ByteCode 2011

Static Resource Analysis for Java Bytecode Using Amortisation and Separation Logic

Damon Fenacci[1] Kenneth MacKenzie[2]

School of Informatics
The University of Edinburgh
Edinburgh, UK

Abstract

In this paper we describe a static analyser for Java bytecode which uses a combination of amortised analysis and Separation Logic due to Robert Atkey. With the help of Java annotations we are able to give precise resource utilisation constraints for Java methods which manipulate various heap-based data structures.

Keywords: Java, JVM, Bytecode, Resource analysis, Amortisation, Separation Logic

1 Introduction

In [4], Robert Atkey shows how methods from amortised complexity analysis can be combined with Separation Logic to obtain a technique for resource analysis of imperative programs which manipulate heap-based data-structures such as trees and linked lists. He shows how to apply this method to a small stack-based virtual machine, similar to the JVM. In this paper we describe an analyser which applies this analysis to real JVM bytecode, using specifications obtained from programmer-supplied annotations in Java source code.

Outline

We begin with an outline of the ideas involved in the analysis. This is followed by a detailed description of the Java annotations used to communicate the

[1] Email: `D.Fenacci@sms.ed.ac.uk`
[2] Email: `kwxm@inf.ed.ac.uk`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

specifications to our analyser, together with examples of the analyser in action. We also describe some of the methods used in the implementation of the analyser.

Acknowledgments

The work described in this paper was carried out in the RESA project (EPSRC Follow-on Fund grant number EP/G006032/1) at the University of Edinburgh.[3] More information can be found in [3]. We would like to thank Robert Atkey for extensive discussions.

2 Specifying resource consumption

Much of this paper is based on previous work of Robert Atkey [4]. This work is somewhat technical from the point of view of non-experts, and in this section we will attempt to give a non-technical overview. We present a fairly simple example here, but we hope to make an online demonstration available on the RESA webpages including more complex examples.

Specifying heap behaviour

Consider the following Java code:

```
class IntList {
   int head;
   IntList tail;

   IntList concat (IntList p, IntList q) {
      if (p == null) return q;
      else {
         IntList t = p;
         while (t.next != null)
            t = t.next;
         t.next = q;
         return p;
      }

   ...
   }
}
```

This defines a simple class of linked lists with integer entries and a method which concatenates two lists `p` and `q`. If `p` is empty then `append` returns `q`, otherwise a pointer `t` traverses `p` until it reaches the final cell, then adjusts its `next` field to point to `q` and returns `p`.

Suppose that we want to describe the behaviour of `concat`. Let us introduce an assertion `lseg(a,b)` which states that there is a well-formed list segment in the heap for which `a` points to the first cell and `b` points to the final

[3] http://groups.inf.ed.ac.uk/resa/

cell. Intuitively, in the heap we have a picture of the form shown in Figure 1, with all cells distinct.

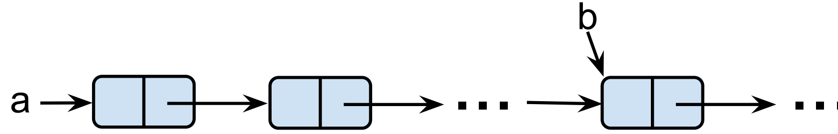

Fig. 1. List segment `lseg(a,b)`.

A first attempt at a specification for the `concat` method as might be as follows [4] :

```
@Requires (lseg(@arg p,null), lseg(@arg q,null))  // precondition
@Ensures (lseg (@ret, null))                       // postcondition
IntList concat (IntList p, IntList q) { ... }
```

The intended meaning of this specification is that *if* when we enter the method, the arguments `p` and `q` point to well-defined list segments, *then* when we reach the end of the method, the return value will also point to a well-defined list segment. The specification may be regarded as a contract with the user: if the inputs to the method satisfy the precondition, then the output is guaranteed to satisfy the postcondition. Ideally, we will be able to *prove* that the implementation of the method does actually guarantee this behaviour.

Unfortunately, there is a problem with the above specification. If `p` and `q` are pointers to the same location in memory (in other words, they are just different names for the same list), then we will end up with a circular structure: we will iterate along to the end of the list pointed to by `p` and then adjust the `next` pointer to point back to the head of the list: see Figure 2.

