Customizable Resource Usage Analysis for Java Bytecode

Jorge Navas,[1] Mario Méndez-Lojo,[1] Manuel V. Hermenegildo[1,2]

[1] Dept. of Computer Science, University of New Mexico (USA)
[2] Dept. of Computer Science, Tech. U. of Madrid (Spain) and IMDEA-Software

Abstract. Automatic cost analysis of programs has been traditionally studied in terms of a number of concrete, predefined *resources* such as execution steps, time, or memory. However, the increasing relevance of analysis applications such as static debugging and/or certification of user-level properties (including for mobile code) makes it interesting to develop analyses for resource notions that are actually application-dependent. This may include, for example, bytes sent or received by an application, number of files left open, number of SMSs sent or received, number of accesses to a database, money spent, energy consumption, etc. We present a fully automated analysis for inferring upper bounds on the usage that a Java bytecode program makes of a set of *application programmer-definable* resources. In our context, a resource is defined by programmer-provided annotations which state the basic consumption that certain program elements make of that resource. From these definitions our analysis derives functions which return an upper bound on the usage that the whole program (and individual blocks) make of that resource for any given set of input data sizes. The analysis proposed is independent of the particular resource. We also present some experimental results from a prototype implementation of the approach covering an ample set of interesting resources.

1 Introduction

The usefulness of analyses which can infer information about the costs of computations is widely recognized since such information is useful in a large number of applications including performance debugging, verification, and resource-oriented specialization. The kinds of costs which have received most attention so far are related to execution steps as well as, sometimes, execution time or memory (see, e.g., [21, 28, 29, 16, 8, 17, 32] for functional languages, [30, 7, 15, 34] for imperative languages, and [13, 12, 14, 26] for logic languages). These and other types of cost analyses have been used in the context of applications such as granularity control in parallel and distributed computing (e.g., [23]), resource-oriented specialization (e.g., [10, 27]), or, more recently, certification of the resources used by mobile code (e.g., [11, 4, 9, 3, 18]). Specially in these more recent applications, the properties of interest are often higher-level, user-oriented, and application-dependent rather than (or, rather, in addition to) the predefined, more traditional costs such as steps, time, or memory. Regarding the object of certification, in the case of mobile code the certification and checking process is often performed at the bytecode level [22], since, in addition to other reasons of syntactic convenience, bytecode is what is most often available at the receiving (checker) end.

We propose a fully automated framework which infers upper bounds on the usage that a Java bytecode program makes of *application programmer-definable* resources. Examples of such programmer-definable resources are bytes sent or received by an application over a socket, number of files left open, number of SMSs sent or received, number of accesses to a database, number of licenses consumed, monetary units spent, energy consumed, disk space used, and of course, execution steps (or bytecode instructions), time, or memory. In our context, resources are defined by programmers by means of *annotations*. The annotations defining each resource must provide for some user-selected elements corresponding to the bytecode program being analyzed (classes, methods, variables, etc.), a value that describes the cost of that element for that particular resource. These values can be constants or, more generally, functions of the input data sizes. The objective of our analysis is then to statically derive from these elementary costs an upper bound on the amount of those resources that the program as a whole (as well as individual blocks) will consume or provide.

Our approach builds on the work of [13, 12] for logic programs, where cost functions are inferred by solving recurrence equations derived from the syntactic structure of the program. Also, most previous work deals only with concrete, traditional resources (e.g., execution steps, time, or memory). The analysis of [26] is parametric but it is designed for Prolog and works at the source code level, and thus cannot be applied to Java bytecode due to particularities like virtual method invocation, unstructured control flow, assignment, the fact that statements are low-level bytecode instructions, the absence of backtracking (which has a significant impact on the method used in [26]), etc. More importantly, the presentation of [26] is descriptive in contrast to the concrete algorithm provided herein. In [1], a cost analysis is described that does deal with Java bytecode and is capable of deriving cost relations which are functions of input data sizes. However, while the approach proposed can conceptually be adapted to infer different resources, for each analysis developed the measured resource is fixed and changes in the implementation are needed to develop analyses for other resources. In contrast, our approach allows the application programmer to define the resources through annotations in the Java source, and without changing the analyzer in any way. In addition, the presentation in [1] is again descriptive, while herein we provide a concrete, memo table-based analysis algorithm, as well as implementation results.

