Science of Computer Programming 111 (2015) 426–457

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# Space consumption analysis by abstract interpretation: Inference of recursive functions [☆]

Manuel Montenegro [*], Ricardo Peña, Clara Segura

*Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain*

H I G H L I G H T S

- First space analysis for a functional language with regions.
- We use abstract interpretation on the infinite domain of multivariate monotonic functions.
- Our bounds go beyond multivariate polynomials.
- We have formally proved the correctness of all the results and implemented all the algorithms.

A R T I C L E   I N F O

*Article history:*
Received 6 August 2012
Received in revised form 14 February 2014
Accepted 11 April 2014
Available online 28 April 2014

*Keywords:*
Resource analysis
Abstract interpretation
Functional languages
Regions

A B S T R A C T

We present an abstract interpretation-based static analysis for inferring heap and stack memory consumption in a functional language. The language, called *Safe*, is eager and first-order, and its memory management system is based on heap regions instead of the more conventional approach of having a garbage collector. This paper begins by presenting *Safe* features by means of intuitive examples, and then defines its formal semantics, including the memory consumption of particular program executions. It continues by giving the abstract interpretation rules for non-recursive function definitions, and then how the memory consumption of recursive ones is approximated.

An interesting property of our analysis is that, under certain reasonable conditions, the inferred bounds are *reductive*, which means that by iterating the analysis using as input the prior inferred bound, we can get tighter and tighter bounds, all of them correct. In some cases, even the exact bound is obtained. However, and due to lack of space, reductivity is not presented in this paper. The complete development can however be found in a technical report available at the authors' site.

The paper includes a related work discussion, and small examples. Bigger case studies are presented in the fore-mentioned technical report.

## 1. Introduction

Among the set of desirable properties of a program, the most decisive ones are those related with its correctness, which ensures that a program does what the programmer expects it to do. This class of properties is commonly known as *functional properties*. Besides these, there are some other desirable properties that are relevant to the safety of software systems. These

[☆] Work supported by the projects TIN2008-06622-C03-01/TIN (STAMP), TIN2009-14599-C03-01 (DESAFIOS10), S2009/TIC-1465 (PROMETIDOS) and the MEC FPU grant AP2006-02154.

[*] Corresponding author.
*E-mail addresses:* montenegro@fdi.ucm.es (M. Montenegro), ricardo@sip.ucm.es (R. Peña), csegura@sip.ucm.es (C. Segura).

http://dx.doi.org/10.1016/j.scico.2014.04.006
0167-6423/© 2014 Elsevier B.V. All rights reserved.

are called *non-functional properties*. An example is the fact that a program performs its task in a given amount of time, or that its memory need not exceed a given limit. These two examples become part of a broader research field, whose name is *resource analysis*. In this framework, a program is conceived as a resource consumer (resource may be understood as time, memory, energy, etc.) and the aim is to compute an upper bound to the resources being consumed by every possible execution of the program.

In this work, we are particularly interested in the analysis of *memory bounds*. Memory consumption is specially relevant to several scenarios: for instance, when programming embedded devices, it is necessary to make sure that the programs running in these devices do not stop working because they try to use more memory than is available. It is also useful to know in advance how much memory will be needed by the program in order to reduce hardware and energy costs. Although relatively new, the field of resource analysis has gained considerable attention in the last years, mainly due to the application of mathematical techniques (such as linear programming, or recurrence solving) to programming languages. In particular, the inference of memory bounds is a very complex task that involves several auxiliary analyses, each one a challenge by itself.

The first results on memory consumption analysis were targeted towards the functional programming paradigm. The developed techniques were subsequently adapted to mainstream languages, such as Java or C++.

Hughes and Pareto introduce in [24] a first-order functional language with a type and effect system guaranteeing termination and execution in bounded space. This system is a combination of Tofte and Talpin's approach to regions and of sized types [25,37].

The first fully automatic way to infer closed-form memory bounds is due to Hofmann and Jost [21]. Their analysis, based on a type system with resource annotations, can infer linear heap memory bounds on first-order eager functional programs with explicit deallocation. These techniques have been applied to subsets of imperative languages, such as Java [22] and C [18]. The annotated type system also serves as a basis for a stack consumption analysis due to Campbell [8,9]. Hofmann and Jost's approach is extended in [27,26] to higher-order programs. The latter work provides a general framework that can accommodate different notions of cost. More recently, Hoffmann and Hofmann have extended [21] to polynomial memory bounds [20,19], and Simões et al. [43] have extended it in a different direction: they provide a system computing the memory cost of functional programs with lazy evaluation.

The classical approach to resource analysis, due to Wegbreit [47], involves the generation of a recurrence relation from the program being analysed, and, in a second phase, the computation of a closed-form expression (without recursion) equivalent to that recurrence relation. Vasconcelos and Hammond pursue this approach in [46], which is fully automatic in the generation of recurrence equations, but requires the use of an external solver for obtaining a closed form. The COSTA system [3] follows a similar approach, but it provides its own recurrence relation solver, PUBS [2], which can handle multivariate, non-deterministic recurrence relations. COSTA is an abstract interpretation-based analyser which works at the level of Java bytecode, and supports several notions of cost, such as the number of executed bytecode instructions, heap consumption, and number of calls to a particular method. Since memory management in Java is based on garbage collection, their approach to memory consumption is parametric on the behaviour of the garbage collector [5]. The bounds computed by this system go beyond linear expressions; it can compute polynomial, logarithmic, and exponential bounds.

This paper describes the automatic analysis of memory bounds for a first-order functional language called *Safe*. This language has been developed in the last few years as a research platform for analysing and formally certifying properties of programs, with regard to memory usage. It was introduced for investigating the suitability of functional languages for programming small devices and embedded systems with strict memory requirements.

The absence of a notion of state makes reasoning about the functional properties of a program easier. Due to this lack of state, functional languages are in general better suited to several static analyses. However, the inference of memory bounds requires special attention. In most functional languages memory management is delegated to the runtime system, which allocates memory as it is needed by the program, provided there is enough space available. A *garbage collector* is in charge of determining, at runtime, which parts of the memory are no longer needed, and can be safely disposed of. The main advantage of this approach is that programmers do not bother about low-level details on memory management, but there are also some drawbacks. On the one hand, the time delay introduced by garbage collection may prevent a program from providing an answer in a required reaction time, which may be unacceptable in the context of real-time systems. There has been some successful work on real-time garbage collectors. For instance, in [42] the author guarantees a worst-case execution time by calling the memory recovery operations within the critical threads, but the price to be paid appears to be having a rather complex system. On the other hand, garbage collection makes it difficult to predict at compile time the lifetimes of data structures, specially in those cases where the runtime system does not specify under which conditions a garbage collection takes place.

In order to compute memory bounds for *Safe*, we have decided to dispense with the garbage collector and to have a heap structured as a region stack in which regions are allocated and deallocated in constant time. Given this memory model, we use abstract interpretation-based techniques [14] for inferring non-linear, monotonic, closed-form expressions bounding the heap and stack memory costs of a program.

Since the memory needs of a program usually depend on its input, the bounds we obtain in our analysis are multivariate functions on the sizes of the inputs. The bounds given by this memory consumption analysis are considered correct if they are equal to or greater than the actual worst-case runtime consumptions of the program being analysed.