The Implementation of Retention in a Coroutine Environment*

John P. Kearns and Mary Lou Sofia
Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA

Summary. An efficient storage management scheme for languages supporting both coroutines and procedures is discussed. This scheme requires identifying those program components whose run time instances might need retention. These components could include both procedures and coroutines. A compile-time marking technique for determining these components in a block structured coroutine environment is presented. The impact that variations in the form of the coroutine control structures have on the technique is also addressed. Using the compile time marking as a guide, the storage management scheme can contract gracefully into a stack when programs use only a procedure call/return control discipline.

1. Introduction

Mixed Control Forms

In many modern high level languages (HLL), several different forms of inter-component control are supported. For example, Ada [1], Concurrent Pascal [11], and Mesa [18] combine concurrent processes (tasks) with recursive procedures; SIMULA-67 [8] and SL5 [12] mix coroutines with recursive procedures. Lindstrom [16] has demonstrated that the combination of backtracking [5, 9, 10] and coroutines in an extension of Pascal [13] may be used for the natural expression of algorithms for complex search problems.

The purpose of this work is not to explore the semantic ramifications of mixed control forms, but rather to consider certain problems presented to the

* This work supported in part by NSF grants MCS-7906102 and MCS-8119341
implementor of a HLL which supports multiple control forms. Specifically, we address the storage management problems which arise when procedures are mixed with “higher” control forms (e.g. coroutines, backtracking, and concurrency).

The LIFO nature of control flow imposed by the semantics of procedure/function call and return makes the stack a natural and efficient storage structure for the activation records in an HLL supporting control forms no more general than recursive procedures. When one moves into the area of retentive control, however, the stack exhibits severe performance problems, as storage deallocation is not tied to the locus of control exiting an instance of a program component. Bobrow and Wegbreit [6] advanced the notion of a “generalized stack” which allows for the coexistence of stackable activation records (e.g., for procedures) with retained activation records (e.g., for coroutines) in the same storage structure. In the generalized stack there is a need to copy an activation record whenever it requires space for temporary storage and finds itself blocked from expansion by another activation record. A recent study [15] has shown (empirically) that for a language supporting coroutines and recursive procedures, the cost of copying is prohibitive. Prellocation of storage for temporaries within activation records was shown to lead to substantial savings both in execution time and memory requirements.

If an implementation of the above HLL preallocates in each activation record sufficient extra storage to preclude the possibility of copying, then the storage structure is effectively a simple heap. In a context in which the use of coroutines constitutes the dominant form of control flow, the heap was shown to be more efficient than any generalized stack variant. If, however, procedure calls dominate, the heap imposes a substantial performance penalty upon an implementation [7].

A general purpose high level language system which provides programmers with both procedures and coroutines would ideally manage storage so that users do not pay for unused generality. Conversely, the implementation is obligated to support all control forms in a reasonably efficient manner. In the storage management component of an implementation of a HLL it appears that the storage management techniques most appropriate for certain types of activation records are in direct conflict with each other. In particular, some activation records are best stored in a stack-like storage structure while others require a heap-like structure.

This realization is the rationale behind the development of the stack-heap [15, 14], a composite storage structure intended for a general purpose HLL supporting both coroutines and procedures. The stack-heap consists of a stack and a heap, each managed independently. Activation records stored in the stack contain no preallocation of storage for temporaries; the heaped ones do. The heap is managed on a simple first-fit, partial compaction, basis. A reference counting technique defines the deletion strategy for the heap. Links (e.g., access and dynamic) will, in general, interconnect activation records stored in the stack and those stored in the heap.