

Incorporating Precise Garbage Collection in an Ada Compiler

Francisco García-Rodríguez¹, Javier Miranda^{2,*}, and José Fortes Gálvez¹

¹ Departamento de Informática y Sistemas
Universidad de Las Palmas de Gran Canaria
Canary Islands, Spain
`francisco.garcia.100@gmail.com`
`jfortes@dis.ulpgc.es`

² Instituto Universitario de Microelectrónica Aplicada
Universidad de Las Palmas de Gran Canaria
Canary Islands, Spain
`jmiranda@iuma.ulpgc.es`

Abstract. In recent years an increasing effort to develop garbage collectors for real-time applications has been undertaken by the Java community. Currently it seems appropriate to evaluate the effort required to integrate such a facility into Ada.

This paper presents an ongoing project to accomplish this goal by modifying the GNAT compiler to incorporate support for precise garbage collection. The approach taken can be immediately applied to current Ada 95 code, and allows coexistence of explicit and implicit deallocation. The text describes the extra code generated by a modified version of the front end and the corresponding run-time support for a mark-and-sweep collector.

1 Introduction

Garbage collectors have proved to be valuable for the construction of safer programs. Complex designs with intensive memory usage can benefit from implicit memory deallocation [13]. In the context of real-time applications, garbage collectors have been traditionally avoided because of efficiency and schedulability considerations. However, during the past decade the industry has invested substantial resources to develop garbage collectors appropriate for real-time systems, e.g., Real-Time Java implementations [10,16], embedded systems [2] and hardware assisted methods [15]. While Ada implementations have traditionally provided memory management facilities beyond unchecked deallocation and controlled types, support for garbage collection has been ignored by most.

In this paper we summarize the current state of an ongoing academic project consisting in the modification of the GNU Ada compiler (GNAT) to incorporate

* This work was done during a six-month visit to the NYU Courant Institute funded by the Spanish Minister of Education and Science under project PR2006-0356.

precise recovery of implicitly deallocated data. The approach taken relies on a new implementation-defined pragma used to mark data types for implicit deallocation. Legacy sources using explicit deallocation can be easily adapted for garbage collection, or for simultaneous use of both deallocation paradigms, thus allowing to evaluate the impact of the garbage collector on existing applications.

The rest of this paper is structured as follows. Section 2 briefly introduces the garbage collector scenery. Section 3 presents the current design of the integrated support for our garbage collector. Section 4 describes the overall workings by means of a short example. Section 5 presents the results of some performance tests. We close with some conclusions and the bibliography.

2 Garbage Collectors

Garbage collectors have evolved and become widely used since the early 1960s. Initially, they were confined to specialized areas and programming languages, specially symbolic languages as Lisp, for which one of the first garbage collectors was built by McCarthy [14].

Limited resources in earlier computers gained garbage collection a reputation for slowing down program execution to unacceptable levels. Recently, the use of garbage collection in the Java Virtual Machine has shown that the reliability of implicit deallocation of dynamic variables and the automatic recovery of its memory space can be combined with sufficient efficiency on current machines [20].

Despite having matured, adapted to new requirements, and taken advantage of new computing features, most garbage collectors continue to work around the same original algorithms. These fall under two basic categories, namely *tracing* and *reference counting* [19,13].

Algorithms. Tracing collectors locate and free those heap-allocated data blocks that are currently unreachable, directly and through any path, from any pointer in the rest of storage areas (typically stack and static areas). Knowledge of such pointers is essential for the proper functioning of precise garbage collectors and are commonly referred as *roots*, hence their collective name of *root set*. The tracing approach has been used to develop most garbage collector variations, including support for cache optimization, concurrency, data compaction, generational sets, etc. [19,13]

In reference counting [9], a counter is associated with each individually heap-allocated data block to count the number of references to it. Accordingly, counters must be updated at every pointer assignment. When a counter reaches zero, its corresponding data block can be deallocated. Unreachable cyclic data structures prevent their node counters from reaching zero, and thus, in the basic algorithm, they cannot be freed when they become garbage. However, this can be solved with modern techniques [3].

