GC2: a generational conservative garbage collector for the ATerm library

Pierre-Etienne Moreau, Olivier Zendra

JLAP 56
29/12/2003 DISK used
ARTICLE IN PRESS
No. of pages: 30 , DTD=4.31

Available at
www.ElsevierMathematics.com
POWERED BY SCIENCE DIRECT®
The Journal of Logic and
Algebraic Programming xx (2003) xxx–xxx

THE JOURNAL OF
LOGIC AND
ALGEBRAIC
PROGRAMMING

www.elsevier.com/locate/jlap

GC^2: a generational conservative garbage collector for the ATERM library

Pierre-Etienne Moreau *, Olivier Zendra

LORIA/INRIA-Lorraine, 615, rue du Jardin Botanique, B.P. 101, 54602 Villers-lès-Nancy Cedex, France

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.

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 [30] is a concrete implementation of the ATERM data type. 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 pars-

* Corresponding author.
E-mail addresses: pierre-etienne.moreau@loria.fr (P.-E. Moreau), olivier.zendra@loria.fr (O. Zendra).
URLs: http://www.loria.fr/∼moreau (P.-E. Moreau), http://www.loria.fr/∼zendra (O. Zendra).

1567-8326/$ - see front matter
doi:10.1016/j.jlap.2003.12.003

ers, 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 [32,33] 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 lives 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 in order to free the developer from the burden of memory management. This collector is conservative, that is, able to find most roots of the object graph by scanning the execution stack and registers, without help from the developer.[1] Thus, only a conservative, non-moving garbage collector may be used to reclaim the memory in the ATERM Library. 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 purely functional style 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 GC[2] memory management algorithm.

[1] Except for protected terms, as mentioned in Section 2.3.

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 four 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 indicate 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 it 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 [34] 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 is invited to refer to the original presentation of the ATERM Library [30].

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 [18] for a detailed comparison.

The first implementation of the ATERM Library [30] is based on a mark-and-sweep algorithm because it can "easily" be implemented in C without support from the programmer or the compiler [9,10].

The *mark-and-sweep* algorithm (also sometimes called *mark-scan*) was the first automatic memory management algorithm developed [21]. 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 are *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 whether 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 (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. Fig. 1 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 Fig. 1 are marked this way. Of course, when a reference points to an already marked object, it is 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 Fig. 1, objects `c`, `d` and `g` are reclaimable because they are unmarked.

The *sweep* phase then starts. It consists in traversing sequentially the whole memory managed by the garbage collector (see bottom of Fig. 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 object 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

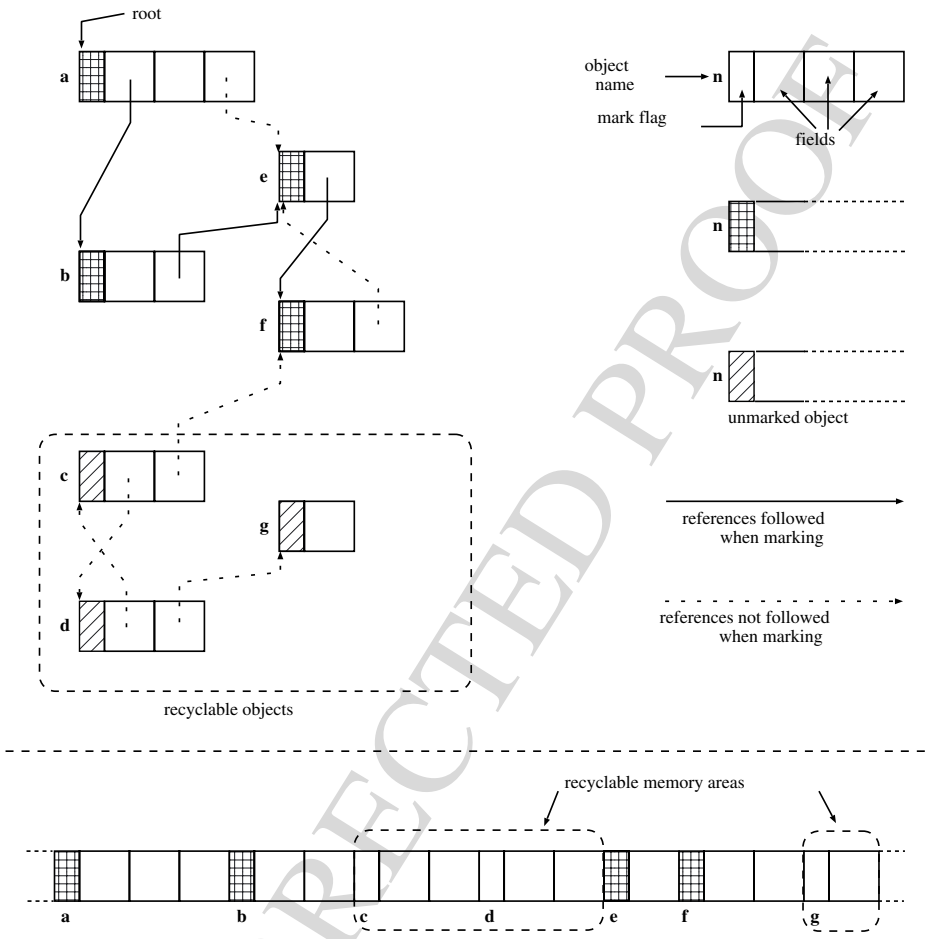

Fig. 1. Mark-and-sweep. Top: Object graph after the mark phase. Bottom: Memory layout corresponding to the top graph.

garbage collection algorithm where the collector execution tends to be more evenly spread and interleaved with the mutator's, except when a large data structure is dropped, in which case additional techniques have to be put into work to spread the pause time.[2]

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 [28], Prolog [1], Lisp [35], C/C++ [10], Eiffel [12], and Java (Sun's KVM [26]).

Its first advantage—although it is not relevant in the context of the ATERM Library—lies in the fact that 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.

[2] For brevity, we do not describe these techniques or even simple reference counting in this paper. The interested reader can refer to the literature (for example [18]) for details.

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 *thrashing*. 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 is often used as an argument to consider the mark-and-sweep algorithm unsuitable for long-running applications, for example in servers. It should be noted nonetheless that in practice non-compacting collectors are often used successfully in very long-running applications. Furthermore, techniques exist to limit fragmentation [24], but fall beyond the scope of the present paper.

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. This is called a misidentification. In this case, an unused object can be marked as live and some unused objects may not be reclaimed.

Furthermore, since this collector is a conservative one and has no compiler support, it may not be a normal, fully copying collector. Indeed, in a normal moving collector, each time an object is moved, all references to this object have to be modified (updated to the new location). But as we mentioned, in a conservative collector, misidentifications may occur. This means that a non-reference word (say, an integer) in the stack may be considered as a reference and be updated to the new location, which causes an incorrect value to be present in the word. 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 that is referenced by the stack or by a register; other objects may be moved safely.