2 Overview of the Approach

We start by illustrating the overall approach through a working example. The Java program in Fig. 1 emulates the process of sending text messages within a cell phone. The source code is provided here just for clarity, since the analyzer works directly on the corresponding bytecode. The phone (class `CellPhone)` receives a list of packets (`SmsPacket`), each one containing a single SMS, encodes them (`Encoder`), and sends them through a stream (`Stream`). There are two types of encoding: `TrimEncoder`, which eliminates any leading and trailing white spaces, and `UnicodeEncoder`, which converts any special character into its Unicode($\backslash uxxxx$) equivalent. The length of the SMS which the cell phone ultimately sends through the stream depends on the size of the encoded message.

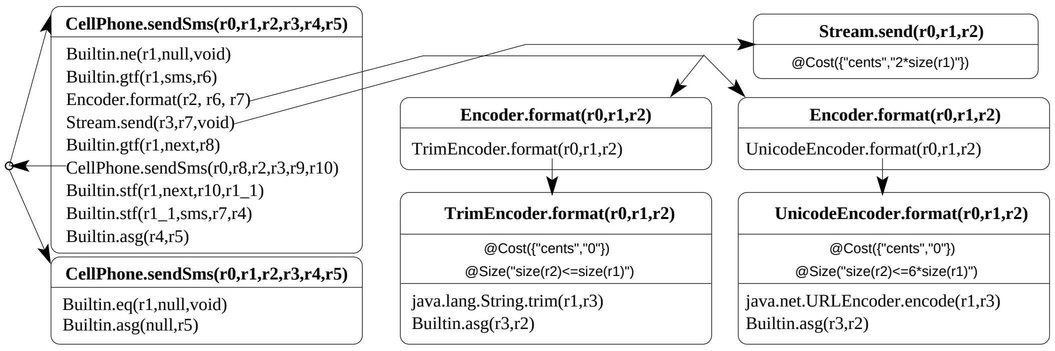
A *resource* is a fundamental component in our approach. A resource is a user-defined notion which associates a basic cost function with some user-selected elements (class, method, statement) in the program. This is expressed by adding Java annotations to the code. The

```
import java.net.URLEncoder;

public class CellPhone {

 SmsPacket sendSms(SmsPacket smsPk,
                   Encoder enc,
                   Stream stm) {
   if (smsPk != null) {
     String newSms = enc.format(smsPk.sms);
     stm.send(newSms);
     smsPk.next=sendSms(smsPk.next,enc,stm);
     smsPk.sms = newSms;
   }
   return smsPk;
 }
}
class SmsPacket{
  String sms;
  SmsPacket next;
}
```

```
interface Encoder{
  String format(String data);
}
class TrimEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=size(s)")
  public String format(String s){
    return s.trim();
  }
}
class UnicodeEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=6*size(s)")
  public String format(String s){
    return URLEncoder.encode(s);
  }
}
abstract class Stream{
  @Cost({"cents","2*size(data)"})
  native void send(String data);
}
```



```
CellPhone.sendSms(r0,r1,r2,r3,r4,r5)
Builtin.ne(r1,null,void)
Builtin.gtf(r1,sms,r6)
Encoder.format(r2, r6, r7)
Stream.send(r3,r7,void)
Builtin.gtf(r1,next,r8)
CellPhone.sendSms(r0,r8,r2,r3,r9,r10)
Builtin.stf(r1,next,r10,r1_1)
Builtin.stf(r1_1,sms,r7,r4)
Builtin.asg(r4,r5)

CellPhone.sendSms(r0,r1,r2,r3,r4,r5)
Builtin.eq(r1,null,void)
Builtin.asg(null,r5)

Stream.send(r0,r1,r2)
@Cost({"cents","2*size(r1)"})

Encoder.format(r0,r1,r2)
TrimEncoder.format(r0,r1,r2)

Encoder.format(r0,r1,r2)
UnicodeEncoder.format(r0,r1,r2)

TrimEncoder.format(r0,r1,r2)
@Cost({"cents","0"})
@Size("size(r2)<=size(r1)")
java.lang.String.trim(r1,r3)
Builtin.asg(r3,r2)

UnicodeEncoder.format(r0,r1,r2)
@Cost({"cents","0"})
@Size("size(r2)<=6*size(r1)")
java.net.URLEncoder.encode(r1,r3)
Builtin.asg(r3,r2)
```