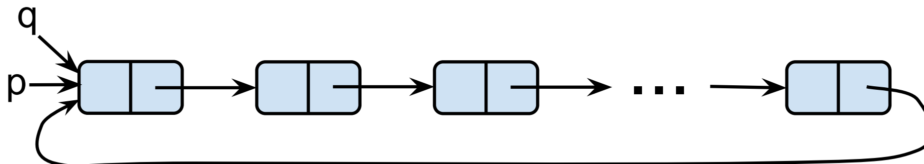

Fig. 2. `concat` result in case of two pointers (`p` and `q`) pointing to the same list segment.

This violates the intended meaning of our `lseg` predicate, and hence the postcondition is false. One might attempt to deal with this by modifying the method to check whether `p==q`, but that would still not work since if the lists pointed to by `p` and `q` share any cells we will still end up with heap structures containing loops (Figure 3). Modifying the method to detect such situations would make it unnecessarily complicated; a better strategy is to amend the assertions to exclude problematic inputs from the outset. The key to this is

[4] For clarity, we have used an idealised annotation syntax here. Restrictions on the syntax of Java annotations mean that the specifications used in practice are somewhat more complex; these will be described in detail in Section 3.

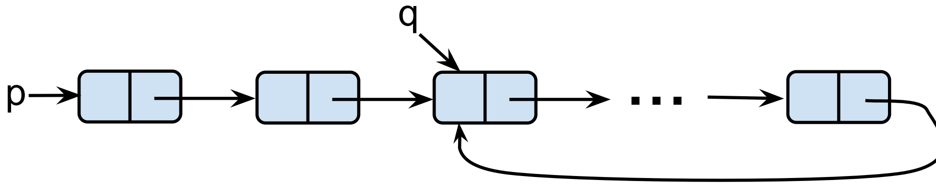

Fig. 3. `concat` result in case of a pointer `q` pointing to an internal element of a list segment pointed by `p`.

to base the assertions on *Separation Logic* [22], a logic which is designed for arguing about non-overlapping structures and has proved very useful in the analysis of heap-allocated data structures in recent years.

Separation Logic has a number of novel logical connectives for arguing about non-overlapping objects. For example, in addition to the usual logical conjunction $\wedge$ ($A \wedge B$ means that A is true and B is true), Separation Logic has the *separating conjunction* $*$: $A * B$ is true if A is true and B is *separately* true. In the context of heap-allocated structures $A * B$ will be interpreted as meaning that A and B are both true, but on *disjoint regions of the heap*. See [22] for full details of Separation Logic.

If we use the separating conjunction to rewrite our original specification as

```
@Requires (lseg(@arg p,null) * lseg(@arg q,null))
@Ensures (lseg (@ret, null))
```

then it becomes valid: if the user supplies two well-formed lists which share no memory cells then the return value is also a well-formed list.

Amortised analysis with Separation Logic

The techniques of the previous section allow us to specify functional properties of methods which manipulate heap-allocated structures. However, our main interest is in the resource consumption of methods, where "resource" refers to some quantity which is consumed by the method: for example, we might consider the number of heap objects allocated by a method, the number of bytecode instructions executed, or the number of network packets sent. In this paper, we will consider the problem of finding the number of times a special method called `consume` is executed, but this can easily be replaced by other methods or bytecode instructions in order to deal with other resources.

In [4], it is shown that assertions in Separation Logic can be neatly extended to include information about resource consumption, and that it is possible both to verify annotations and to infer resource usage for methods where iteration is driven by the processing of heap allocated data structures.

This is based on the idea of *amortised analysis* [26] of algorithms involving data structures. The approach we will take here is to imagine that each node of a data structure is equipped with a number of tokens, and that one of these

is consumed each time the node is processed. Consider our previous example, with some calls to `consume` added:

```
IntList concat (IntList p, IntList q) {
   consume();
   consume();

   if (p == null) return q;
   else {
      IntList t = p;
      while (t.next != null) {
         t = t.next;
         consume();
      }
      t.next = q;
      return p;
   }
```

We see that `consume` is called twice at the start of the method and then once for each node in the list `p`. We can express this by extending our Separation Logic specifications to include costs:

```
@Requires (lseg(1, @arg p,null) * lseg(0, @arg q,null), 2)
@Ensures (lseg (0, @ret, null), 0)
```

The extra numeric annotations are interpreted as saying that if we enter the method with one token for each node of `p` plus two extra tokens then the method can successfully execute and we are left with no tokens at the end; since each call to `consume` requires one token, we see that `consume` is called at most `length(p)+2` times.