2.4. Possible improvement: a generational algorithm

Although the current memory management system built in the ATERM Library—which we very briefly described in Section 2.3—is quite satisfactory, we think there is still some potential to improve it. Indeed, the ATERM Library garbage collector is simply a classical mark-and-sweep algorithm, applied to 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 detail 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 papers [3,11,13,15,16,25,33,35] 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* [29], which basically says that "most objects die young".

Consequently, it seemed interesting to separately handle these two kinds of data in the garbage collector. One way to do this is to physically segregate these two categories, putting them in two different *generation* sub-spaces. The idea is that this way, 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.

Of course, although using generational collection seems promising, it is no silver bullet [4]: experiments are still needed to know whether a specific type of applications can benefit from it. Furthermore, 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 Fig. 2).

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 Fig. 2).

This cycle consists in tracing, like in the mark-and-sweep algorithm (see Section 2.3), 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 Fig. 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.

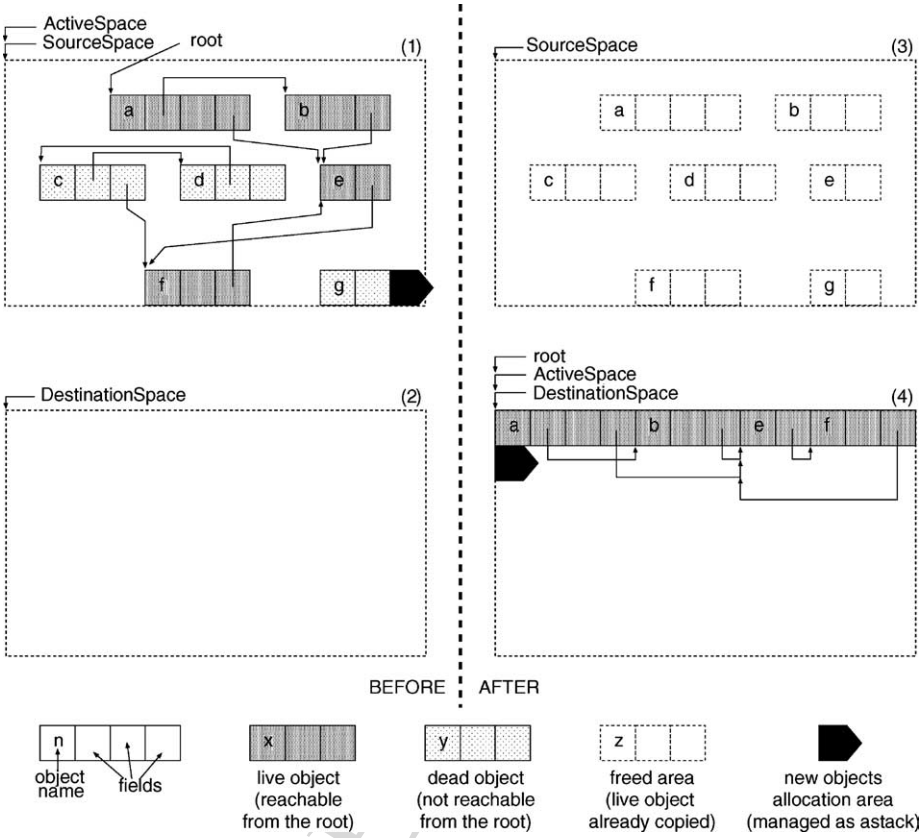

Fig. 2. Copying garbage collection. Left: Memory state just before copy. Right: Memory state just after copy (pointers `SourceSpace` and `DestinationSpace` not updated yet).

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 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 Fig. 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 Fig. 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 no particular interest in this

paper, we shall not present them in depth. Interested readers can find all necessary details in the bibliography, especially [18].

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.

Fig. 3 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 Fig. 2. The left part of Fig. 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

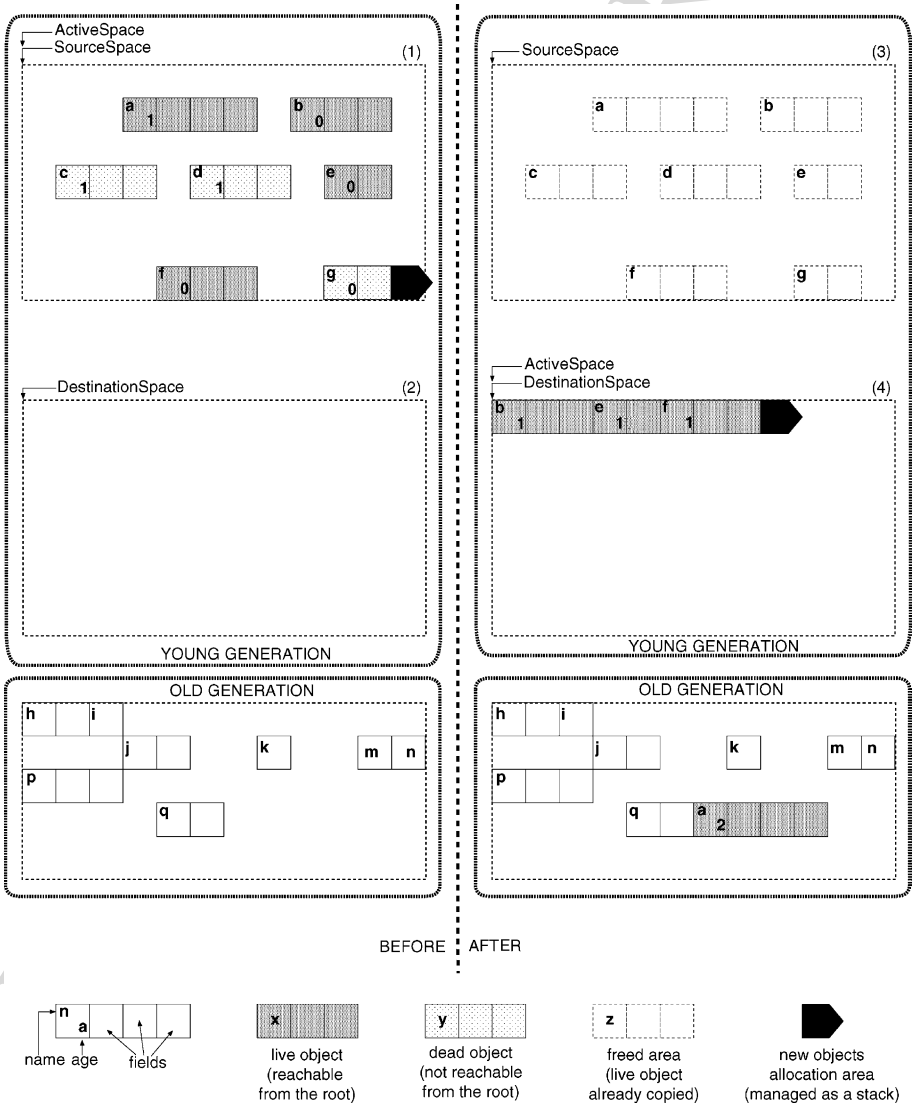

Fig. 3. Generational garbage collection (on the young generation). Left: Memory state just before copy. Right: Memory state just after copy (pointers `SourceSpace` and `DestinationSpace` not updated yet).