Fig. 1. Motivating example: Java source code and Control Flow Graph

objective of the analysis is to approximate the usage that the program makes of the resource. In the example, the resource is the cost in cents of a dollar for sending the list of text messages, since we will assume for simplicity that the carrier charges are proportional (2 cents/character) to the number of characters sent. This domain knowledge is reflected by the user in the method that is ultimately responsible for the communication (`Stream.send`), by adding the annotation `@Cost({"cents","2*size(data)"})`. Similarly, the formatting of an SMS done in any implementation of `Encoder.format` is free, as indicated by the `@Cost({"cents","0"})` annotation. The analysis understands these resource usage expressions and uses them to infer a safe upper bound on the total usage of the program.

Step 1: Constructing the Control Flow Graph. In the first step, the analysis translates the Java bytecode into an intermediate representation building a Control Flow Graph (CFG). Edges in the CFG connect *block methods* and describe the possible flows originated from conditional jumps, exception handling, virtual invocations, etc. A (simplified) version of the CFG corresponding to our code example is also shown in Fig. 1.

The original `sendSms` method has been compiled into two block methods that share the same signature: class where declared, name (`CellPhone.sendSms`), and number and type of the formal parameters. The bottom-most box represents the base case, in which we return null, here represented as an assignment of `null` to the return variable r_5; the sibling corresponds to the recursive case. The virtual invocation of `format` has been transformed into a static call to a block method named `Encoder.format`. There are two block methods which are compatible in signature with that invocation, and which serve as proxies for the intermediate representations of the interface implementations in `TrimEncoder.format` and `UnicodeEncoder.format`. Note that the resource-related annotations have been carried through the CFG and are thus available to the analysis.

Step 2: Inference of Data Dependencies and Size Relationships. The algorithm infers in this phase *size relationships* between the input and the output formal parameters of every block method. For now, we can assume that size of (the contents of) a variable is the maximum number of pointers we need to traverse, starting at the variable, until `null` is found. The following equations are inferred by the analysis for the two `CellPhone.sendSms` block methods :

$$\begin{array}{ll} \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) & \leq 0 \\ \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) & \leq 7 \times s_{r_1} - 6 + \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3}) \end{array}$$

The size of the returned value r_5 is independent of the sizes of the input parameters *this*, *enc*, and *stm* (s_{r_0}, s_{r_2} and s_{r_3} respectively) but not of the size s_{r_1} of the list of text messages *smsPk* (r_1 in the graph). Such size relationships are computed based on *dependency graphs*, which represent data dependencies between variables in a block, and user annotations if available. In the example in Fig. 1, the user indicates that the formatting in `Unicode-Encoder` results in strings that are at most six times longer than the ones received as input `@Size("size(ret)<=6*size(s)")`, while the trimming in `TrimEncoder` returns strings that are equal or shorter than the input (`@Size("size(ret)<=size(s)")`). The equation system shown above is approximated by a recurrence solver included in our analysis in order to obtain the closed form solution $\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 3.5 \times s^2_{r_1} - 2.5 \times s_{r_1}$.