We do not require that the annotations specify the minimal number of tokens required, only a number which is sufficient to allow the method to complete. For example

```
@Requires (lseg(7, @arg p,null) * lseg(2, @arg q,null), 5)
```

would also be a valid precondition: if we have the specified number of tokens then the method is still able to complete, but this time we will have some tokens left over which would enable further processing of the result at a later stage. The left-over tokens can be included in the postcondition. For example

```
@Requires (lseg(7, @arg p,null) * lseg(2, @arg q,null), 5)
@Ensures (lseg(2, @ret, null), 3)
```

is also a valid (albeit non-optimal) specification.

Atkey shows that it is possible to use a linear programming technique based on ideas of Hofmann and Jost [15] to verify that resource annotations such as those above are valid; in fact, he shows that it is actually possible to *infer* minimal resource annotations and thus the resource consumption of a method. We will see examples of this later.

3 Amortised Analysis for Java bytecode

We have developed an analyser which implements the ideas of the previous section. Source programs are equipped with Java annotations giving preconditions and postconditions of the type described above; the current version also requires annotations giving loop invariants.

Our analyser works on compiled class files. The Java compiler stores source annotations in the class file, and the analyser retrieves these and uses them to perform the analysis on JVM bytecode.

The technique of analysing the bytecode rather than the source code has several advantages:

- The bytecode is what is actually executed. We do not need any knowledge of the inner workings of a particular compiler, and indeed the analyser is independent of the compiler used.
- This approach extends the JVM verification paradigm. Many Java applications are supplied in the form of compiled classfiles with no source code, and a bytecode analyser can be used to check the behaviour of the class prior to execution. There is no requirement for the user of the code to trust the supplier.
- The analysis is not restricted to bytecode obtained from Java source; in principle our analysis could be made work on bytecode obtained from other languages targeting the JVM (Scala [5], for example), or even on handwritten bytecode.

Java annotations for amortised analysis

We use Java annotations to equip methods with resource-usage specifications. In this section we will expand on the informal description given in earlier sections.

Java annotations are a specific form of metadata that can be added to Java source code. They can be associated with Java classes, fields, methods and method parameters, and are embedded in classfiles when compiling Java code. They are defined using class-like structures in files named after the annotation.

In order to be able to provide amortised analysis on methods we have defined three annotations. These allow user to specify method preconditions and postconditions together with loop invariants which are required by the analyser.

- `@Requires(assertions)` for preconditions

[5] http://www.scala-lang.org/

```
assertions = [exvars] assertion ('||' assertion)* ;
assertion = '{' data-assertions '|'
                heap-assertions '|'
                resource-assertion '}' ;

exvars = 'exists' typedvar + '.' ;
typedvar = id ':' type ;
     (* id is a Java identifier: [A-Za-z_][A-Za-z0-9_]* *)
type = 'int' | 'long' | 'float' | 'double' | 'ref' ;

data-assertions = data-assertion (',' data-assertion)* ;
data-assertion  =
     term '==' term
   | term '!=' term ;

heap-assertions = heap-assertion (',' heap-assertion)* ;
heap-assertion =
     '[' field '->' location ']'
   | 'lseg (' resource-assertion ',' term ',' term ')'
   | 'tree (' resource-assertion ',' term ')' ;


term =
     id             (* existential variable *)
   | '@arg' id (* only @arg variables allowed in preconditions *)
   | '@var' id
   | 'null'
   | '@ret' ;  (* @ret only allowed in postcondition *)

field = '(' term ')' '.' id ':' type ;

location = var | '?' ;

resource-assertion = linear-expression ;
```

Fig. 4. Assertion EBNF

- `@Ensures(assertions)` for postconditions
- `@Invariant(assertions)` for loop invariants

All of them are given in the form of a Java `String` containing assertions. A shortened version of the assertion syntax in EBNF is shown in Fig. 4.

