Symbolic and Analytic Techniques for Resource Analysis of Java Bytecode

David Aspinall¹, Robert Atkey², Kenneth MacKenzie¹, and Donald Sannella¹

1 School of Informatics, The University of Edinburgh, Edinburgh
2 Computer and Information Sciences, University of Strathclyde, Glasgow

Abstract. Recent work in resource analysis has translated the idea of amortised resource analysis to imperative languages using a program logic that allows mixing of assertions about heap shapes, in the tradition of separation logic, and assertions about consumable resources. Separately, polyhedral methods have been used to calculate bounds on numbers of iterations in loop-based programs. We are attempting to combine these ideas to deal with Java programs involving both data structures and loops, focusing on the bytecode level rather than on source code.

1 Introduction

The ability to move code and other active content smoothly between execution sites is a key element of modern computing platforms. However, it presents huge security challenges, aggravating existing security problems and presenting altogether new ones. One challenging security issue in this context is control of resources (space, time, etc.), particularly on small devices, where computational power and memory are very limited.

A promising approach to security is proof-carrying code [31], whereby mobile code is equipped with an independently verifiable certificate consisting of a condensed proof of its security properties. A major advantage of this approach is that it sidesteps the difficult issue of trust: there is no need to trust either the code producer, or a centralized certification authority. Work on the PCC approach to resource security includes [35] and [7].

This approach requires infrastructure on the side of the code producer as well as the code consumer. The code producer needs to produce not just downloadable code, as before, but also a proof of its security properties. The code consumer needs a way of checking such proofs. Arbitrarily complex methods may be used by the code producer to construct proofs, while their verification by the code consumer is a straightforward check of validity. The burden for the code producer is considerably eased by the use of a certifying compiler which employs static analysis of the source code alongside standard compilation to supply the information required to produce these proofs automatically. The information provided by the analysis — in the case of resource analysis, concerning upper bounds on usage of space, time, etc. — is potentially of great interest to the code producer as an aid to the development of high-quality code, prior to and independent of its use for producing security certificates.
Recent developments in static analysis methods now make it feasible to consider an alternative but related approach to security. Instead of requiring the code producer to supply a proof, whether via static analysis of source code or by other means, one can perform an analogous analysis directly on the downloadable bytecode to determine its properties. This could be done by the code consumer on receipt of downloadable code, dispensing with the need for a proof. Alternatively, the code producer could perform the analysis and use the result to produce a proof certificate. An interesting third alternative is that an intermediary, for example a software distributor, could perform such an analysis on uncertified bytecode, transforming it to proof-carrying code. The fact that the original source code is not required is essential to making this feasible in commercial practice.

Here we consider two quite different approaches to the analysis of resource consumption of Java bytecode. The first, in §2, translates the idea of amortised resource analysis to imperative languages to enable automated resource analysis of programs that iterate through data structures. The second, in §3, uses polyhedral methods to calculate resource bounds of iterative procedures controlled by numerical quantities. In §4 we briefly describe some ideas for future work and plans for integrating the two kinds of analysis to deal with Java programs involving both data structures and loops.

2 Amortised Resource Analysis

Amortised resource analysis is a technique for specifying and verifying resource bounds of programs by exploiting the tight link between the structure of the data that programs manipulate and the resources they consume. For instance, a program that iterates through a list doing something for every element can either be thought of as requiring $n$ resources, where $n$ is the length of list, or as requiring 1 resource for every element of the list, where we never know the global length property of the list. Taking the latter view can simplify both the specification and the verification of programs’ resource usage.

This work conceptually builds on the work of Tarjan and Sleator on amortised complexity analysis [36], where “credits” and “debits” may be virtually stored within data structures and used to pay for expensive operations. By storing up credit for future operations in a data structure, we amortise the cost of operations on the data structure over time. Hofmann and Jost [21] applied this technique to first-order functional programs to yield an automated resource analysis. Atkey [3] has recently adapted this work to integrate with Separation Logic [22,34] to extend the automated technique to pointer-manipulating imperative programs. In this section we give an overview of Atkey’s work and describe some examples.

2.1 Integrating the Banker’s Method and Separation Logic

Separation Logic is built upon a notion of resources and their separation. The assertion $A \ast B$ holds for a resource if it can be split into two resources that