Step 3: Resource Usage Analysis. In the this phase, the analysis uses the CFG, the data dependencies, and the size relationships inferred in previous steps to infer a resource usage equation for each block method in the CFG (possibly simplifying such equations) and obtain closed form solutions (in general, approximated –upper bounds). Therefore, the objective of the resource analysis is to statically derive safe upper bounds on the amount of resources that each of the block methods in the CFG consumes or provides. The result given by our analysis for the monetary cost of sending the messages (`CellPhone.sendSms`) is

$$\begin{array}{ll} Cost_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) & \leq 0 \\ Cost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) & \leq 12 \times s_{r_1} - 12 + Cost_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3}) \end{array}$$

i.e., the cost is proportional to the size of the message list (`smsPk` in the source, r_1 in the CFG). Again, this equation system is solved by a recurrence solver, resulting in the closed formula $Cost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 6 \times s^2_{r_1} - 6 \times s_{r_1}$.

3 Intermediate program representation

Analysis of a Java bytecode program normally requires its translation into an intermediate representation that is easier to manipulate. In particular, our decompilation (assisted by the Soot [31] tool) involves elimination of stack variables, conversion to three-address statements, static single assignment (SSA) transformation, and generation of a Control Flow Graph

(CFG) that is ultimately the subject of analysis. The decompilation process is an evolution of the work presented in [25], which has been successfully used as the basis for other (non resource-related) analyses [24]. Our ultimate objective is to support the full Java language but the current transformation has some limitations: it does not yet support reflection, threads, or runtime exceptions. The following grammar describes the intermediate representation; some of the elements in the tuples are named so we can refer to them as $node.\mathsf{name}$.

$$
\begin{array}{lll}
CFG & ::= & Block^{+} \\
Block & ::= & (\mathsf{id}{:}\mathbb{N}, \mathsf{sig}{:}Sig, \mathsf{fpars}{:}Id^{+}, \mathsf{annot}{:}expr^{*}, \mathsf{body}{:}Stmt^{*}) \\
Sig & ::= & (\mathsf{class}{:}Type, \mathsf{name}{:}Id, \mathsf{pars}{:}Type^{+}) \\
Stmt & ::= & (\mathsf{id}{:}\mathbb{N}, \mathsf{sig}{:}Sig, \mathsf{apars}{:}(Id|Ct)^{+}) \\
Var & ::= & (\mathsf{name}{:}Id, \mathsf{type}{:}Type)
\end{array}
$$

The Control Flow Graph is composed of *block methods*. A block method is similar to a Java method, with some particularities: a) if the program flow reaches it, every statement in it will be executed, i.e, it contains no branching; b) its signature might not be unique: the CFG might contain several block methods in the same class sharing the same name and formal parameter types; c) it always includes as formal parameters the returned value *ret* and, unless it is static, the instance self-reference *this*; d) for every formal parameter (*input* formal parameter) of the original Java method that might be modified, there is an extra formal parameter in the block method that contains its final version in the SSA transformation (*output* formal parameter); e) every statement in a block method is an invocation, including builtins (assignment `asg`, field dereference `gtf`, etc.), which are understood as block methods of the class `Builtin`.

As mentioned before, there is no branching within a block method. Instead, each conditional `if` *cond* $stmt_1$ `else` $stmt_2$ in the original program is replaced with an invocation and two block methods which uniquely match its signature: the first block corresponds to the $stmt_1$ branch, and the second one to $stmt_2$. To respect the semantics of the language, we decorate the first block method with the result of decompiling *cond*, while we attach $\overline{cond}$ to its sibling. A similar approach is used in virtual invocations, for which we introduce as many block methods in the graph as possible receivers of the call were in the original program. A set of block methods with the same signature *sig* can be retrieved by the function $getBlocks(CFG, sig)$.

User specifications are written using the annotation system introduced in Java 1.5 which, unlike JML specifications, has the very useful characteristic of being preserved in the bytecode. Annotations are carried over to our CFG representation, as can be seen in Fig. 1.

