Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

Markus Mohnen

Lehrstuhl für Informatik II, Aachen University of Technology
Ahornstraße 55, 52056 Aachen, Germany
email: markusm@zeus.informatik.rwth-aachen.de

Abstract. This paper describes a compile-time garbage collection (ctgc) method in the setting of a first-order functional language with data structures. The aim is to obtain information on positions in a program where certain heap cells will become obsolete during execution. Therefore we develop an abstract interpretation for the detection of inheritance information which allows us to detect whether the heap cells of an argument will be propagated to the result of a function call. The abstract interpretation itself is independent of the evaluation strategy of the underlying language. However, since the actual deallocations take place after termination of functions, the information obtained by the abstract interpretation can only be applied in an eager context, which may be detected using strictness analysis in a lazy language. In order to increase efficiency we show how the number of recomputations can be decreased by using only parts of the abstract domains. The worst case time complexity is essentially quadratic in the size of the program. We illustrate the method developed in this paper with several examples and we demonstrate how to use the results in an eager implementation. Correctness of the analysis is considered, using a modified denotational semantics as reference point. Experimental results demonstrate the practicability of our method. A main goal of our work is to keep both the run-time and the compile-time overhead as small as possible.

1 Introduction

One of the advantages of modern functional languages is the ability to work with dynamic data structures without the necessity to control memory allocations explicitly. On the other hand, this prevents a programmer from expressing the reusability of heap cells of intermediate data structures. However, since the available memory is restricted in concrete computers, there must be some means of deallocating obsolete memory cells. But well-known garbage collection mechanisms make no (or only little) use of the special structure of the underlying functional programs.

We present an abstract interpretation which exploits this special structure. The underlying observation is that data cells which were created specially for a certain function call become obsolete, if they are not inherited to the function result, i.e. if they are not reachable from the top cell of the result. In contrast to other ctgc approaches (like for instance [JM89]) we do not try to reuse obsolete
cells as soon as possible, i.e. during function execution, because this would cause severe changes in the abstract machine and some runtime overhead. Therefore, application of our technique is only possible in 'eager' situations, i.e. either in an eager language or at positions in lazy programs which were detected as eager by a strictness analysis.

Furthermore, we show that our method can be implemented very efficiently by using test arguments instead of the full domains. The worst case complexity for general programs is essentially $O(n^2)$ where $n$ is the number of function definitions of the program. Especially for realistic programs we will show that the computation of the fixpoint can be done almost in linear time.

We will present experimental results which prove that the memory consumption is dramatically improved. Moreover, in combination with traditional garbage collection (gc), even the run-time is improved, which is due to less frequent gc cycles.

We finish this introduction by an outline of the contents of the paper. The next section gives a more detailed intuitive description of our method. Section 3 formalises this intuition and emphasises the problem of arbitrary data structures and their finite representation. After that, we give a brief discussion of the correctness of our method and explain how to use the results of the abstract interpretation in an eager implementation. The next section contains a discussion of efficiency issues. We show how to decrease the time spent for computing the fixpoint and give experimental results. The paper concludes with a comparison with related work and some prospects for further research.

2 Intuitive Description

We want to address the problem in the setting of a first-order functional language with constructors. For the moment, we will assume eager evaluation, although the abstract interpretation itself is also applicable to programs of a lazy language. A constructor on the right hand side on a function definition causes the allocation of a corresponding heap cell, if it is encountered during the execution. We assume boxed representation of basic types, i.e. the heap cell representing a constructor contains just pointers to other heap cells, which may represent constructors or basic values. The loosening of some of these restrictions will be discussed in Sec. 8.

The general idea of the optimisation can be summarised in the following way: if there is a subterm $f(g(t))$ on the right hand side of a function definition and we know that $f$ does not inherit parts of the result of the evaluation of $g(t)$, then deallocate all cells belonging to these parts which were allocated during the execution of $g(t)$. The deallocation takes place when the call of $f$ terminates. In an eager implementation this can easily be done, since the termination point of a function is fixed.

In order to illustrate which kind of information we want to derive, we consider the following example:

```plaintext
datatype List0fInt ::= Nil | Cons(int, List0fInt)
append : List0fInt x List0fInt -> List0fInt
append(Nil, L) := L
```