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 us 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 (Fig. 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 Fig. 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 increment 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 useful 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 [22], ML [3], Eiffel [17], Java (HotSpot virtual machine from SUN 1.3 JDK) [27], 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 [18]. 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 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 [36], 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 memorywise [35].

The reason for this is a bit unclear to us. We might surmise that the fact that mark-and-sweep matches less closely the generational concepts than copying collectors, as indicated in [14], explains their lower use for generational systems [6,7]. It is also possible that mark-and-sweep algorithms tend to be viewed as somewhat "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 of the generational approach on performance.

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 *generational conservative garbage collector*, which we called GC^2.

As mentioned in Section 3.2, with generational collectors, 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 to be able to manage the various generations in different ways. Inter-generational references 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 the main characteristic of the ATERM Library: the fact that a term is never modified. Therefore, objects are built in a bottom-to-top way and a newly created term only contains references to older terms.

4.1. Overall concepts and data structures

To avoid any misunderstanding, we make a distinction between the notion of *generation* 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 *generations*: this corresponds to an organizational view of the heap. In order to simplify the presentation and the implementation, only two generations of objects are considered in what follows. There is no strong relation between the notion of (object) 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 first generation to the set of young objects, and the second generation to the set of old objects.

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 algorithm we present 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 *first generation* and the *second generation*. 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 first generation is swept.
- Major mark phase: all live objects are marked, be they young or old.
- Major sweep phase: the whole heap (first and second generations) is swept.

According to this general scheme, we have two requirements: first, being able to distinguish young objects from old ones, and second, being able to identify the part of the heap that corresponds to the first generation.

The first requirement can be solved by storing the *age* of each object in its *header* (as already explained in Section 2.1, in the ATERM Library, each object has a header that stores some information, for example the mark flag).

To fulfill the second requirement—identifying the first generation from the second one—a naive solution could consist in maintaining a list of young objects and a list of old objects, 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 first-generation blocks and a list of second-generation blocks. The list of first-generation blocks corresponds to the first generation, and the list of second-generation blocks to the second generation.

This notion of block is also used to improve the 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, called *lists of free cells* in the following. When a new object has to be created, the allocator first checks whether a corresponding free cell is available. If not, some memory has to be allocated or reclaimed. Depending on a heuristics (described in Section 4.4.4), the system decides either to allocate a new block or to start a mark-and-sweep cycle.

When a mark-and-sweep cycle has to be performed, the system also has to choose between a *minor* and a *major* collection. During a major collection, all 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 first generation—the list of first-generation blocks—is swept in order to reclaim unreachable young objects.

4.2. Properties

As mentioned earlier, in this section we describe how the characteristics of the ATERM Library can be exploited to design a new and efficient *generational conservative garbage collector*. The main property we rely on in the ATERM Library is that *a term created by the ATERM Library is never modified*: no update is possible. As a consequence, a term can only contain references to older terms and no old-to-young pointer may occur.

In the following, we will describe the main operations involved in our garbage collector and we will show how they depend on the following invariants and properties and maintain them as well:

(1) There is no old-to-young pointer.
(2) The second generation contains only old objects.
(3) A major collection reclaims all unreachable (old and young) objects, except those referenced by a misidentified pointer.
(4) A minor collection only marks young objects and only sweeps the first generation.
(5) A minor collection does not reclaim any old object.
(6) A minor collection reclaims all unreachable young objects, except those referenced by a misidentified pointer.

4.3. Overview of the algorithm

In this section, we briefly present our GC^2 algorithm. Implementation details and information on how GC^2 is integrated in the ATERM Library follow in Section 4.4.

4.3.1. Memory allocation

The main goal of memory allocation is to provide some free memory when needed. As mentioned above, when a new object has to be created, the allocator first checks whether a corresponding free cell is available. If not, some memory has to be allocated or reclaimed.

By definition, a newly allocated object is a young object and belongs to the first generation. This means that all free cells should belong to first-generation block. This memory allocation policy impacts promotion policy in two ways:

- free memory that belongs to a second-generation block cannot be used, unless the block is de-promoted, or is reclaimed and then re-allocated;
- when a block is promoted, its free memory should be removed from the list of free cells.

4.3.2. Block promotion

This operations selects a first-generation block and moves it to the second generation: the block is removed from the list of first-generation blocks and inserted into the list of second-generation blocks. As mentioned previously, if this block contains free cells, they have to be removed from the corresponding list of free cells (since a block contains fixed size objects, a block is associated to only one list of free cells).

In order to maintain invariant 2 (“the second generation contains only old objects”), a block may be promoted only when it does not contain any young object.

4.3.3. Major collection

During this phase, all live objects are marked, be they young or old, and the whole heap (first and second generations) is swept. By definition, property 3 is maintained: “a major

collection reclaims all unreachable (old and young) objects, except those referenced by a misidentified pointer".

Note that memory recovered in the second generation is not inserted into the free cells lists. The free memory of a second-generation block is returned to the garbage collector only when this block becomes completely empty.

4.3.4. Minor collection

During this phase, all young live objects are marked, and the part of the heap that corresponds to the first generation is swept. By definition, property 4 is maintained: "a minor collection only marks young objects and only sweeps the first generation".

Since a first-generation block may contain old objects (block promotion policy is independent of object age evolution), a reachable old object may belong to a first-generation block. But, since only young objects are marked during a minor collection, it shall not be marked. To ensure the correctness of the algorithm, it suffices to modify the minor sweep phase in such a way that old objects belonging to a first-generation block are never deleted. This modification allows us to safely mix old and young objects in a first-generation block. A potential drawback of this strategy is that some unused memory—the unreachable old objects—is not reclaimed during the minor sweep phase. This is not an issue, because these unreachable old objects will be collected when the following major collection occurs. This modification ensures that property 5, "a minor collection does not reclaim any old object", is maintained by the GC^2 algorithm.

The completeness of the algorithm is ensured by the first invariant. Since "there is no old-to-young pointer", all live young objects are either referenced by a local variable stored in the system stack, or referenced by another young object. As a consequence, all live young objects are marked during a minor-mark phase. Thanks to invariant 2, "the second generation contains only old objects", all unreachable young objects belong to the first generation. Thus, they are reclaimed by the minor sweep phase.

4.4. Integration in the ATERM Library and heuristics

4.4.1. Object representation

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 our current implementation of GC^2, 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 two collections cycles. In our algorithm, we decided to consider 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.

When integrating our algorithm to the ATERM Library, the main impact on object representation was on how object headers were encoded. As presented in Section 2, 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-bit 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 GC^2 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-bit 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 practice, we consider that these limitations are not too restrictive.