Data assertions consist of comma-separated lists of assertions of the form `term == term` or `term != term` and are used to provide the analyser with information about when references point to the same or different Java objects. These are mostly required in loop invariants. It would be possible to use dataflow analysis to deduce this information, but we have not done this yet.

There are two types of *heap assertion*, which are Separation Logic predicates enriched with indications of field content. The first type of heap assertion indicates that part of the heap forms either a list segment or a tree (e.g. `lseg(r1, @arg x, null)`, `tree(r1, @arg x)`); both of these expand to more complex assertions built from Separation Logic primitives. The second type of heap assertion indicates to the analyser that a particular field points to a particular heap location or to some undetermined location; these are mostly required in preconditions and postconditions, where they facilitate

interprocedural analysis by exposing information about heap structure for use in reasoning with Separation Logic.

Finally, *resource assertions* are linear expressions such as `3*r1 + 5*r2 + r3 + 7` which specify constants and variables which we want the analyser to use to infer the resources associated with the method before and after execution (ie, in the precondition and postcondition); they are also used in list and tree predicates to indicate resources associated with each node of the structure.

We have also introduced a dummy method called `consume` which does nothing except tell the static analyser that at the point where it is introduced, a unit of resource is being used. We can imagine such a dummy method being hidden inside library code in the future, stating the amount of resource used by each method defined in each class, so that programmers will not need to add it explicitly but rather implicitly use it by invoking library methods. In the present implementation though, libraries have not been modified and developers have to specify resource consumption explicitly in their code.

Invariant localisation in Java code

Loop invariants have to be given for each loop for the amortised analysis to be effective but the Java language does not allow the inclusion of annotations inside the code. This is problematic if the method being analysed contains two or more loops, since we need a way to decide which invariant refers to which loop. We could simply give invariants in the same order as the loops appear in the code, but it is possible that a compiler might produce bytecode in which the order in which the loops appear in the bytecode does not correspond to the order in the source code. Our way to obviate this limitation was to identify each loop with an integer identifier. Each loop invariant is given a identifier (`@Invariant(id, assertion)`) and the same identifier has to match the argument of a dummy method `Loop.invariant(id)` placed just before the loop in the code. Thus by searching the code, the analyser can associate the declared invariants with the corresponding loop.

4 Examples and output interpretation

We illustrate the operation of the analyser by returning to our earlier list concatenation example.

```
$ cat examples/IntList.java

import uk.ac.ed.inf.resa.*;

public class IntList {

  private int data;
  private IntList next;
```

```
  ...

  @Requires("{ | lseg(1, @arg p, null) * lseg (0, @arg q, null)| 2 }")
  @Ensures("{ | lseg(0, @ret, null) | 0 }")
  @Invariant("{ @var t != null | lseg (0, @var p, @var t) *"
             + "lseg (1, @var t, null) * lseg (0, @var q, null) | 0}")
  public static IntList concat (IntList p, IntList q) {
    Amortised.consume();
    Amortised.consume();
    if (p==null) return q;

    IntList t = p;
    Loop.invariant (0);
    while (t.next != null) {
      t = t.next;
      Amortised.consume();
    }
    t.next = q;

    return p;
  }
  ...
}
```

We have supplied a loop invariant which describes the state of resource usage as the program progresses. Whenever we reach the head of the `while` loop, we have used up the resources associated with the part of the first input list `p` which has already been processed (between `p` and `t`), we still have one unit of resource available in the remaining part of `p` (between `t` and `null`), and we do not require any resources to process `q`. Note the `@arg` and `@var` annotations attached to variable names. These are used to distinguish between the current value of a variable (`@var`) and the value of a method argument at entry to the method (`@arg`); they are not strictly necessary in this example, but are required in more complex examples where variables representing method arguments are modified. The `@ret` annotation refers to the method return value.