Example 1 We now focus our attention on the two block methods in Fig. 1, which are the result of (de)compiling the `CellPhone.sendSms` method. Input formal parameters r_0, r_1, r_2, r_3 correspond to *this*, *smsPk*, *enc*, and *stm*, respectively. In the case of r_1, the contents of its fields *next* and *sms* are altered by invoking the `stf` and accessed by invoking the `gtf` (abbreviation for `setfield` and `getfield`, respectively) builtin block methods. The output formal parameter r_4 contains the final state of r_1 after those modifications. The value returned by the block methods is contained in r_5. Space reasons prevent us from showing any type information in the CFG in Fig 1. In the case of `Encoder.format`, for example, we say that there are two blocks with the same signature because they are both defined in class `Encoder`, have the same name (`format`) and list of types of formal parameters {`Encoder,String,String`}.

```
resourceAnalysis(CFG, res) {
 CFG← classAnalysis(CFG)
 mt← initialize(CFG)
 dg← dataDependencyAnalysis(CFG, mt)
 for (SCC: SCCs)
    //in reverse topological order
    mt← genSizeEqs(SCC, mt, CFG, dg)
    mt← genResourceUsageEqs(SCC, res, mt, CFG)
 return mt
}
```

```
normalize(Eqs) {
 for (size relation $p \leq e_1$ : Eqs)
   do
      if (expression $s$ appears in $e_1$
           and $s \leq e_2 \in$ Eqs)
        replace ocurrences of $s$ in $e_1$ with $e_2$
   while there is change
 return Eqs
}
```

Fig. 2. Generic resource analysis algorithm and normalization.

4 The resource usage analysis framework

We now describe our framework for inferring upper bounds on the usage that the Java bytecode program makes of a set of resources defined by the application programmer, as described before. The algorithm in Fig 2 takes as input a Control Flow Graph in the format described in the previous section, including the user annotations that assign elementary costs to certain graph elements for a particular resource. The user also indicates the set of resources to be tracked by the analysis. Without loss of generality we assume for conciseness in our presentation a single resource.

A preliminary step in our approach is a class hierarchy analysis [5, 24], aimed at simplifying the CFG and therefore improving overall precision. Then, another analysis is performed over the CFG to extract data dependencies, as described below. The next step is the decomposition of the *CFG* into its strongly-connected components. After these steps, two different analyses are run separately on each strongly connected component: a) the size analysis, which estimates parameter size relationships for each statement and output formal parameters as a function of the input formal parameter sizes (Sec. 4.1); and b) the actual resource analysis, which computes the resource usage of each block method in terms also of the input data sizes (Sec. 4.2). Each phase is dependent on the previous one.

The *data dependency analysis* is a dataflow analysis that yields *position dependency graphs* for the block methods within a strongly connected component. Each graph $G = (V, E)$ represents data dependencies between positions corresponding to statements in the same block method, including its formal parameters. Vertexes in V denote positions, and edges $(s_1, s_2) \in E$ denote that s_2 is dependent on s_1 (s_1 is a *predecessor* of s_2). We will assume a `predec` function that takes a position dependency graph, a statement, and a parameter position and returns its nearest predecessor in the graph. Fig. 3 shows the position dependency graph of the `TrimEncoder.format` block method.

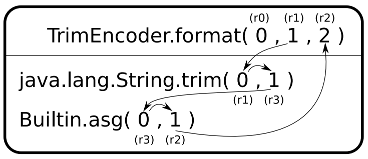

TrimEncoder.format(0 , 1 , 2)
(r0) (r1) (r2)
java.lang.String.trim(0 , 1)
(r1) (r3)
Builtin.asg(0 , 1)
(r3) (r2)

Fig. 3.

4.1 Size analysis

We now show our algorithm for estimating parameter size relations based on the data dependency analysis, inspired by the original ideas of [13, 12]. Also, we provide a concrete algorithm for performing the analysis, rather than the more descriptive presentations of the related work discussed previously. Our goal is to represent input and output size relationships for each statement as a function defined in terms of the formal parameter sizes. Unless otherwise stated, whenever we refer to a parameter we mean its position.

The size of an input is defined in terms of measures. By *measure* we mean a function that, given a data structure, returns a number. Our method is parametric on measures, which can be defined by the user and attached via annotations to parameters or classes. For concreteness, we have defined herein two measures, int for integer variables, and the *longest path-length* [1] ref for reference variables. The longest path-length of a variable is the cardinality of the longest chain of pointers than can be followed from it. More complex measures can be defined to handle other data types such as cyclic structures, arrays, etc. The set of measures will be denoted by $\mathcal{M}$.

