Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 330 (2016) 27–46

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Memory Consumption Analysis for a Functional and Imperative Language

Jérémie Salvucci[1] and Emmanuel Chailloux[2]

Sorbonne Universités,
UPMC Univ. Paris 06,
UMR 7606, LIP6,
4 Place Jussieu, F-75005 Paris, France

Abstract

The omnipresence of resource-constrained embedded systems makes them critical components. Programmers have to provide strong guarantees about their runtime behavior to make them reliable. Among these, giving an upper bound of live memory at runtime is mandatory to prevent heap overflows from happening. The paper proposes a semi-automatic technique to infer the space complexity of ML-like programs with explicit region management. It aims at combining existing formalisms to obtain the space complexity of imperative and purely functional programs in a consistent framework.

Keywords: ML, regions, static analysis, memory analysis.

1 Introduction

Deploying software in constrained environments requires strong guarantees about its runtime behavior. In memory-constrained embedded systems, dynamic allocation is often prohibited to keep execution time analyses doable and avoid heap overflows. We introduce a programming language and a resource consumption analysis to enable dynamic allocation while providing an upper bound of live memory at compile time.

In this paper, we propose a language *à la ML* mixing purely functional and imperative features with an explicit region mechanism. To retrieve information about a program memory interactions, we rely on a static type & effect system and manual memory management through region related primitives. The type system aims at ensuring the absence of memory-related errors at compile time. To

[1] `jeremie.salvucci@lip6.fr`

[2] `emmanuel.chailloux@lip6.fr`

http://dx.doi.org/10.1016/j.entcs.2016.12.013
1571-0661/© 2016 Published by Elsevier B.V.

perform this, programmers have to manage memory manually through a restricted set of primitives describe in section 3. The effect system helps generalize terms and discriminate between purely functional and imperative styles at the function level.

The analysis relies on the correctness of the type system to consider only error-free programs with respect to memory. It combines several existing resource consumption analyses depending on the style inferred by the effect system. For instance, a function which does not allocate memory do not require analysis whereas a function which allocates memory and performs side-effects needs careful handling. On pure functions, we apply automatic amortized analysis [7] adapted to the region mechanism. On imperative functions, regions offer spatial information for side-effects, we use invariants on iteration spaces provided by the programmer as annotations. Both analyses return a symbolic expression characterising the space complexity of the analyzed function for each region involved in the computation. The composition of these symbolic expressions with a careful handling of side-effects give the program memory consumption.

To allocate memory and to reclaim memory are orthogonal operations. Allocating memory does not require information about the current state of the memory graph. Whereas, reclaiming memory requires a global view of the heap to distinguish reachable from unreachable values. In this work, we use regions to gather enough information at compile time to prevent overpessimistic upper bounds by considering regions freed by the programmer in a sound way.

The main goal of this paper is to introduce a framework to combine various memory consumption analyses depending on the programming style used at function level to provide an upper bound of live memory at compile time considering reclaimed memory. In the remainder, related works are presented in section 2. We describe the language in section 3 with its type & effect system on which we base our analysis. Then, we show how to deal with purely functional and imperative features in sections 4, 5 as described above and section 6 composes them in a consistent framework. Then, we show how it works on an example in section 7. Finally, we conclude with a discussion about current limitations and further improvements.

2 Related works

Resource consumption analysis started in the late 70s with METRIC [14] targetting the best, worst and average execution times of programs written in a pure subset of Lisp. Based on recurrence relations, it can be adapted to memory consumption analysis. Contrary to time, memory can be reclaimed. Hence, new methods have emerged from both purely functional and imperative communities to obtain upper bounds on live memory.

Sized types [9] have been applied to the core part of HUME [13], a purely functional language with an eager evaluation mechanism. It infers linear space complexities and provides an upper bound on allocated memory without requiring the user intervention.

Automatic amortized analysis [6], based on Tarjan's work [11], has been used in