If we compile `IntList.java` and invoke the analyser then the output is as follows.

```
$ resa -amortised examples/IntList concat
...
Solved VCs
Verification successful
```

This shows that the analyser has succeeded in proving that the precondition and postcondition are satisfied. However, if we amend the precondition to say

```
@Requires("{ | lseg(1, @arg p, null) * lseg (0, @arg q, null)| 1 }")
```

then the verification fails because there is only one "constant" unit of resource available, and two are required by the calls to `consume` at the start of the method:

```
LP is infeasible
```

More interestingly, we can supply *generic* annotations and the analyser will infer suitable values for them.

```
@Requires("{ | lseg(x1, @arg p, null) * lseg (x2, @arg q, null)| x3 }")
@Ensures("{ | lseg(y1, @ret, null) | y2 }")
@Invariant("{ @var t != null | lseg (z1, @var p, @var t) *"
            + "lseg (z2, @var t, null) * lseg (z3, @var q, null) | z4}")
```

With these annotations the analyser outputs

```
Optimal solution for resource variables:
x1 = 1, x2 = 0, x3 = 2
y1 = 0, y2 = 0
z1 = 0, z2 = 1, z3 = 0, z4 = 0
```

We can also specify the amount of resource which we require when the method returns: if we replace the postcondition with `@Ensures("{ | lseg(3, @ret, null) | 7 }")` then we get `x1 = 4, x2 = 3, x3 = 9`, so that if the postcondition is to be satisfied then we must have 4 units of resource for each element of `p`, 3 for each element of `q`, and 9 extra units before the method is called.

In addition to this example we have been able to successfully analyse a number of other standard operations on lists, such as reversal, iteration, and deleting and inserting elements, together with similar operations for trees.

5 Implementation details

Our analyser is implemented in OCaml, and Java class files are represented by a collection of datatypes. For program analysis, the most important part of this is the representation of method bytecode. Our design is intended to be fairly general-purpose since we intend to support multiple analysis techniques, including the amortised analysis described earlier.

Java classfiles are initially converted into a low-level OCaml representation which is a fairly faithful representation of structure of the class as described in the JVM specification [19]. This is then converted into a higher-level representation which is more suited to analysis. We will give an outline of this representation here, but space limitations preclude a detailed description.

Bytecode instructions are decompiled into a form where the JVM stack has been eliminated. We keep track of which constants and local variables have been loaded onto the stack, and these are represented by a datatype called `value`, which has constructors for constants of type `int`, `long`, `float`, `double`, `String` and `class`, together with variables. Variables are represented by integer identifiers which are tagged with the type of the corresponding value: this is one of `I`, `L`, `F`, `D`, or `A` (representing integers, long integers, floats, doubles, and addresses). We do not have special types for `boolean`, `byte`, `char` and `short` since the JVM treats all of these as 32-bit integers for most purposes. We also tag all references with the single type `A`, and make no attempt to keep track of the most specific class or interface to which the variable belongs; it would be possible to infer this information, but so far we have not required this.

There are two kinds of variables: *local* variables and *stack* variables. The former represent values loaded onto the stack from JVM local variables by means of instructions such as `iload`, and the latter represent intermediate values which have been created on the stack by arithmetic operations, method calls and so on. Each variable is marked with an integer which for local variables represents the number of the associated JVM local variable, and for stack variables is simply a counter which is incremented every time a new value is created on the stack and decremented when that value is consumed. Some care is required here since stack operations can duplicate values on the stack. The decompilation process handles this by creating new copies of variables using a special pseudo-instruction called `Copy`. Another complication is that one can load a local variable r onto the stack and then modify the contents of r before the earlier value (still on the stack) has been consumed; again, the `Copy` operation is used to avoid confusion by creating a new variable which represents the earlier value

Instructions which act on the stack are represented by a datatype of operations which take values as arguments. This has 19 constructors which suffice to represent all of the JVM operations (`putfield`, `getfield`, `invokevirtual` and so on) except for those which involve control-flow transfer. Basic blocks are represented by a list of pairs consisting of operations together with the local variable or stack location where the result (if any) of the operation is to be stored. At the end of a basic block we have a member of a `continuation` datatype: this represents various types of jump, comparison, switch, and return operation, together with information specifying which blocks control can flow to after leaving the current block. We do not provide any representation for the `jsr` and `ret` instructions used by exception handlers, since these complicate analysis and are supposedly deprecated in current Java releases (although we have encountered them in a few of the standard API classes in `rt.jar`). If one of these instructions is encountered during decompilation then an exception is thrown and the class is rejected.

The bytecode for an entire method is represented by an array of blocks, stored in preorder. This can be regarded as a graph, and we have a module which computes useful information such as successors, predecessors, and dominators, and can also provide other views of the graph, such as postorder and reverse postorder, which can be useful in some analyses.