Finally, some recent research focus on alternative, region-based methods [18].

Integration. Garbage collection has been included in some programming languages since their inception, e.g., Lisp and Java. In these cases, semantic restrictions are generally present in the language specification to facilitate its implementation, e.g., restricting or avoiding pointer arithmetic. This allows compiler writers to build a proper framework for the garbage collector and let it work with an exact knowledge of the memory layout, eventually resulting in detailed and complete recovery of unreachable data, what is known as *precise* garbage collection.

The situation is quite different when trying to introduce a garbage collector after the language has been defined and implemented. Most problems relate to memory identification, and heuristics must be devised to identify pointers. False references might lead to unreachable data blocks not being collected. However, Zorn has argued that they will not lead to significant memory leaks [20].

A widely used solution to this problem is the fully *conservative* collection [7,6], particularly the family of collectors implemented by Boehm et al. [5], which are used in C and C++ implementations among others. Zorn [20] has suggested that this solution can be as time-effective as explicit memory deallocation, at the cost of considerable memory usage overhead. Some collectors mix both conservative and precise styles, as in the compacting garbage collector by Bartlett [4].

Finally, integration of garbage collection with real-time restrictions poses particular problems. For instance, Real-Time Java [8] offers a mixture of programmer-controlled and implicit deallocations, which some consider inappropriate and for which they propose less intrusive and more homogeneous approaches [16,2].

3 Adding Garbage Collection to GNAT

Ellis and Detlefs [11] analyze the steps and conditions to incorporate a garbage collector into a compiler for a language with no specific provisions for it. Inspired by their work, we established the following guidelines to design a widely acceptable implementation through evolutionary improvements.

- *Precise memory identification.* We consider that the more predictable behavior and lower execution support overhead of precise collectors are valuable, specially when extending our project for concurrent and real-time programming. Moreover, in these cases, the programmer may possibly want to identify some data structures for explicit deallocation.
- *Minimize changes to the compiler.* Collector support must be as unintrusive as possible, and must avoid slowing down code sections which do not use implicit deallocation. Following the GNAT architecture, we restrict our changes to the front end and run-time support.
- *Minimize the impact on the programmer.* It is important to facilitate the use of the garbage collector with legacy applications and libraries that use explicit deallocation.
- *Flexibility.* It is desirable to allow the programmer to choose the collector. The implementation should be as independent of the collector algorithm as possible.

Garbage Collection Algorithm. According to these requirements, we have chosen a basic *mark-and-sweep* [14] algorithm for our initially nonconcurrent implementation of a garbage collector. We found it to be the least intrusive in the compiler and in its generated code. It only requires the ability to locate the root pointers needed to start the mark phase, and to register during allocation calls the data block addresses required for the sweep phase.

A reference count algorithm, although not requiring the building and maintenance of a root set, requires the insertion of code at pointer assignments. In addition, single deallocations become recursive.

Nonetheless, most of the infrastructure has been designed to provide support for future implementation of other garbage collection algorithms.

Pragma *Garbage_Collected*. In order to use implicit deallocation, the programmer simply has to apply a new implementation-defined pragma *Garbage_Collected* on those data types whose dynamic variables will be implicitly deallocated by the garbage collector. These types will be called *collected* types. At compile time the pragma leads to extra semantic checks and code expansion to provide the support discussed in the following sections.

The approach we have taken has almost immediate application, or easy adaptation, to previous sources that use explicit deallocation. Moreover, both explicit and implicit deallocation paradigms can be simultaneously used in the same program, thus allowing the gradual incorporation of implicit deallocation in legacy software. Nevertheless, we do not exclude the possibility of providing eventually, as an option, general garbage collection compatible with real-time.