4.4.2. Memory allocation

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

The original implementation of the memory allocator, sketched above, is quite simple: when an object has to be allocated and no more memory is available, a block is allocated. This block is prepared according to the size of the object to be allocated, that is divided into elementary cells which are put into the corresponding list of free cells. Then, the first cell is removed from this list and is used to store the object.

This algorithm is interesting for its simplicity but it is not optimal with respect to memory fragmentation and locality. Indeed, given a block which contains some free memory, the corresponding empty cells may not be contiguous in the associated 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 prepared. Furthermore, each time an object is allocated, a cell has to be removed from the list of free cells. The total unitary cost for each cell allocation—excluding object initialization—is 15 assembly instructions.[3] In the remainder of this section, we present our new memory allocation scheme which reduces memory fragmentation and decreases this unitary cost to 9 assembly instructions only in GC^2.

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 prepare 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 recovered it is removed from the ATERM hash-table and the corresponding memory cell is inserted into the associated list of free cells (remember that for each size a free list is defined).

In our new collector, we have a more radical approach: we rebuild entirely all the lists of free cells during the sweep phase. More precisely, at the beginning of the sweep phase, all the lists of free cells (one list per size) are set to empty. Then, when sweeping a block,

[3] When compiled with `gcc 2.95.4` with the `-O` optimization option on a Pentium III machine.

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. As we will show in the next Section 4.4.3, a consequence of this memory allocation approach is to also simplify the promotion of a block.

4.4.3. Sweep phase and block promotion

As mentioned earlier, a first-generation block may contain old objects, in addition to young ones. It could seem interesting to be also able to mix old and young objects in a second-generation 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 a *second-generation block* and x belongs to a *first-generation 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 second generation (remember that second-generation 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 belong to a second-generation block.

To avoid this problem, a solution consists in promoting a block to the second generation only when it exclusively contains old objects. This promotion is performed during the sweep phase, because when sweeping the list of first-generation 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 second 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 first generation.

An intermediate solution seems to be the most interesting in practice. A block can be promoted if it contains only old objects *and* is "reasonably full". The amount of free space tolerated in a block for promotion to the second 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 second generation may *never* be used to allocate a new object. In one sense, this block memory is frozen until the block becomes completely empty.[4] 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 first-generation block). This strengthens the

[4] This may cause memory retention if the block never becomes empty.

fact that a block should be promoted only if it is sufficiently full and the objects within it are not supposed to die soon. It is clear that a block can be promoted after either a minor or a major sweep phase. However, for efficiency reasons, in our implementation, we restricted the promotion to major sweep phases only.

This promotion is itself a very cheap operation, since it consists simply in removing the block from the list of first-generation blocks and putting it in the list of second-generation blocks. After that, it is no longer allowed to allocate new objects in this block. To make the implementation correct, it is necessary to remove the empty cells that belong to the block from the corresponding list of free cells. 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 constant time, by just resetting the list of free cells (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 parameter (i.e. the maximum size of the list of free blocks), the block is either returned to the system or inserted into the list of free blocks.

4.4.4. Generational heuristics

When some memory has to be allocated the system decides either to allocate a new block or to start a mark-and-sweep cycle. This decision process is based on several heuristics, that depend on various parameters of the algorithm. The four most important parameters are:

`good_gc_ratio`: This threshold (reclaimed memory size/heap size) corresponds to the amount of reclaimed memory during the last collection. For example, when more than 75% of the memory is reclaimed in a collection cycle, it is considered as a "good" collection phase.

`max_nb_minor_since_last_major`: This corresponds to the maximal number of minor collections that should be performed between two major collections. By default, this maximal number is 10.

`small_allocation_rate_ratio`: This threshold (size of allocated blocks/heap size) indicates the amount of allocated memory since the last collection. In practice, this ratio is 50%.

`old_increase_rate_ratio`: This threshold (variation of old objects in first-generation blocks between the two last major collections) characterizes the evolution of the old objects that belong to the first generation. In practice, when the size of old objects that belong to the first generation increases by more than 50%, a major collection is performed to allow block promotions.

Relying on these parameters, the decision process can informally be described as follows:

> If the last collection was successful (good reclaiming), go on garbage collecting: do a minor collection, unless it has been too long since the last major one occurred; in the latter case, do a major collection, to allow the second generation to be collected as well from time to time.

If the last collection was not good enough, and we have not allocated much (small increase in memory), then allocate again. But if we have already allocated a lot, we do a collection anyway. It is going to be a minor collection, unless we have too many old objects in the first generation; in the latter case, a major collection is triggered to try and promote objects to the second generation.

This is also expressed, in a more formal and programmatic way, by the (pseudo-) code that manages allocation and collection decisions:

```
if(reclaimed_memory_ratio_during_last_gc > good_gc_ratio) {
    if(nb_minor_since_last_major < max_nb_minor_since_last_major) {
       MINOR_COLLECTION
    } else {
       MAJOR_COLLECTION
    }
} else {
    if(allocation_rate < small_allocation_rate_ratio) {
       ALLOCATE_BLOCK
    } else {
       if(old_increase_rate < old_increase_rate_ratio) {
         MINOR_COLLECTION
       } else {
         MAJOR_COLLECTION
       }
    }
}
```

5. Experimental results

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* [29], we aimed at dramatically reducing the time spent marking and collecting objects. However, as reported in the literature [2,14], 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 to 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 Section 4 has a major 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 are very few reasons to think that the new algorithm can be slower. The only introduced bottlenecks could be related to maintaining ages during marking and 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 in this section, the absence of overhead in practice is confirmed by the experimental results. This can be explained by the simple fact that in the worst case (when the *weak generational hypothesis* [29] is not verified) the generation algorithm has exactly the same behavior as the non-generational one: only major collections occur.

In this section, we thus 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 three main representative examples that correspond to typical uses, from very focused examples to large applications.

As explained in [30], 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 [31], ELAN [8,20], TOM [23] and Stratego [19] 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 three of these systems (ASF+SDF, ELAN and TOM) 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:

5.0.4.1 Fibonacci: a TOM implementation of the Fibonacci function, using Peano integers. This benchmark computes the 32th Fibonacci number.

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

5.0.4.3 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).

For each of these three benchmarks, we provide three graphs to better understand the impact of the GC^2 algorithm: the first one showing execution times, the second detailing memory behavior with maximal sharing and the third one detailing memory behavior without sharing.

Figs. 4, 7 and 10 are the execution times graphs that 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 graph compares our new implementation of the ATERM Library including the GC^2 garbage collector ("new" bar) with the old official implementation (version 1.6.6, "old" bar). For each benchmark program, execution times (in s) 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 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).