Proof search

The most important part of the amortised analysis is the proof search procedure which is used to verify that the precondition of a method implies its postcondition, and also to check or infer the associated resource annotations.

This uses the method described by Atkey in [4], and indeed our implemen-

tation uses an adapted version of code from a prototype implementation by Atkey, which analysed a textual form of a subset of JVM bytecode. The basic idea is to visit the code in postorder, working backwards from the postcondition to infer weakest preconditions for each instruction, and then to prove that the weakest precondition for the first instruction is implied by the (user-supplied) precondition for the method.

Visiting the nodes of the CFG in postorder ensures that when we visit a given node n, the preconditions for all of its successors are already known, and can thus be combined (by logical conjunction) to obtain a postcondition for n. However, this strategy fails when we encounter a back-edge $s \to t$ (ie, when t is at the head of a loop). In this case we will not have visited t when we arrive at s, and it is for this reason that we require annotations for loop invariants: the annotation will provide a precondition for t, and hence will be available to contribute to the postcondition for t.

The proof search procedure also collects linear constraints describing the resource usage of the bytecode instructions, as described in [4]; once the analysis has been completed and the basic Separation Logic annotations have been checked to make sure that the precondition implies the postcondition, the resource constraints are converted into a linear programming problem which is then solved using the Parma Polyhedra Library [5].

6 Conclusions and Further Research

We have shown that it is possible to apply Atkey's amortised analysis technique to realistic JVM bytecode. However, there are a number of issues that would merit further research.

- The annotations are somewhat complex, and it would be desirable to simplify them. A first step would be to dispense with the data equality and non-equality annotations, and we believe that it would be possible to do this using dataflow techniques. A more difficult problem would be to remove the loop invariants, which are generally the most difficult part of the specification. Techniques for automatically inferring loop invariants have been studied, and one possible avenue of attack for the kind of loop invariants used here would be the methods described in [10].
- A minor inconvenience is that assertion syntax is only checked during analysis, and not at compilation time. The Sun `javac` compiler supports plugins for annotation processing [6], and it would be very helpful to use this feature to detect errors in assertion syntax during compilation.
- Amortisation techniques are useful for analysing the resource consumption of loops which process heap-allocated structures, but may be less useful for

[6] See JSR 269: Pluggable Annotation Processing API.

loops which are controlled by numeric quantities. Our analyser can attack such loops by using combinatorial techniques for the enumeration of lattice points in polyhedra (see [6], and [9] for an application of related techniques to resource analysis for Java programs); a preliminary description of this appears in [3], and we will give more details in a subsequent paper. An interesting problem would be to automatically combine the two techniques, so that the analyser can deal with complex methods with minimal intervention from the user.

- The amortised analysis can only handle bounds on resource usage which are linear in the size of the data structures involved. The experience of other authors [8] suggests that this is sufficient in many (but by no means all) practical situations. The linear-programming-based inference technique we use here is intrinsically tied to linear bounds; nevertheless, recent work [14,13] has shown how it can be extended to deduce non-linear bounds in certain specialised cases. There are other inference techniques which can deal with non-linear bounds, and we will discuss some of these later.
- The analyser currently only supports linked lists and trees, and the proof-search implementation depends quite strongly on this. It would be desirable to find more generic annotation and analysis techniques, for example for dealing with data structures in the standard Java API which are accessed via interfaces. One method here might be to attach annotations to existing library classes in order to enable the analysis of client code.

Related work: resource inference

There has been a considerable amount of work on resource inference in recent years, and we now attempt to describe some of this.

The linear-programming-based inference technique used here originated with Hofmann and Jost in [15], and has subsequently been applied to an OCaml-style language [23] and the Hume language for embedded systems [8]. Jost's thesis [17] contains a useful overview of this area of research, and recasts much of the existing work in a consistent and systematic framework. Some more recent developments appear in [14,13].

A number of other resource-inference techniques have been proposed and implemented, for Java and also for other languages. The COSTA system of Albert et al. [1,2] converts JVM bytecode into a collection of "cost equations" which are then solved to obtain a symbolic (and possibly non-linear) bound for resource usage. In contrast with our method, COSTA is fully automatic and does not require the user to supply annotations.