Currently, the pragma can only be applied to tagged types. If a collected type is derived, the pragma must be applied at least to its ultimate ancestor. The compiler propagates the collected-type property to all its derivations. However, for documentation purposes, if the pragma is not applied to a type derived from a collected type, the front end generates a warning message suggesting to the programmer the addition of the pragma to such sources.

The mentioned restrictions result from the following considerations.

- *Object type identification.* We rely on the ability of some run-time support to identify the type of a given object. In Ada this support is only available by default for tagged types through the tag of the object (see the standard package `Ada.Tags` [12, Section 3.9]). In future versions we will investigate how to remove this restriction.
- *Ultimate ancestor and all its derivations.* Allowing the use of the pragma on a type with a noncollected ancestor leads to complex schemes that need further analysis. For example, if the programmer applies the pragma to a type `T2` derived from tagged type `T1`, which is defined in some legacy library (and is therefore noncollected), the garbage collector may erroneously deallocate objects of type `T2` that may remain referenced by `access-to-T1'Class` objects in the legacy library.

A direct consequence of the latter restriction is that controlled types cannot be implicitly deallocated, because their ultimate ancestor is defined in a standard Ada package [12, Section 7.6]. The main benefits of this corollary restriction is that it helps us to keep our conceptual model simple. We plan to investigate the combination of controlled and collected types in future versions.

Finally, pragma `Controlled` is used in Ada to prevent any automatic reclamation of storage (garbage collection) for the objects created by allocators of a given access type [12, Section 13.11.3]. Therefore, it is an error to apply this pragma to a collected type.

Data Structures. Our prototype implementation has the following three main data structures. The following sections describe how they are updated and used by the run-time system.

- *Descriptor Table*, containing one descriptor for each collected type. Each descriptor entry is generated by the compiler and used to know at run-time the type size and alignment, and the address of some functions generated by the frontend to know the object layout and deallocate objects of such type. The tag of the collected type is used as the hash key to register the descriptor in the run-time system during program initialization.
- *Collectable-Block Table*, containing an entry for each allocated block of each collected type. Each entry contains the block’s address and meta-data (as needed by the garbage collection algorithm, e.g., mark bit). This physical separation from the respective block’s data avoids intrusive components that could lead to problems with legacy applications containing Ada representation clauses [12, Section 13.5.1]. The table is implemented as a hash table, and addresses are used as hash keys.¹ The entries are inserted at each allocation, and are removed when their corresponding blocks are deallocated.
- *Root-Set Stack*, containing the address of each access-to-collected-type object in the program stack or static area.

3.1 Root-Set Maintenance

Siebert [17] reviews the best known strategies for real-time root scanning. Just like collectors, they can be precise, conservative or mix both styles. As mentioned before, we prefer the well defined behavior of precise algorithms.

Our implementation keeps the root-set stack continuously updated. Whenever the declaration of a variable holding one or more pointers to collected types is placed in the stack, the address of each such pointer is saved in the root-set stack. When the execution of the program leaves the scope containing such variable declarations, all these addresses are removed from the root-set stack.

The following example shows the definition of a collected type *Cell*. Part of the code that is automatically generated by our modified front end is shown as comments.

¹ This approach has the added benefit of allowing us to check whether an address points to a collectable block.

```

type Cell;
pragma Garbage_Collected ( Cell );

type Cell_Ptr is access all Cell;
— for Cell_Ptr 'Storage_Pool use GC.Pool;

type Cell is tagged record
  Prev : Cell_Ptr;
  Key  : Positive;
  Next : Cell_Ptr;
end record;

```

The *collected pool* (`GC.Pool`) is used to allocate objects created by allocators of collectable objects. The programmer is free to use the standard storage pool, as well as other pools, to manage explicitly deallocated data.

For each declaration of an access type to collectable type (like `Cell_Ptr` in the example), extra code is generated in place by the front end to associate such an access type with the collected pool.

