Liveness-Based Garbage Collection

K. Prasanna Kumar[1], Amey Karkare[2], and Amitabha Sanyal[1]

[1] IIT Bombay, Mumbai 400076, India
{prasanna,as}@cse.iitb.ac.in,
[2] IIT Kanpur, Kanpur 208016, India
karkare@cse.iitk.ac.in

Abstract. The memory requirement of a lazy functional programming language can be decreased in two ways. The first is to use the result of a *liveness analysis* during garbage collection to preserve only *live* data—a subset of *reachable data*. This results in collection of greater amount of garbage and consequently fewer garbage collections. Apart from evaluated data this will also results in early release of memory hogging closures. The second way is to eagerly evaluate closures that are guaranteed to be evaluated in the future. The information that is required for this eager evaluation comes from *strictness analysis*.
For a first-order lazy functional language, we present an uniform framework for liveness and strictness ana lysis. We start by for a first-order functional for language we formulate a context-sensitive liveness analysis for structured data and prove it correct. We then use a 0-CFA-like conservative approximation to annotate each allocation and function-call program point with a finite-state automaton—which the garbage-collector inspects to curtail reachability during marking. As a result, fewer objects are marked (albeit with a more expensive marker) and then preserved (e.g. by a copy phase).
Experiments confirm the expected performance benefits—increase in garbage reclaimed and a consequent decrease in the number of collections, a decrease in the memory size required to run programs, and reduced overall garbage collection time for a majority of programs.

1 Introduction

Most modern programming languages support dynamic allocation of heap data. Static analysis of heap data is much harder than analysis of static and stack data. Garbage collectors, for example, conservatively approximate the liveness of heap objects by their reachability from a set of memory locations called the *root set*. Consequently, many objects that are reachable but not live remain uncollected, causing a larger-than-necessary memory demand. This is confirmed by empirical studies on Haskell [**?**], Scheme [**?**] and Java [**?**] programs.

Here we consider a first-order pure functional language and propose a liveness analysis which annotates various program points with a description of variables and fields whose object references may be dereferenced in the future. The garbage collector then only marks objects pointed by live references and leaves other, merely reachable, objects to be reclaimed. (Although not strictly necessary, a collector would normally *nullify* dead variables and fields rather than leaving dangling references.) Since there are fewer live objects than reachable objects, more memory is reclaimed. Additionally, since the collector traverses a smaller portion of the heap, the time spent for each collection is also smaller. The work is presented in the context of a stop-the-world non-incremental garbage collector (mark-and-sweep, compacting or copying) for which we also show a monotonicity result: that our technique can never cause more garbage collections to occur in spite of changing the rather unpredictable execution points at which collections occur. We anticipate that our technique is applicable to more modern collectors (generational, concurrent, parallel), but leave such extensions to future work.

We first define a *fully context-sensitive* (in the sense that its results are unaffected by function inlining) liveness analysis and prove it correct. However, fully context-sensitive methods often do not scale, and this analysis would also require us to determine, at run-time, the internal liveness of a function body at each call. Hence, similarly to the 0-CFA approach, we determine a context-independent *summary* of liveness for each function which safely approximates the context-dependence of all possible calls [**?**,**?**,**?**]. (Note that an intraprocedural context-insensitive method which assumes no information about function callers would be too imprecise for our needs.) In essence our approach sets up interprocedural data-flow equations for the liveness summaries of functions and shows how these can be solved symbolically as context-free grammars (CFGs). We can then determine a CFG for each program point; these are then safely approximated with finite-state automata which are encoded as tables for each program point. For garbage collection purposes only *GC points* (program points representing calls, to **cons** or to a user-function) need to be stored.

We previously proposed an intraprocedural method for heap liveness analysis for a Java-like language [**?**] which statically inserted statements nullifying dead references to improve garbage collection; by contrast nullification here occurs dynamically (which can work better with aliasing) when the garbage collector acts on liveness annotations to avoid traversing dead references. A workshop paper [**?**] outlined the basic 0-CFA-style-summary interprocedural approach to functional-program liveness analysis. The current paper adds the context-sensitive analysis and better formalization along with experimental results.

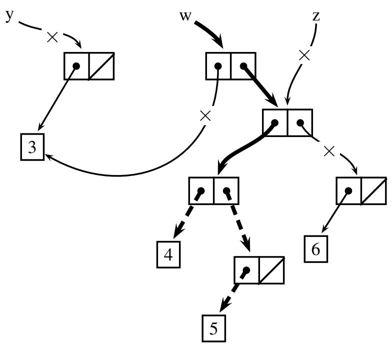
Motivating Example Figure 1(a) shows an example program. The label π of an expression e denotes a program point. During execution of the program, it represents the instant of time just before the evaluation of e. We view the heap as a graph. Nodes in the heap, also called (**cons**) *cells* contain **car** and **cdr** *fields* containing values. Edges in the graph are *references* and emanate from *variables* or fields. Variable and field values may also be atomic values (**nil**, integers etc.) While it is convenient to box these in diagrams, our presented analysis treats them as non-heap values.

Figure 1(b) shows the heap at π. The edges shown by thick arrows are those which are made live by the program. In addition, assuming that the value of any reachable part of the program result may be explored or printed, the edges marked by thick dashed arrows are also live. A cell is marked and preserved during garbage collection, only if it is reachable from the root set through a path of live edges. All other cells can be reclaimed. Thus if a garbage collection takes at π with the heap shown in Figure 1(b), only the cells `w` and (**cdr** `w`), along with (**car** (**cdr** `w`)) and all cells reachable from it, will be marked and preserved.

```
(define (append l1 l2)
  (if (null? l1) l2
      (cons (car l1)
            (append (cdr l1) l2))))

(let z ←(cons (cons 4 (cons 5 nil))
              (cons 6 nil)) in
  (let y ← (cons 3 nil) in
      (let w ← (append y z) in
         π:(car (cdr w)))))
```

(a) Example program.



(b) Memory graph at π. Thick edges denote live links. Traversal stops at edges marked × during garbage collection.

Fig. 1. Example Program and its Memory Graph.

Organization of the paper Section **??** gives the syntax and semantics of the language used to illustrate our analysis along with basic concepts and notations. Liveness analysis is described in Section **??** followed by a sketch of a correctness proof relative to a non-standard semantics. Section **??** shows how to encode liveness as finite-state automata. Section **??** reports experimental results and Section **??** proves that a liveness based collector can never do more garbage collections than a reachability based collector.