The size analysis algorithm is given in pseudo-code in Fig. 4; its main steps are:

1. Assign an upper bound to the size of every parameter position of all statements, including formal parameters, for all the block methods with the same signature (genSigSize).
2. For a given signature, take the set of size inequations returned by (1) and rename each size relation in terms of the sizes of input formal parameters (normalize).
3. Repeat the first step for every signature in the same strongly-connected component (genSizeEqs).
4. Simplify size relationships by resolving mutually recursive functions, and find closed form solutions for the output formal parameters (genSizeEqs).

Intermediate results are cashed in a memo table *mt*, which for every parameter position stores measures, sizes, and resource usage expressions defined in the $\mathcal{L}$ language:

$$\begin{array}{lll}
\langle expr \rangle & ::= & \langle expr \rangle \langle bin\_op \rangle \langle expr \rangle \mid (\sum \mid \prod) \langle expr \rangle \\
& \mid & \langle expr \rangle^{\langle expr \rangle} \mid log_{num} \langle expr \rangle \mid -\langle expr \rangle \\
& \mid & \langle expr \rangle ! \mid \infty \mid \text{num} \\
& \mid & \text{size}([\langle measure \rangle,]\text{arg(r num)}) \\
\langle bin\_op \rangle & ::= & + \mid - \mid \times \mid / \mid \% \\
\langle measure \rangle & ::= & \textbf{int} \mid \textbf{ref} \mid \ldots
\end{array}$$

The size of the parameter at position i in statement *stmt*, under measure m, is referred to as $\texttt{size}(m, stmt, i)$. We consider a parameter position to be *input* if it is bound to some data when the statement is invoked. Otherwise, it is considered an *output parameter position.* In the case of input parameter and output formal parameter positions, an upper bound on that size is returned by getSize (Fig. 4). The upper bound can be a concrete value when there is a constant in the referred position, i.e., when the val function returns a non-infinite value:

Definition 1. *The concrete size value for a parameter position under a particular measure is returned by* $\texttt{val} : \mathcal{M} \times \mathcal{S}tmt \times \mathbb{N} \rightarrow \mathcal{L}$*, which evaluates the* syntactic *content of the actual parameter in that position:*

$$\texttt{val}(m, stmt, i) = \begin{cases} n \text{ if } stmt.\mathsf{apars}_i \text{ is an integer } n \text{ and } m{=}\textbf{int} \\ 0 \text{ if } stmt.\mathsf{apars}_i \text{ is } \texttt{null} \text{ and } m{=}\textbf{ref} \\ \infty \text{ otherwise} \end{cases}$$