Figs. 5, 8 and 11 show the memory behavior of the three benchmarks. The x-axis represents time, in number of allocated cells, and the y-axis is the memory footprint. Each graph contains four lines. The solid line is for objects in the old garbage collector for the ATERM Library, whereas the three other ones pertain to our new GC^2 garbage collector. The bold

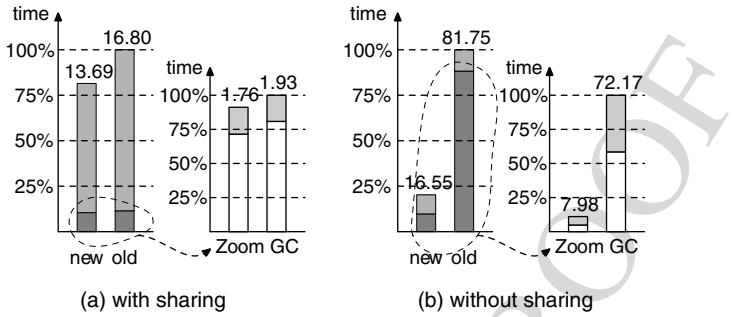

Fig. 4. Execution times for 32th Fibonacci number. (a) With sharing and (b) without sharing.

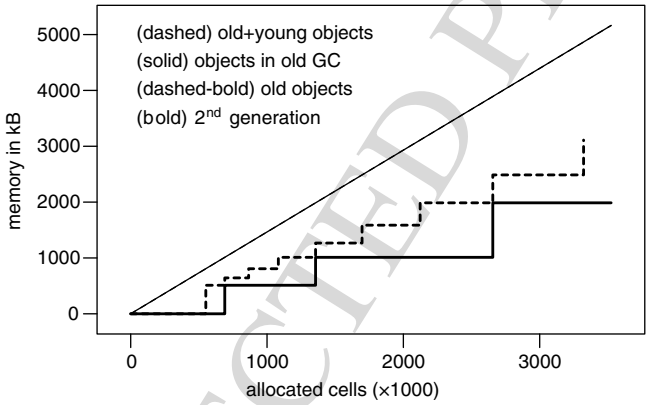

Fig. 5. Memory for 32th Fibonacci number (with sharing).

line shows the evolution of the second generation, that contains only old objects, while the dashed-bold line comprises all old objects (including the ones in the first generation). Finally, the dashed line shows the memory footprint for the whole live set, including young and old objects. Figs. 6, 9 and 12 provide the same information, but for the version without sharing.

As reported in [5], the time to garbage collect is extremely dependent on the total size of the available heap. As illustrated by Figs. 8 and 9, for the Primes benchmark, we made the measurements with a fixed and a specified heap size, in order to more precisely compare the two collectors. However, this fixed heap size has no particular impact on the Fibonacci and the ASF+SDF example since the used memory roughly corresponds to the minimal footprint.

All the figures we present in the following pages were obtained by running the benchmarks on a Pentium III 800 MHz with 512 MB of RAM under FreeBSD 4.7 and with gcc version 2.95.4.

5.1. Fibonacci example

Fig. 4 illustrates the computation of the 32th Fibonacci number, using a Peano representation of integers.

On the maximal sharing version (Fig. 4a), our new algorithm proves efficient, since it decreases overall execution time by 19%—13.69 s instead of 16.80. However, when

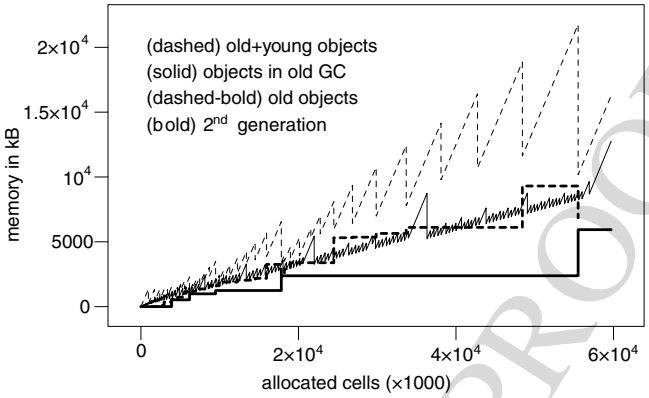

Fig. 6. Memory 32th Fibonacci number (without sharing).

looking at the zoomed graph, the decrease caused by the garbage collector operation itself appears more moderate: 9% (1.76 s versus 1.93). So most of the improvement is indirect, and is found in the mutator, not the collector.

The zoomed graphs also clearly show that marking represents most of the time spent (about 75%) in both old and new garbage collectors. This indicates that, as expected, Fibonacci with Peano integers is a program with a large live set.

These results can be explained by looking at Fig. 5. Note that on this figure, the dashed and the solid lines are overlapped, which indicates that the memory footprints are the same with both collectors. On this figure, the effect of sharing can been seen clearly: overall memory footprint (live set) increases in a very monotonous way, with almost no object ever dying. Old objects appears to account for most of the live set, although the young ones also tend to grow. On this example, the generational approach is convincing because all objects eventually become "old" and never die. Consequently, the second generation keeps growing steadily.

Such a benchmark is thus a challenging one, since collecting never reclaims memory; garbage collection is thus rather useless. Although this is true for the old collector of the ATERM Library, it is not exactly the case for our new generational collector. Indeed, by eventually promoting old objects in the second generation, it can avoid some of the work when doing a minor collection. This explains the 9% advantage it gains when compared with the old collector.

The version of Fibonacci without sharing (Fig. 4b) shows a different picture. In this case, our new generational garbage collector proves very efficient, offering an 80% decrease in overall execution time—16.55 s versus 81.75. Furthermore, this gain comes entirely from the garbage collector itself: collection takes 89% less time with our new collector (7.98 s) than with the old one (72.17 s). This is explained by the fact that Fibonacci without sharing is a very memory intensive program, where the garbage collector takes a large amount of total execution time. Since our collector is better, it represents "only" 48% of total execution time (7.98 s out of 16.55), whereas this ratio raises to 88% with the old collector (72.17 s in 81.75), which doesn't leave much for the mutator execution.

When looking at Fig. 6, these good results for GC^2 are easy to understand. In this version of Fibonacci without sharing, most of the newly allocated objects die early, so the young objects total size decreases a lot after each garbage collection cycle. However, some objects do not die and become old. Once an object is old, the probability of it dying is

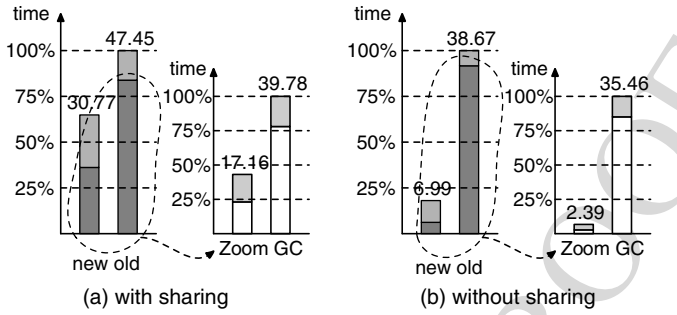

Fig. 7. Execution times for Primes. (a) With sharing and (b) without sharing.

