An Automaton-Driven Frame Disposal Algorithm and its Proof of Correctness

M Satpathy, A Sanyal and G Venkatesh
e-mail: {mrs,as,gv}@cse.iitb.ernet.in

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay-400076, India

Abstract. Activation records or frames of function calls, in a functional programming implementation, are either maintained in a stack or in heap. A frame is usually treated alive till the function returns, though long before that its requirement may have been over.

In this paper, we define the concept of disposing a frame at earliest point in time and do dispose a frame as soon as we are sure that it will no longer be required. To do this we first construct a finite automaton from the program text and use this automaton to guide the frame disposal. We also prove that the disposal strategy is correct.

The advantages are many. It reduces the size of root-set from which the garbage collector starts pointer-chasing to scan the live data. It also delays the occurrences of garbage collection and in the process may improve upon the number of such occurrences in a program execution.

Keywords: Functional Programming, Garbage Collection and Static Analysis.

1 Introduction

In an implementation of a functional language, the activation record or the frame of a function could either be kept in stack or in heap. The default strategy, in both the cases, is that a frame is considered alive till the execution of the associated function is complete. But in many cases, somewhere during the evaluation of its body, the function might cease to refer to its frame for the evaluation of its remaining part. In our discussion, the implementation strategy is call-by-need or lazy. So the natural choice is to keep frames in heap though it is possible to maintain many of them in stack [4]. If a frame is known to be obsolete, it can immediately be returned to the memory-allocator, or, the later could be informed that the frame is no more live. Disposing a frame as early as possible, and not waiting till the function returns, has the following advantages.

- Minimizing space leakage: Garbage collectors collect the whole of live data by traversing pointers from a root-set which comprises of the machine registers and the runtime stack. But there can be cases when a function frame is reachable from the root-set, but it is obsolete in the sense of our earlier description. Traversing from such obsolete frames to collect live data will in
turn declare some garbage as genuine data. This problem is otherwise known as *space leakage* [8]. Providing the garbage collector with the information that some frames are obsolete, helps in reducing the amount of space leakage.

- Reducing burden on the garbage collector: If frames are in heap, they are to be collected by a garbage collector. Disposing them as early as possible may prevent the heap area from getting filled up too often, thereby avoiding garbage collection in some cases. Moreover, the frame space that gets collected do not have to undergo the expensive operations usually used to detect garbage objects at run time.

The organization of the paper is as follows. Section 2 defines the concept of frame disposal at near-earliest point in time and describes how frames could be disposed under this strategy with information from the runtime call-tree. Section 3 describes the algorithm to construct an automaton from a given program and how it can be used for disposal. Section 4 describes how to specialize the automaton for efficiency. Section 5 contains the correctness proof. Section 6 extends the near-earliest strategy to the earliest strategy. The final section concludes the paper.

## 2 Disposal of frames in near-earliest time

When a function body gets executed according to some evaluation order, at some point in between, all information, necessary for the computation of function result, may have already been available. Then at this point in evaluation, the frame can be disposed of without violating the safety criteria. This could better be seen from some examples.

**Example 1:** Let \( F \ x \ y = 1 + (x + y) \) in \( F \ 1 \ 2 \).

We assume right-to-left order of evaluation of the arguments to a strict primitive. Evaluation starts by a call to \( F \), whose frame contains the bindings of \( x \) and \( y \) to 1 and 2 respectively. \( y \) will be evaluated first, followed by the evaluation of \( x \). As soon as the evaluation of \( x \) is complete, observe that the frame is no longer required and hence could be disposed of.

**Example 2:** Let \( F \ x \ y = 1 + (x + y) \); \( G \ z = 2 * z \) in \( F \ (G \ 2) \ 3 \).

Evaluation starts with a call to \( F \), with \( x \) bound to \( (G \ 2) \) and \( y \) bound to 3. Figure 1(a) shows how the evaluation proceeds. \( y \) is evaluated first. Then for the evaluation of \( x \), a call to \( G \) is made with its only argument \( z \) bound to 2. Now we are in the body of \( G \) and for the evaluation of \( z \) we have just to get its value from \( G \)'s frame. After getting this value observe that it is not just that the frame of \( G \) is no longer required but so is the frame of \( F \). So at this point in evaluation, both the frames of \( G \) and \( F \) could be disposed of.

**Example 3:** Let \( F \ x = H \ 1 + (2 * G \ x) \)

\[
G \ z = z + 1; \ H \ p = p - 1
\]

in \( F \ 2 \).