Given these declarations, the following example presents as comments the actions performed by the run-time system to manage the root-set stack.

```

procedure Foo is
  — GC.MarkTop;
  P : Cell_Ptr; — GC.Push (P' Address);
  N : Cell;     — GC.Push (N.Prev' Address,
                — N.Next' Address);

begin
  ...
  — GC.PopUntilMark;
end Foo;

```

The top of the root-set stack is saved when the procedure is entered, and is restored before the procedure returns. During the elaboration of each declaration of a pointer (like *P*) or composite object containing pointers (as the *Prev* and *Next* components of *N*) to collected type, the addresses of such pointers are pushed onto the root-set stack.

3.2 Layout Functions

For each collected type, our modified front end generates a layout function. These functions are necessary for the tracing process to follow the pointers stored in collectable blocks. They return an array of offsets that is used by the garbage collector to locate such pointers. In order to support both fixed and variant records, the code generated by the front end makes use of the standard Ada attribute `Position` [12, Section 13.5.2]. Here is the function generated for type `Cell` of our example.

```

function Cell_Offsets
  (Obj_Addr : Address) return GC.Offset_Array
is
  type Cell_Acc is access Cell;
  function To_Cell_Acc is
    new Unchecked_Conversion (Address, Cell_Acc);

  Offsets : Offset_Array (1 .. 2) :=
    (1 => To_Cell_Acc (Obj_Addr).Prev' Position ,
     2 => To_Cell_Acc (Obj_Addr).Next' Position );
begin
  return Offsets ;
end Cell_Offsets ;

```

3.3 Coexistence with Noncollected Types

References between both kind of types, collected and noncollected, present some difficulty. Figure 1 shows the storage areas considered by our conceptual model and the pointers allowed by our implementation. Pointers like *a* and *b* have a known lifespan and are managed as already explained. Pointers like *c* can be overlooked because handling them is programmer's duty. Pointers from noncollected pools are obviously not managed by the collector: the collector cannot know how long they live, nor any data blocks that they may point into the collected heap. Accordingly, their presence would imply that the automatic deallocation of any data block in the collected heap is no longer safe.

Our compiler rejects pointers from noncollected types to collected ones. A more complex solution could accept these references, at the cost of some execution time overhead on noncollected types.

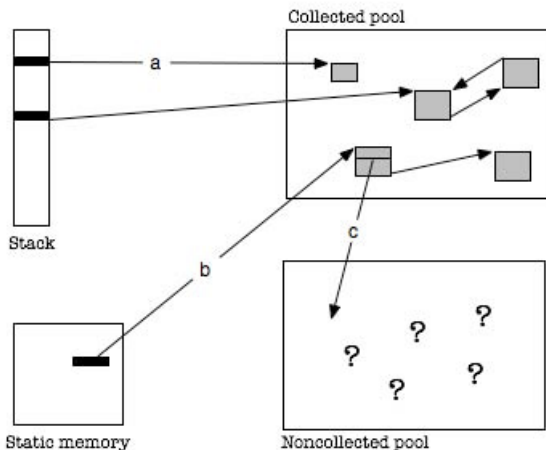


Fig. 1. Storage areas and allowed pointers

4 The Collector at Work

Let us complete our previous example with an overview of the steps taken during the execution of code with automatic deallocation generated by our compiler.

The following example code first builds two double linked nodes, with values 8 and 13, and then creates and links a new node with value 31 replacing node with value 13, which becomes garbage. See Fig. 2.

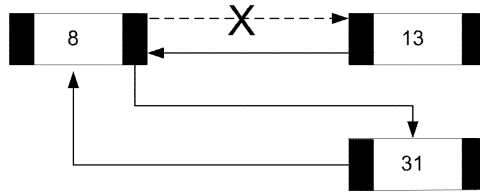


Fig. 2. Double-linked list example

Similarly to our previous example, here we have the code of this program and the extra actions it performs when executed.

```

procedure Example is
  — GC.MarkTop;
  P1 : Cell_Ptr;   — GC.Push (P' Address);
  P2 : Cell;       — GC.Push (P2.Prev' Address,
                  — P2.Next' Address);
begin
  P1 := new Cell;

  — Element A
  P1.all := Cell'(null, null, 8);

  — Element B
  P1.Next := new Cell;
  P1.Next.all := Cell'(P1, null, 13);

  — Element C makes B garbage!
  P1.Next := new Cell;
  P1.Next.all := Cell'(P1, null, 31);

  P2.Next := new Cell;   — GC triggered here!
  — GC.PopUntilMark;
end Example;