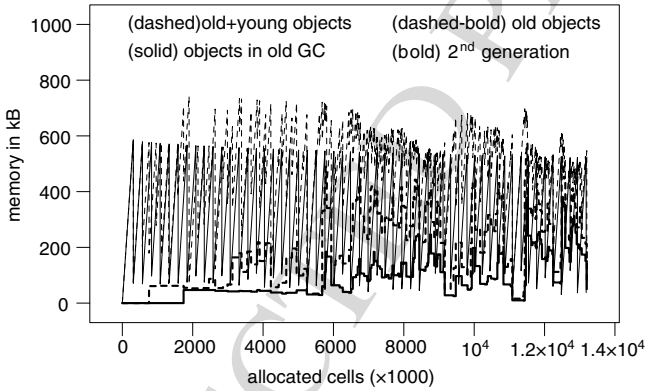

Fig. 8. Memory for Primes (with sharing).

very low, since the old objects lines (dashed-bold) rarely decreases. GC^2 performs many less collections—but much more effective ones—than the old collector, as can be seen by comparing the dashed and the solid lines. This contributes to reducing the total cost of our new collector. Furthermore, most of the collections done in GC^2 are minor ones (whose effect is seen on the dashed line, whereas major collections also impact the bold lines). Since a large part of the heap is made of old objects (dashed-bold line), the marking time in minor collections is strongly decreased.

Thus, the facts that our algorithm does not spend time on the second generation, and spends less time on old objects in first generation, explain the much better performance it offers when compared to the old non-generational garbage collector. In such situations, where a lot of the live objects are long-lived ones, our generational GC^2 algorithm logically performs well, compared to a non-generational one.

5.2. Primes example

The Primes example (Fig. 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 (Fig. 7a), this implies that the list has to be completely rebuilt each time. This explains why the garbage collector represents a very large part of the total execution time: 56% in our new implementation (17.16 s of 30.77) and 83% in the old one (39.78 s of 47.45). This makes this benchmark very different from the previous one, where the garbage collection times rep-

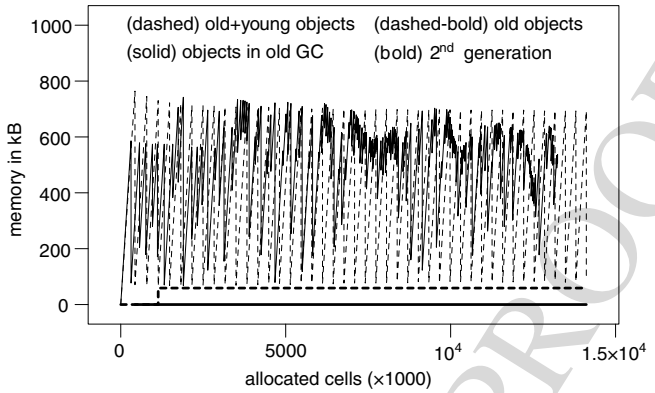

Fig. 9. Memory for Primes (without sharing).

resented on the maximal sharing versions a much more reduced part of the total execution times. This Primes benchmark thus puts heavy pressure on memory, even on the more "reasonable" maximal sharing version. In such cases, our GC^2 algorithm performs quite well. Indeed, whole program execution with our new garbage collector takes 35% less time than with the old ones (30.77 s versus 47.45). The garbage collection only part represents with our algorithm 57% less time (17.16 s instead of 39.78) than with the old collector.

When considering Fig. 8, that represents the memory behavior for Primes with maximal sharing (and bounded memory), it is clear that Primes is very memory intensive. The variations in memory are numerous and quite steep, which indicates that collections are many and quite effective, both with the old garbage collector and our new one. At first sight, both collectors seem to have the same behavior. Indeed, our minor collections (evidenced by the dashed line) impact memory very much like the collections performed by the old collector (solid line): many very successful collections manage to reclaim a lot of objects and keep the memory bounded. However, significant differences do exist, that become more obvious when looking at old objects in GC^2. Indeed, although the second generation (bold line) remains small, the total size of old objects (dashed-bold line) increases and becomes an important part of the heap. This means that minor collections in GC^2 save time, since they do not act on old objects. Since minor collections are numerous, this saving adds up and explains part of the speed advantage of our new collector versus the old one. The bold lines also tell us that when a major collection is performed, it reclaims a number of old objects, though not as many as young objects. This gives precise extra information about the behavior of the Primes example: in addition to the numerous short-lived objects, there are objects that live long enough to be considered old, but die after some time. Primes is thus very different from Fibonacci, where objects were long-lived ones.

The large pressure the Primes benchmark puts on the garbage collector is of course even truer in the non-sharing version, in Fig. 7b. There, garbage collection times represent 34% of the total execution time for our algorithm (2.39 s of 6.99) and 92% for the old one (35.46 s of 38.67). In such a case, it can clearly be said that the efficiency of GC^2 shines through. Indeed, overall program speedup is excellent: Primes with no sharing with our new algorithm takes 82% less time (6.99 s compared to 38.67). Speedup when considering only the garbage collection algorithms is even more obvious: ours takes only 7% of the time of the old one (2.39 s instead of 35.46), which corresponds to an execution speed multiplied by almost 15.

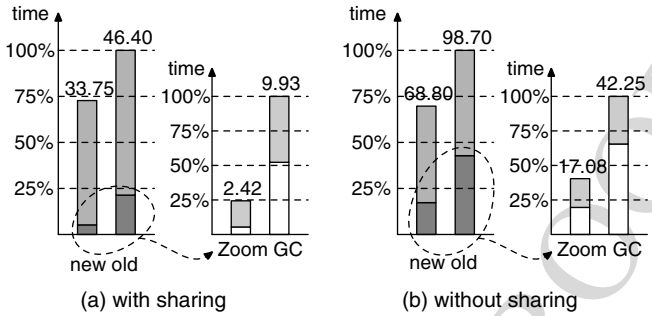

Fig. 10. Execution times for ASF+SDF Compiler. (a) With sharing and (b) without sharing.

Memory behavior for Primes without sharing is shown in Fig. 9. Overall, the picture is very much like that of Primes with maximal sharing: a lot of effective collections for both collectors. Important differences between the maximal sharing and the non-sharing versions of Primes appear nonetheless. First, with GC^2, old objects, as shown by the bold and dashed-bold lines, are very few. In fact, the second generation remains empty, while old objects in the first generation represent a constant amount of memory.[5] This memory represents one tenth of the total footprint, and corresponds to constant terms: the natural integers.

A second important difference with the maximal sharing version of this benchmark can be seen by looking closely at the solid line (old collector). There, a huge number of collections are performed, each thus reclaiming a smaller part of the total heap on average. It can be said that the very high memory pressure of this benchmark "excites" the old collector very much. On the contrary, our new GC^2 algorithm keeps a collection frequency roughly similar to what it had with the maximal sharing version, hence having few collections but very effective ones. This greatly explains the huge speed advantage of our new collector over the old one.