If the content of that input parameter position is a variable, the algorithm searches the data dependency graph for its immediate predecessor. Since the intermediate representation is in SSA form, the only possible scenarios are that either there is a unique predecessor

```
genSizeEqs(SCC, mt, CFG, dg) {
  Eqs ← $\emptyset^{|SCC|}$
  for (sig: SCC)
    Eqs[sig] ← genSigSize(sig, mt, SCC, CFG, dg)
  Sols ← recEqsSolver(simplifyEqs(Eqs))
  for (sig: SCC)
    insert(mt, size, sig, Sols[sig])
  return mt
}

genSigSize(sig, mt, SCC, CFG, dg) {
  Eqs ← $\emptyset$
  BMS ← getBlocks(CFG, sig)
  for (bm: BMs)
    Eqs ← Eqs $\cup$ genBlockSize(bm, mt, SCC, dg)
  return normalize(Eqs)
}

genBlockSize(bm, mt, SCC, dg) {
  Eqs ← $\emptyset$
  for (stmt: bm.body)
    I ← stmt input parameter positions
    Eqs ← Eqs $\cup$ genInSize(stmt, I, mt, dg)
    Eqs ← Eqs $\cup$ genOutSize(stmt, mt, SCC)
  K ← bm output formal parameter positions
  Eqs ← Eqs $\cup$ genInSize(bm, K, mt, dg)
  return Eqs
}

genInSize(elem, Pos, mt, dg) {
  Eqs ← $\emptyset$
  for (pos: Pos)
    m ← lookup(mt, measure, elem.sig, pos)
    s ← getSize(m, elem.id, pos, dg)
    Eqs ← Eqs $\cup$ {size(m, elem.id, pos) $\leq$ s}
  return Eqs
}

genOutSize(stmt, mt, SCC) {
  $\{i_1, \ldots, i_l\}$ ← stmt input positions
  sig ← stmt.sig
  $\{m_{i_1}, \ldots, m_{i_l}\}$ ← {lookup(mt, measure, sig, $i_1$), ...,
                    lookup(mt, measure, sig, $i_l$)}
  $\{s_{i_1}, \ldots, s_{i_l}\}$ ← {size($m_{i_1}$, stmt.id, $i_1$), ...,
                    size($m_{i_l}$, stmt.id, $i_l$)
  Eqs ← $\emptyset$
  O ← stmt output parameter positions
  for (o: O)
    $m_o$ ← lookup(mt, measure, sig, o)
    if (sig $\notin$ SCC)
      $Size_{user}$ ← $\mathcal{A}^o_{sig}(s_{i_1}, \ldots, s_{i_l})$
      $Size_{alg'}$ ← max(lookup(mt, size, sig, o))
      $Size_{alg}$ ← $Size_{alg'}(s_{i_1}, \ldots, s_{i_l})$
      $Size_o$ ← min($Size_{user}$, $Size_{alg}$)
    else
      $Size_o$ ← $\mathcal{S}ize^o_{sig}(m_o, s_{i_1}, \ldots, s_{i_l})$
    Eqs ← Eqs $\cup$ {size($m_o$, stmt.id, o) $\leq$ $Size_o$}
  return Eqs
}

getSize(m, id, pos, dg) {
  result ← val(m, id, i)
  if (result $\neq \infty$)
    return result
  else
    if ($\exists$(elem, $pos_p$) $\in$ predec(dg, id, pos))
      $m_p$ ← lookup(mt, measure, elem.sig, $pos_p$)
      if (m = $m_p$)
        return size($m_p$, elem.id, $pos_p$)
  return $\infty$
}
```

Fig. 4. The size analysis algorithm

whose size is assigned to that input parameter position, or there is none, causing the input parameter size to be unbounded (∞).

Consider now an output parameter position within a block method, case covered in genOutSize (Fig. 4). If the output parameter position corresponds to a non-recursive invoke statement, either a size relationship function has already been computed recursively (since the analysis traverses each strongly-connected component in reverse topological order), or it is provided by the user through size annotations. In the first case, the size function of the output parameter position can be retrieved from the memo table by using the lookup operation, taking the maximum in case of several size relationship functions, and then passing the input parameter size relationships to this function to evaluate it. In the second scenario, the size function of the output parameter position is provided by the user through size annotations, denoted by the $\mathcal{A}$ function in the algorithm. In both cases, it will able to return an explicit size relation function.

Example 2 We have already shown in the `CellPhone` example how a class can be annotated. The `Builtin` class includes the assignment method `asg`, annotated as follows:

```
public class Builtin {
  @Size{"size(ret)<=size(o)"}
  public static native Object asg(Object o);
  // ...rest of annotated builtins
}
```

which results in equation $\mathcal{A}^1_{\mathsf{asg}}(\mathtt{ref}, \mathtt{size}(\mathtt{ref}, asg, 0)) \le \mathtt{size}(\mathtt{ref}, asg, 0)$.

If the output parameter position corresponds to a recursive invoke statement, the size relationships between the output and input parameters are built as a symbolic size function. Since the input parameter size relations have already been computed, we can establish each output parameter position size as a function described in terms of the input parameter sizes.

At this point, the algorithm has defined size relations for all parameter positions within a block method. However, those relations are either constants or given in terms of the immediate predecessor in the dependency graph. The algorithm rewrites the equation system such that we obtain an equivalent system in which only formal parameter positions are involved. This process, called *normalization*, is shown in Fig. 2

After normalization, the analysis repeats the same process for all block methods in the same strongly-connected component (SCC). Once every component has been processed, the analysis further simplifies the equations in order to resolve mutually recursive calls among block methods within the same SCC in the simplifyEqs procedure.

