A new data structure for implementing extensions to Prolog

Serge Le Huitouze

IRISA
Campus de Beaulieu, 35042 Rennes Cedex
FRANCE
serge@irisa.fr

Abstract

We propose a new data structure for implementing delayed computations in Prolog as efficiently as commonly employed solutions, together with the optimal memory management needed for this data structure. It appears that this data structure (and the associated memory management) is also useful for implementing other extensions where there is a need for reversibly modifying terms other than in the leaves, e.g. functional extensions.

We first recall that optimal memory management in Prolog is possible only when considering the binding states in a precise manner.

Then we describe the classical implementation of delayed computations, along with the new memory management problems involved. The conclusion is that a correct implementation, w.r.t memory management, is only possible when introducing a new data structure which is known by the garbage collector.

A memory management machine, called MALI, has been realized in our group following the principles described, and offering such a data structure, called the attributed variable. On top of this machine, we have implemented a number of extensions to Prolog.

In the course of our experiments, we have verified that the initial choice in the design of MALI, i.e. to make it independent from the procedural semantics of Prolog, was a good choice. Indeed, we have been able to implement various logic programming systems extending the so-called “pure-Prolog”, ranging from PrologII to $\lambda$-Prolog and fair proof strategies.

1 Introduction

The implementation of Prolog has been studied intensively, resulting in the definition of the WAM [Warren 83], which is the basis of almost all current Prolog compilers. This definition rests upon the procedural semantics of Prolog, transforming the solving of a goal into a procedure call. This compilation scheme results in very high speed in execution.

Comparatively, very little work has been done in the area of memory management for Prolog. For example, in the WAM, the division of the dynamic memory into two
stacks and a heap makes tail recursion optimization and environment trimming possible. However, the heap is recoverable only on backtracking and a garbage collector for the heap is necessary.

Our research group has studied the basic principles of logic programming memory management, and has proposed a solution. This solution is described in the first part of the paper. We implemented this solution, which we wished to be as independent from Prolog as possible. This implementation is the memory MALI, presented in the second part, together with the way of using it to implement a Prolog system.

Finally, in the last two parts, we present extensions which we implemented with MALI. In all these extensions, a new object provided by MALI, the attributed variable, made possible an efficient implementation, through the suppression of associative searches and/or multiple copies.

The generality of MALI and its independence from the procedural semantics of Prolog were of great help in those implementations which, moreover, inherit automatically from a complete garbage collection.

2 Memory management for Prolog

2.1 Resolution state

The search strategy employed in Prolog (depth-first strategy), makes it possible to represent Prolog’s resolution state by a stack of goal statements (abbreviated as GS), the top being the current GS to be erased, the rest being the waiting GS to be considered in a LIFO manner.

This state changes on two occasions.

. When there is more than one clause to be tried in a resolution step, the top GS is duplicated in the stack.

. During a resolution step, the first goal in the top GS is replaced with the body of the chosen clause, and a substitution is applied to the new GS thus obtained.

2.2 Representation of the resolution state: OR-sharing

Various Prolog implementations differ mainly in the way they represent GS and terms. However, they share some characteristics:

. a stack for the traversal of the search tree is always present

. different GSs share their representation when possible

Let’s examine the second point in greater detail. It is tempting to replace the duplication of the top GS by the duplication of a reference to it. This doesn’t work so well, because the top (the current) GS will be modified by substitutions and thus will alter the other waiting GSs. Noticing that the only changes occur via variable substitutions, the remedy is to record these changes in a new data structure, called the trail, which is used when popping a waiting GS from the stack, to update some bound variables. That’s what we call **OR-sharing**. The consequence of OR-sharing is that the pushed GSs are altered by the current state of execution. They will recover their exact value when popped from