5.3. ASF+SDF example

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: the ASF+SDF Compiler, a large application that does an intensive use of the ATERM Library. This benchmark is indeed in a way much more significant than the previous ones, because it consists of a very large real world application.

Execution time results for the ASF+SDF Compiler benchmark are presented in Fig. 10. In the maximal sharing version—Fig. 10a—our new algorithm represents 7% of the total execution time (2.42 s of 33.75), whereas the old one accounts for 21% (9.93 s of 46.40) of the total time. When considering only garbage collection, our generational algorithm takes 76% less time than the old one, with 2.42 s versus 9.93. On the whole program execution, our generational algorithm also has a significant impact, allowing the ASF+SDF Compiler to take 27% less time than the version with the old garbage collector (33.75 s versus 46.40).

The zoomed part of Fig. 10a allows us to better compare the two algorithms and the relative impact of marking (white) and sweeping (grey). In the old garbage collector, marking represents more than half of the collection time, whereas sweeping accounts for most of the

[5] Except of course at the very beginning of the execution, since a few collections are needed to age the objects.

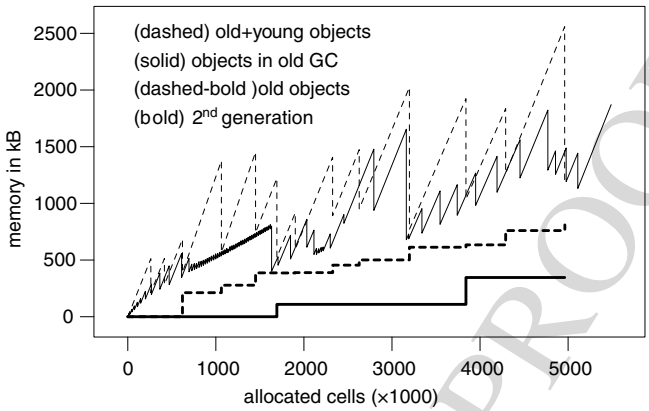

Fig. 11. Memory for ASF+SDF Compiler (with sharing).

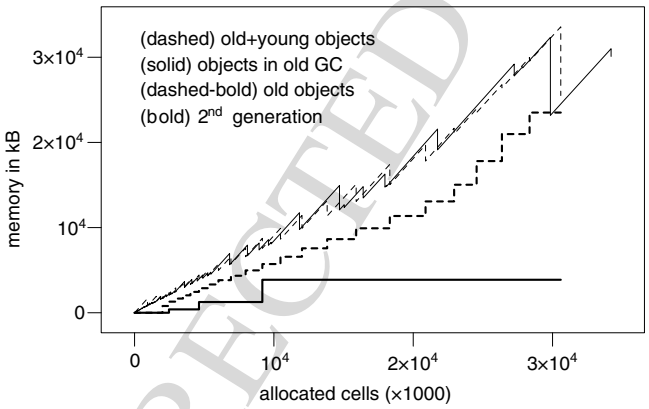

Fig. 12. Memory for ASF+SDF Compiler (without sharing).

time spent in our new GC^2 generational collector (more than three quarters). This seems typical of our generational approach, where old objects act as mark barriers, thus saving time at the mark phase.

This is confirmed by looking at Fig. 11, that shows the memory behavior for the ASF+SDF Compiler with maximal sharing. There, it is clear than the size of the old objects (dashed-bold line) increases steadily and represents a large part of the heap. Indeed, the ASF+SDF Compiler is an application that handles big terms and features a large set of live objects. 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. On the contrary our new algorithm, being a generational one, is at its best in this case because when doing a minor collection it marks only the young objects, and does not mark (or even sweep) most of the old objects, hence a large speed advantage. Fig. 11 indeed shows that GC^2 performs many more minor collections than major ones. Another reason for the speed advantage of our algorithm is that, as can been seen by comparing the dashed and the solid lines, GC^2 triggers less collections that the old garbage collector, especially in phases of high allocation rate (eg. around 1000×10^3 allocated cells). Our heuristics thus appear better than the old collector's.

Fig. 10b shows that overall the no-sharing version of the ASF+SDF Compiler benchmark behaves very much like the maximal sharing version, when considering execution

times. The main difference are total execution times and the part taken by the garbage collector, which increase because of the higher memory pressure. In the non-sharing version, our new algorithm represents 25% of the total execution time (17.08 s of 68.80), whereas the old one accounts for 43% (42.25 s of 98.70) of the total time.

When considering only garbage collection, our generational algorithm decreases collection time by 60%, with 17.08 s versus 42.25. Overall program execution time decreases by 30% with our generational algorithm (68.80 s instead of 98.70), which means the ASF+SDF Compiler has become 1.4 times as fast as the non-generational version. The zoomed graph in Fig. 10b shows us that in both collectors, time due to marking has significantly increased, which is logical considering the much larger live set because of the absence of sharing.

Details of the memory patterns for the non-sharing version of the ASF+SDF Compiler benchmark are provided by Fig. 12. This non-sharing version behaves a bit differently from the maximal sharing version. The main difference is that total memory increases much faster and higher, mostly because of the old objects. Since the footprint of young objects remains relatively small—about a quarter of the total footprint—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 Fig. 10b. Note that the constant, small size of the second generation indicates that our promotion policy can be improved in this benchmark, since more old objects should be in this second generation instead of the first one. The issue is likely to be caused by young objects in blocks containing mostly old objects, thus preventing the blocks from being promoted to the second generation.

5.4. Synthesis

As shown by the previous, varied, benchmarks, our new GC^2 generational algorithm significantly improves the performance of applications realized with the ATERM Library.

Garbage collection times improvements get larger as the memory pressure increases. On the non-sharing benchmarks, collection times decrease by 68–93%, and overall program execution times by 30–82%. On the maximal sharing benchmarks, which are probably more representative of applications realized with the ATERM Library, garbage collection times decrease by 9–76%, and overall program execution times by 19–35%. Memorywise, GC^2 behaves quite well, although it tends to use a little more memory than the old collector (unless it is memory-limited of course).

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 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 on several benchmarks that clearly show the interest of our approach. With memory footprints roughly comparable to those of the old collector, our new GC^2 algorithm offers significant performance gains. Collection times decrease by up to 76% and overall execution times by up to 35%, on a series of benchmarks used with the usual maximal sharing setting of the ATERM Library. Furthermore, the gains our new collector allows when compared to the old one increase with the memory pressure. Consequently, on the less representative non-sharing tests, for example, improvements reach 93% in collection times and 82% in overall execution times. Of course, programs and behaviors are likely to exist where our collector performs worse than the old one; however, we are unaware of such programs.

This very good performance of our new GC^2 algorithm led it to be integrated in the ATERM Library of CWI and become its default collector.[6]

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 "purely 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.

Acknowledgements

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.

Finally, we also want to thanks the anonymous reviewers for their valuable comments and suggestions that led to substantial improvements in the paper.