```

When execution of procedure *Example* begins, the current top of the root-set stack is saved by *GC.MarkTop*. Then, the address of *P1*, as well as addresses of *P2.Last* and *P2.Next*, are saved in the stack. Next, the successive allocations of *Cell* objects are captured by the collected pool *GC.Pool*, and each newly allocated data block's address, along with its mark-bit set to unmarked, is stored in the

table of collectable blocks. So far, only normal program code along with the extra support code has been executed.

Now let us suppose that the final allocation for *P2* triggers a mark-and-sweep garbage collection. The execution of the mutator program is then temporarily paused and the tracing process starts from each address stored in the root-set stack. For instance, the address in *P1* is examined in the collectable-block table to check whether it points to some block in `GC.Pool1`. If it does, its mark bit is set to *marked* and, by reading the block's tag, its related descriptor is read from the descriptor table. From *Cell*'s descriptor, its associated layout function is invoked to obtain the offsets of pointer components to collected-type objects. This information is used by the algorithm to trace collectable blocks and thus to recursively mark all the accessible ones.

Next the sweep phase proceeds. By exploring every allocated address contained in the table of collectable blocks, each one is visited, and its mark bit set to unmarked for the next mark phase of algorithm. In our example, only *Cell* block with value 13 has its mark-bit initially unmarked when visited, since it could not be reached during the mark phase. Therefore, the *Cell*-specific unchecked deallocator is called for this block after finding its type descriptor from the block's tag.

5 Testing

We have applied some performance tests to have a preliminary evaluation of the overhead of our current implementation. All these tests use linked lists of heap-allocated nodes and perform the following actions: build the list, remove half of the list elements at different positions while preserving the list integrity, and finally manually launch the garbage collector. The Ada compiler is our modified version of the GAP 2005 distribution [1]. We used a 2.4 GHz Pentium 4 PC with 512 MB RAM under the Ubuntu 5.10 Linux distribution to run the tests.

In each test we evaluate the cost of our implementation support (including the additional code expansion and the run-time cost of the garbage collector) versus the cost of the same algorithm using explicit deallocation only. The following table summarizes our test results.

Test variant	Expanded code length	Execution time dep. on list length		
		5,000	10,000	20,000
Without any support	663	261.4ms	1.670s	11.59s
With support code plus collector	953	267.8ms	1.684s	11.74s
No. of push instructions		49,986	99,986	199,986

The column titled *Expanded code length* has the number of lines of the expanded code. This information was obtained using a GNAT switch that produces a listing of the expanded code in Ada source form.

The source test program is 63-lines long. According to the table results, after expansion, $953 - 663 = 290$ lines correspond to the additional code generated

by our modified frontend to maintain the root-set stack and the collectable-block table, as well as the collected-type related functions for table management, descriptor construction, layout, and deallocation.

The next column presents the execution time of the program using a list with 5000, 10000, and 20000 nodes. The execution times shown in the table have been averaged over 8 runs of the test program (these values were obtained from the *user* time information provided by the Unix `time` command). The execution time growth rate is justified by the quadratic complexity of the list algorithm used in the test program.

These figures must be regarded with caution, as they represent only preliminary estimations that will be reduced as the implementation improves, in particular with the integration of more efficient collection algorithms.

