# Memory Representation Issues for Prolog Implementation

W F Clocksin

Computer Laboratory
University of Cambridge
Corn Exchange Street
Cambridge CB2 3QG

## Introduction

The stack-based Prolog implementation of Warren (1977) has been effective and influential. However, when considering future implementations of Prolog-like languages, the problem of incremental garbage collection must be addressed. Program sizes are increasing, and there is more use of sophisticated programming techniques (Warren, 1982; Shapiro, 1983) which prevent the use of the conventional expedient of explicitly backtracking to reclaim data space. Moreover, a more comprehensive GC method than Warren's is required. Such sizes and techniques, together with expectations of system performance, conspire to restrict the effectiveness of conventional GC methods such as mark-and-sweep and stop-and-copy. In this paper we discuss the design issues implied by considering the use of a real-time incremental GC method (Baker, 1978; Leibermann and Hewitt, 1983).

## The Heap

Real-time incremental GC methods have been designed for use with the heap-based memory used by LISP systems. It is instructive to consider the impact of heap-based memory for Prolog. Heap-based memory for Prolog is not new (see MicroProlog (McCabe) and POPLOG (Mellish)), but conventional mark-and-sweep techniques are used. Problems do not arise on a system with limited address space or for small applications, but realistically large applications are more demanding, and seem to require a more sophisticated treatment of heap GC[1].

We shall consider a conventional heap memory, the contents of which are a finite number of *cells*. Each cell consists of N *components* (N > 0), and is referred to by a unique integer called a *pointer*. A component of a cell is at

---

[1]. The problem is not resolved by using a separate GC processor: access to the shared memory is a bottleneck. If the GC processor operates concurrently with the Prolog processor, then severe memory contention results. If the GC processor does not operate concurrently with the Prolog processor, then GC time is limited by memory cycle time, and little is gained by using a separate processor.

least large enough to contain one pointer. We identify three rules to observe when transacting with a heap-based memory system:

- A legal pointer to a cell points only to a conventional (usually the first) address of the cell. This restricts the possible implementation of Prolog variables: variables are often implemented as a pointer to the variable component within a cell. POPLOG gets around this problem by creating a separate variable cell to which variables refer. Although GC methods have been devised to support within-cell references (Wegbreit, 1974), we restrict our discussion to conventional heaps.
- It is not possible to compare addresses to determine lifetimes. In a stack-based system, cell addresses can be compared to determine, for example, whether or not the current activation is determinate. Comparisons are also used to support other optimisations based on determinacy. If the local stack (used to hold activation records in a variation of the Italian manner (Bobrow and Wegbreit, 1973)) is not placed in the heap, then most of the relevant comparisons can be performed (this is discussed further below).
- All components of a cell must be valid during the time GC is permitted. This rule prevents certain optimisations performed by Prolog-X (Clocksin, 1982) for example, in which certain component updates are deferred until the are absolutely necessary, in the hope that they will not become necessary due to subsequent discovery of determinism.

## Feasibility

Using an incremental GC causes a bounded amount of garbage collection to occur for each construction of a new heap cell. This affects the performance profile of a Prolog system. It is becoming popular to measure performance in units of LIPS (logical inferences per second), although this unit has not yet been well defined, where LIPS units are derived from measuring the time taken to run a small program such as naive reverse (Wilk, 1983). However, a LIPS rating conferred on a Prolog system in this way merely indicates that the system is a "sprinter"[2]. A Prolog system using an incremental GC will therefore have a lower LIPS rating than if the same system had no GC. However, a system with incremental GC is likely to be more desirable to use, and it is possible to design an incremental GC which can be deactivated for short periods when "sprints" are necessary.

During the execution of a Prolog program, two patterns of memory use can be identified: long-term storage of clauses and on the global stack. Short-term storage has a high rate of turnover and more dense interconnection of

---

[2]. Pursuing this analogy to excess, equipping such a system with a conventional (not incremental) GC will cause the sprinter to take lengthy "rest stops" when running a "long dis-

references. Clearly, to achieve optimum performance, an incremental GC method must treat the two patterns of memory usage appropriately. The similar case which arises in Lisp has only recently received serious attention. While Baker's method treats the heap uniformly (turning over all cells indiscriminantly), the intent of Leiberman and Hewitt's method is to prevent unnecessary turnover of long-term storage. Results of the Leibermann/Hewitt method are inconclusive, and require further design and simulation.

For the purpose of gaining more performance, further special cases can be accommodated. For example, it is not necessary to place the local stack in the heap. Garbage is not created in the local stack. If the local stack is implementaed as a stack, then address comparisons for detecting determinism can be performed, and faster access is possible. However, the stack must be explicitly considered as a source of roots for the GC.

## Benefits

Providing a uniformly addressable heap with an incremental GC confers a few beneficial side-effects. First, the system is easier to interface to heap-based languages such as Lisp and Smalltalk. Second, there is safe and timely recovery of the space used by retracted clauses. This is a problem in non-heap-based Prolog systems (O'Keefe, 1983), for example, space is unexpectedly occasionally not recovered in DECsystem-10 Prolog (because recovery is related to backtracking), and it is possible to generate dangling pointers in certain rare cases in Prolog-X. C-Prolog requires a large (> 10%) overhead in managing the clause database, even when no retracts are used. This is related to the overhead required to maintain a reference-counting GC method specifically for the clause database.

---

tance".