.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[BZ93] David A. Barett and Benjamin G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. In *1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, volume 28 of *SIGPLAN Notices*, pages 187–196. ACM Press, 1993.

[CCZ98] Dominique Colnet, Philippe Coucaud, and Olivier Zendra. Compiler support to customize the mark and sweep algorithm. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*,

pages 154–165, Vancouver, October 1998. ACM Press. ISMM is the successor to the IWMM series of workshops.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Daniel G. Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 261–269, San Francisco, CA, January 1990. ACM Press.

[FF81] John K. Foderaro and Richard J. Fateman. Characterization of VAX Macsyma. In *1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 14–19, Berkeley, CA, 1981. ACM Press.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In Andreas Paepcke, editor, *OOPSLA'91 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 26(11) of *ACM SIGPLAN Notices*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[ISE99] Garbage Collection for ISE Eiffel. Technical report, ISE TR-EI-56/GC, version 3.3.9, Interactive Software Engineering, Inc., 1999.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[JV00] Patricia Johann and Eelco Visser. Warm fusion in stratego: A case study in generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 2000.

[KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.

[KM01] Hélène Kirchner and Pierre-Etienne Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[Min63] Marvin L. Minsky. A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, December 1963.

[MRV01] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern-matching compiler. In Didier Parigot and Mark G. J. van den Brand, editors, *Proceedings of the 1st International Workshop on Language Descriptions, Tools and Applications*, volume 44, Genova (Italy), april 2001. Electronic Notes in Theoretical Computer Science.

[SP93] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. In R. John M. Hughes, editor, *Record of the 1993 Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, University of Glasgow, June 1993. Springer-Verlag.

[Sun00a] The Java HotSpot Performance Engine Architecture. A White Paper About Sun's Second Generation Performance Technology, 2000. Sun Microsystems, Inc.

[Sun00b] The Java Tutorial. Learning the Java Language. Object and Data Basics. Cleaning Up Unused Objects., 2000. Sun Microsystems, Inc.

[Tur85] David A. Turner. Miranda — a non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, September 1985. Springer-Verlag.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[vdBdJKO00] Mark G. J. van den Brand, Hayco A. de Jong, Paul Klint, and Pieter Olivier. Efficient annotated terms. *Software-Practice and Experience*, 30:259–291, 2000.

[vdBKO99] Mark G. J. van den Brand, Paul Klint, and Pieter Olivier. Compilation and Memory Management for ASF+SDF. In *Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 198–213. Springer-Verlag, 1999.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Texas, USA, 16–18 September 1992. Springer-Verlag.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[ZC01] Olivier Zendra and Dominique Colnet. Coping with aliasing in the GNU Eiffel Compiler implementation. *Software - Practice and Experience*, 31(6):601–613, May 2001.

[Zor89] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, March 1989. Technical Report UCB/CSD 89/544.

[Zor91] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU–CS–528–91, University of Colorado at Boulder, May 1991.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399