

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].