6 Conclusions

Garbage collector capabilities are being added to GNAT in an ongoing academic project. The design is focused on changes to the GNAT front end to ease its portability. Precise garbage collection allowing the coexistence of explicitly and implicitly deallocated types is provided. Our current implementation has several limitations that we plan to relax in future enhancements, particularly the collection of untagged and controlled types, and precise collection in a multitasking environment. The latest version of this work is available at <http://www.iuma.ulpgc.es/~jmiranda/gc>.

Acknowledgments

We would like to thank Ed Schonberg and Ben Brosgol for their insightful comments on drafts of this paper.

References

1. AdaCore. GNAT Academic Program (2007), <http://www.adacore.com/home/academia>
2. Bacon, D.F., Cheng, P., Rajan, V.T.: The Metronome: A simpler approach to garbage collection in real-time systems. In: Meersman, R., Tari, Z. (eds.) On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops. LNCS, vol. 2889, pp. 466–478. Springer, Heidelberg (2003)
3. Bacon, D.F., Rajan, V.T.: Concurrent cycle collection in reference counted systems. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 207–235. Springer, Heidelberg (2001)
4. Bartlett, J.F.: Compacting garbage collection with ambiguous roots. Research Report 88/2, Western Research Laboratory, Digital Equipment Corporation (1988)
5. Boehm, H.-J., Demers, A.J., Shenker, S.: Mostly parallel garbage collection. In: PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, pp. 157–164. ACM, New York (1991)

6. Boehm, H.-J.: Space efficient conservative garbage collection (with retrospective). In: McKinley, K.S. (ed.) Best of PLDI, pp. 490–501. ACM, New York (1993)
7. Boehm, H.-J., Weiser, M.: Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18(9), 807–820 (1988)
8. Bollella, G., Brosgol, B., Furr, S., Hardin, D., Dibble, P., Gosling, J., Turnbull, M.: *The Real-Time Specification for Java*. Addison-Wesley, London (2000)
9. Collins, G.E.: A method for overlapping and erasure of lists. *Communications of the ACM* 3(12), 655–657 (1960)
10. TimeSys Corp. RTSJ reference implementation. <http://www.timesys.com/java/>
11. Ellis, J.R., Detlefs, D.: Safe, efficient garbage collection for C++. In: John, R. (ed.) Proceedings of the 1994 USENIX C++ Conference, Cambridge, Massachusetts, pp. 143–178 (1994)
12. Intermetrics Inc. and the MITRE Corporation. Annotated Ada Reference Manual with Technical Corrigendum 1. Language Standard and Libraries. ISO/IEC 8652:1995(E). (2000) <http://www.adaic.org/standards/95aarm/AARM.PDF>
13. Jones, R., Lins, R.: *Garbage Collection: Algorithms for Automated Dynamic Memory Management*. Wiley and Sons, New York (1996)
14. McCarthy, J.L.: Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM* 3(4), 184–195 (1960)
15. Schmidt, W.J., Nilsen, K.D.: Performance of a hardware-assisted real-time garbage collector. In: William, J. (ed.) Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM SIGPLAN Notices, vol. 29, pp. 76–85 (1994)
16. Siebert, F.: Hard real-time garbage-collection in the Jamaica virtual machine. In: Proceedings of the 6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA'99), pp. 96–102. IEEE Computer Society, Los Alamitos (1999)
17. Siebert, F.: Constant-time root scanning for deterministic garbage collection. In: Wilhelm, R. (ed.) *Compiler Construction, CC 2001*, as part of ETAPS 2001. LNCS, vol. 2027, pp. 304–318. Springer, Heidelberg (2001)
18. Tofte, M., Talpin, J.-P.: Region-based memory management. *Information and Computation* 132(2), 109–176 (1997)
19. Wilson, P.R.: Uniprocessor garbage collection techniques. In: Bekkers, Y., Cohen, J. (eds.) *Memory Management, International Workshop (IWMM 92)*. LNCS, vol. 637, pp. 1–42. Springer, Heidelberg (1992)
20. Zorn, B.G.: The measured cost of conservative garbage collection. *Software: Practice and Experience* 23(7), 733–756 (1993)