The SPEED project of Gulwani et al. [11,12] uses a number of techniques (including abstract interpretation and SMT solving) to obtain (non-linear and symbolic) bounds for the number of times a program location is visited, and

has been applied to the complexity analysis of C# programs.

Type-theoretic methods for resource usage inference are described in [21] (inference of symbolic bounds for recursive functional programs) and [20] (heap usage for object-oriented programs).

Related work: static analysis of bytecode

A number of representations for JVM bytecode have appeared in the literature. Many of these ([18], [7], and [2] for example) utilise the technique of introducing new variables to represent values on the JVM operand stack. The Soot framework (see [27] for example) contains a number of intermediate representations for JVM bytecode and has been applied to many problems in optimisation and analysis; a large list of related publications can be found at `http://www.sable.mcgill.ca/`. Another framework for JVM bytecode analysis is Julia, which is described in [25]. Finally, another OCaml representation for JVM classfiles is described in [16]; this has much in common with our representation.

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, March 2007.

[2] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2008.

[3] David Aspinall, Robert Atkey, Kenneth MacKenzie, and Donald Sannella. Symbolic and analytic techniques for resource analysis of Java bytecode. In *Proceedings of the 5th international conference on Trustworthly Global Computing*, TGC'10, pages 1–22, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Robert Atkey. Amortised resource analysis with separation logic. In *ESOP*, pages 85–103, 2010.

[5] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[6] Alexander Barvinok and James E. Pommersheim. An algorithmic theory of lattice points in polyhedra. In *New perspectives in algebraic combinatorics (Berkeley, CA, 1996–97)*, volume 38 of *Math. Sci. Res. Inst. Publ.*, pages 91–147. Cambridge Univ. Press, Cambridge, 1999.

[7] Lennert Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. In *Foundations of Global Computing: Proceedings of the 2nd EATCS Workshop*, number 85.1 in Electronic Notes in Theoretical Computer Science. Elsevier, June 2003.

[8] Armelle Bonenfant, Christian Ferdinand, Kevin Hammond, and Reinhold Heckmann. Worst-case execution times for a purely functional language. In *Proc. Implementation of Functional Languages (IFL 2006)*, volume 4449 of *Lecture Notes in Computer Science*, 2007. To appear.

[9] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, Jun 2006.

[10] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

[11] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[12] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 292–304, New York, NY, USA, 2010. ACM.

[13] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (POPL'11)*, 2011. To appear.

[14] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential - A Static Inference of Polynomial Bounds for Functional Programs. In *In Proceedings of the 19th European Symposium on Programming (ESOP'10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

[15] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, pages 185–197, 2003.

[16] Laurent Hubert, Nicolas Barré, Frédéric Besson, Delphine Demange, Thomas P. Jensen, Vincent Monfort, David Pichardie, and Tiphaine Turpin. Sawja: Static analysis workshop for Java. *CoRR*, abs/1007.3353, 2010.

[17] Steffen Jost. *Automated Amortised Analysis*. PhD thesis, Ludwig-Maximilians-Universität, München, August 2010.

[18] Christopher League, Valery Trifonov, and Zhong Shao. Functional Java bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[19] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[20] Wei ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory usage verification for oo programs. In *In SAS 05*, pages 70–86. Springer, 2005.

[21] Álvaro J. Rebón Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro Vasconcelos. Cost analysis using automatic size and time inference. In *Proceedings of the 14th international conference on Implementation of functional languages*, IFL'02, pages 232–247, Berlin, Heidelberg, 2003. Springer-Verlag.

[22] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[23] Donald Sannella, Martin Hofmann, David Aspinall, Stephen Gilmore, Ian Stark, Lennart Beringer, Hans-Wolfgang Loidl, Kenneth MacKenzie, Alberto Momigliano, and Olha Shkaravska. Mobile Resource Guarantees (project evaluation paper). In *Trends in Functional Programming*, pages 211–226, 2005.

[24] Zhong Shao and Benjamin C. Pierce, editors. *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. ACM, 2009.

[25] Fausto Spoto. JULIA: A generic static analyser for the Java bytecode. Proceedings of FTfjP'2005, Glasgow, 2005.

[26] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[27] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.