Science of Computer Programming 111 (2015) 458–482

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# Space consumption analysis by abstract interpretation: Reductivity properties ☆

Manuel Montenegro *, Ricardo Peña, Clara Segura

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

## HIGHLIGHTS

- First space analysis for a functional language with regions.
- Our bounds go beyond multivariate polynomials.
- Our bounds have the property of reductivity, i.e. they improve by iteration.
- We have formally proved the correctness of all the results and implemented all the algorithms.

## ARTICLE INFO

*Article history:*
Received 6 August 2012
Received in revised form 24 April 2014
Accepted 29 April 2014
Available online 22 May 2014

*Keywords:*
Resource analysis
Abstract interpretation
Functional languages
Regions

## ABSTRACT

In a previous paper we presented 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.

In this paper we concentrate on an important property of our analysis, namely that the inferred bounds are *reductive* under certain reasonable conditions. This 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.

The paper includes several examples and case studies illustrating in detail the reductivity property of the inferred bounds.



## 1. Introduction

In a previous paper [13], we presented an abstract interpretation-based static analysis for inferring heap and stack memory consumption in a functional language. It describes the automatic analysis of memory bounds for a first-order functional language called *Safe*, which 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. Memory consumption is especially relevant for instance when programming embedded devices. It is necessary to ensure that programs will not stop due to lack of memory. It is also useful to know in advance how much memory will be needed in order to reduce hardware and energy costs.

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 [8] a first-order functional language with a type and effect system guaranteeing termination and execution in bounded

☆ 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.2014.04.014
0167-6423/

space. This system is a combination of Tofte and Talpin's approach to regions and of sized types [9,19]. The first fully automatic way to infer closed-form memory bounds is due to Hofmann and Jost [7]. Their analysis, based on a type system with resource annotations, can infer linear heap memory bounds on first-order functional programs with explicit deallocation. This approach is extended in [11,10] to higher-order programs. More recently, Hoffmann and Hofmann have extended their initial work [7] to polynomial memory bounds [6,5], and Simões et al. [20] have further extended it to lazy evaluation.

The COSTA system [2] follows a different approach, since it generates recurrence equations and provides its own recurrence relation solver, PUBS [1], 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.

In order to better compute memory bounds, we have decided *Safe* to have a heap structured as a region stack in which regions are allocated and deallocated in constant time. Given this memory model, we used in [13] abstract interpretation-based techniques for inferring non-linear, monotonic, closed-form expressions bounding the heap and stack memory costs of a program. *Safe*'s type system [12] also supports *polymorphic recursion on regions*, meaning in essence that the recursive internal calls may use different regions than the external call. The language provides also a *destructive pattern matching* feature which allows the programmer to explicitly dispose data structures, or parts of them. Both features result in programs with less memory consumption. Due to technical difficulties, we have excluded them in this work, but we provide more comments on this in Section 5.

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. Even if we restrict ourselves to a first-order functional language like *Safe*, the inference of safe memory bounds is a very complex task, which involves considering several preliminary results, such as size analysis, and call-tree size analysis. Each one of these analyses is by itself a subject of extensive research. In the analysis developed in [13] the size and call-tree information is given externally.

The full development of the analysis can be found in [16], which is an extended and improved version of [13]. We can summarise the original contributions of [13] as follows: (1) it infers space bounds for a functional language with lexically scoped regions; (2) it uses abstract interpretation directly on the infinite domain of multivariate monotonic functions; and (3) the bounds go beyond multivariate polynomials. The additional contributions of this paper are the following:

- Under certain mild conditions on the externally-given call-tree information, our bounds have the nice property of *reductivity*. This means that if we use a correct bound as an input of the abstract interpretation, we get a new bound which is not only correct but also at least as tight.
- We have formally proved the correctness of all the results.
- We have implemented all the algorithms presented here in our *Safe* compiler.

The proofs of the theorems, including the statement and proof of some auxiliary lemmas, are included in [17].

### 1.1. Plan of the paper

After this introduction, in Section 2 we summarise *Safe*'s features, the abstract intepretation rules, and the space inference algorithms already presented in [13,16]. Then, Section 3 is devoted to the reductivity property. We specify and prove under which conditions this property holds. Detailed examples of reductivity are included here. Section 4 presents some medium-sized case studies and the results obtained for many other functions. Section 5 discusses how it is possible to extend these results when polymorphic recursion and explicit destruction are considered. Finally, Section 6 concludes.

## 2. Preliminaries

### 2.1. The Safe language

*Safe* is a first-order eager functional language with a Haskell-like syntax, but with a different approach to memory management. Instead of using a garbage collector, *Safe*'s heap memory model is based on a combination of *regions* and *explicit destruction*. The latter aspect is controlled by the programmer and it is not considered in this paper. However, in Section 5 we sketch the inference of memory consumption in presence of this feature.

A region is a part of the memory, disjoint from other regions, in which data structures are built. In this work, we define a data structure (DS in the following) as the set of cells that stem from a given one (the root cell) and have the same type. A cell is just a piece of memory containing a constructor and the arguments to which it is applied. A naive implementation of cells will assign a fixed number of bytes to all of them, but more clever implementations are possible [15]. We pose the restriction that a data structure must be contained within a single region. For instance, assume a list of lists of integer values, of type [[*Int*]]. The cons-nil spine of the outer list is a DS which may reside in a region different from those of the inner lists, each of which is a separate DS. The integer values of the inner lists do not belong to any region by themselves; they are contained within cells.

Regions are created and destroyed in a stack-like fashion. There is a correspondence between the function call stack and the region stack. The region associated with a function call to $f$ is called its *working region*. An empty region is allocated