References

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

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

[3] Andrew W. Appel, Compiling with Continuations, Cambridge University Press, 1992, pp. 205–214 (Chapter 16).

[4] Henry G. Baker, Infant mortality and generational garbage collection, ACM SIGPLAN Notices 28 (4) (1993) 55–57.

[5] Tim Brecht, Eshrat Arjomandi, Chang Li, Hang Pham, Controlling garbage collection and heap growth to reduce the execution time of java applications, in: Proceedings of the OOPSLA'01 Conference on Object Oriented Programming Systems Languages and Applications, ACM Press, 2001, pp. 353–366.

[6] http://www.cwi.nl/projects/MetaEnv/aterm

[6] Joel F. Bartlett, Mostly-copying garbage collection picks up generations and C++, Technical note, DEC Western Research Laboratory, Palo Alto, CA, October 1989. Sources available from <ftp://gate-keeper.dec.com/pub/DEC/CCgc>.
[7] Hans-Juergen Boehm, Alan J. Demers, Scott Shenker, Mostly parallel garbage collection, ACM SIGPLAN Notices 26 (6) (1991) 157–164.
[8] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Marian Vittek, ELAN: a logical framework based on computational systems, in: J. Meseguer (Eds.), Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, WRLA'96, Electronic Notes in Theoretical Computer Science, vol. 4, September 1996.
[9] Hans-Juergen Boehm, Space efficient conservative garbage collection, in: Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation, ACM SIGPLAN Notices, vol. 28(6), ACM Press, Albuquerque, NM, 1993, pp. 197–206.
[10] Hans-Juergen Boehm, Mark Weiser, Garbage collection in an uncooperative environment, Software—Practice and Experience 18 (9) (1988) 807–820.
[11] David A. Barett, Benjamin G. Zorn, Using lifetime predictors to improve memory allocation performance, in: 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93), SIGPLAN Notices, vol. 28, ACM Press, 1993, pp. 187–196.
[12] Dominique Colnet, Philippe Coucaud, Olivier Zendra, Compiler support to customize the mark and sweep algorithm, in: Richard Jones (Ed.), ISMM'98 Proceedings of the First International Symposium on Memory Management, ACM SIGPLAN Notices, vol. 34(3), ACM Press, 1998, pp. 154–165, ISMM is the successor to the IWMM series of workshops.
[13] L. Peter Deutsch, Daniel G. Bobrow, An efficient incremental automatic garbage collector, Communications of the ACM 19 (9) (1976) 522–526.
[14] Alan Demers, Mark Weiser, Barry Hayes, Daniel G. Bobrow, 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, ACM Press, San Francisco, CA, 1990, pp. 261–269.
[15] John K. Foderaro, Richard J. Fateman, Characterization of VAX Macsyma, in: 1981 ACM Symposium on Symbolic and Algebraic Computation, ACM Press, Berkeley, CA, 1981, pp. 14–19.
[16] Barry Hayes, Using key object opportunism to collect old objects, in: OOPSLA'91 ACM Conference on Object-Oriented Systems, Language and Application, in: Andreas Paepcke (Ed.), ACM SIGPLAN Notices, vol. 26(11), ACM Press, Phoenix, Arizona, 1991, pp. 33–46.
[17] Garbage Collection for ISE Eiffel, Technical report, ISE TR-EI-56/GC, version 3.3.9, Interactive Software Engineering, Inc., 1999.
[18] R.E. Jones, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley, 1996, With a chapter on Distributed Garbage Collection by R. Lins.
[19] Patricia Johann, Eelco Visser, Warm fusion in stratego: a case study in generation of program transformation systems, Annals of Mathematics and Artificial Intelligence (2000).
[20] Hélène Kirchner, 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) (2001) 207–251.
[21] John McCarthy, Recursive functions of symbolic expressions and their computation by machine, Communications of the ACM 3 (1960) 184–195.
[22] Marvin L. Minsky, A Lisp garbage collector algorithm using serial secondary storage, Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, December 1963.
[23] Pierre-Etienne Moreau, Christophe Ringeissen, Marian Vittek, A pattern matching compiler for multiple target languages, in: Proceedings of 12th Conference on Compiler Construction, Warsaw (Poland), Lecture Notes in Computer Science, Springer-Verlag, in press.
[24] J.M. Robson, An estimate of the store size necessary for dynamic storage allocation, Journal of the ACM 18 (3) (1971) 416–423.
[25] Patrick M. Sansom, Simon L. Peyton Jones, Generational garbage collection for Haskell, in: R. John M. Hughes (Ed.), Record of the 1993 Conference on Functional Programming and Computer Architecture, University of Glasgow, June 1993, Lecture Notes in Computer Science, vol. 523, Springer-Verlag, 1993 (University of Glasgow).
[26] Inside the K Virtual Machine (KVM), Presentation at Sun's 2000 Worldwide Java Developer Conference (JavaOne), Sun Microsystems, Inc., San Francisco, California, 2000.

[27] The Java HotSpot Performance Engine Architecture, A White Paper About Sun's Second Generation Performance Technology, Sun Microsystems, Inc., 2000.
[28] David A. Turner, Miranda—a non-strict functional language with polymorphic types, in: Jean-Pierre Jouannaud (Ed.), Record of the 1985 Conference on Functional Programming and Computer Architecture, Lecture Notes in Computer Science, vol. 201, Springer-Verlag, Nancy, France, 1985, pp. 1–16.
[29] David M. Ungar, Generation scavenging: a non-disruptive high performance storage reclamation algorithm, ACM SIGPLAN Notices 19 (5) (1984) 157–167, 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, 1984, pp. 157–167.
[30] Mark G.J. van den Brand, Hayco A. de Jong, Paul Klint, Pieter Olivier, Efficient annotated terms, Software—Practice and Experience 30 (2000) 259–291.
[31] Mark G.J. van den Brand, Paul Klint, Pieter Olivier, Compilation and memory management for ASF + SDF, in: Stefan Jähnichen (Ed.), Proceedings of Compiler Construction 8th International Conference, Lecture Notes in Computer Science, vol. 1575, Springer, 1999, pp. 198–213.
[32] Paul R. Wilson, Uniprocessor garbage collection techniques, in: Yves Bekkers, Jacques Cohen (Eds.), Proceedings of International Workshop on Memory Management, University of Texas, 16–18 September 1992, Lecture Notes in Computer Science, vol. 637, Springer-Verlag, USA, 1992.
[33] Paul R. Wilson, Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.
[34] Olivier Zendra, Dominique Colnet, Coping with aliasing in the GNU Eiffel Compiler implementation, Software—Practice and Experience 31 (6) (2001) 601–613.
[35] Benjamin G. Zorn, Comparative performance evaluation of garbage collection algorithms, Ph.D. Thesis, University of California at Berkeley, March 1989, Technical Report UCB/CSD 89/544.
[36] Benjamin Zorn, The effect of garbage collection on cache performance, Technical Report CU-CS-528-91, University of Colorado at Boulder, May 1991.