In the final step, the analysis submits the simplified system to a recurrence equation solver (recEqsSolver, called from genSizeEqs) in order to obtain approximated upper-bound closed forms. The interesting subject of how the equations are solved is beyond the scope of this paper (see, e.g., [33]). Our implementation does provide a dedicated implementation (an evolution of the solver of the Caslog system [12]) which covers a reasonable set of recurrence equations such as first-order and higher-order linear recurrence equations in one variable with constant and polynomial coefficients, divide and conquer recurrence equations, etc. In addition, the system has interfaces to external solvers such as Purrs [6] or Mathematica.

Example 3 We now illustrate the definitions and algorithm with an example of how the size relations are inferred for the two `CellPhone.sendSms` block methods (Fig. 1), using the ref measure for reference variables. We will refer to the k-th occurrence of a statement *stmt* in a block method as $stmt_k$, and denote `CellPhone.sendSms`, `Encoder.format`, and `Stream.send` by `sendSms`, `format`, and `send` respectively. Finally, we will refer to the size of the input formal parameter position i, corresponding to variable r_i, as s_{r_i}.

The main steps in the process are listed in Fig. 5. The first block of rows contains the most relevant size parameter relationship equations for the recursive block method, while the second block of rows corresponds to the base case. These size parameter relationship equations are constructed by the analysis by first following the algorithm in Fig. 4, and then normalizing them (expressing them in terms of the input formal parameter sizes s_{r_i}). Also, in the first block of rows we observe that the algorithm has returned $6 \times \mathtt{size}(\mathtt{ref}, format, 1)$ as upper bound for the size of the formatted string, $\mathsf{max}(\mathsf{lookup}(mt, \mathtt{size}, format, 2))$. The result is the maximum of the two upper bounds given by the user for the two implementations for `Encoder.format` since `TrimEncoder.format` eliminates any leading and trailing white

Size parameter relationship equations (normalized)	
$\texttt{size}(\texttt{ref}, ne, 0)$	$\leq \texttt{size}(\texttt{ref}, sendSms, 1) \leq s_{r1}$
$\texttt{size}(\texttt{ref}, ne, 1)$	$\leq \texttt{val}(\texttt{ref}, ne, 1) \leq 0$
$\texttt{size}(\texttt{ref}, gtf_1, 0)$	$\leq \texttt{size}(\texttt{ref}, ne, 0) \leq s_{r1}$
$\texttt{size}(\texttt{ref}, gtf_1, 2)$	$\leq \mathcal{A}^2_{gtf}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf_1, 0), \_) \leq s_{r1} - 1$
$\texttt{size}(\texttt{ref}, format, 1)$	$\leq \texttt{size}(\texttt{ref}, gtf_1, 2) \leq s_{r1} - 1$
$\texttt{size}(\texttt{ref}, format, 2)$	$\leq \texttt{max}(\textsf{lookup}(mt, \texttt{size}, format, 2))(\texttt{size}(\texttt{ref}, format, 2))$
	$\leq \texttt{max}(s_{r1}, 6 \times s_{r1})(s_{r_1} - 1)$
	$\leq 6 \times (s_{r1} - 1)$
$\texttt{size}(\texttt{ref}, send, 1)$	$\leq \texttt{size}(\texttt{ref}, format, 2) \leq 6 \times (s_{r1} - 1)$
$\texttt{size}(\texttt{ref}, gtf_2, 0)$	$\leq \texttt{size}(\texttt{ref}, gtf_1, 0) \leq s_{r1}$
$\texttt{size}(\texttt{ref}, gtf_2, 2)$	$\leq \mathcal{A}^2_{gtf}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf_2, 0), \_) \leq s_{r1} - 1$
$\texttt{size}(\texttt{ref}, sendSms, 1)$	$\leq \texttt{size}(\texttt{ref}, gtf_2, 2) \leq s_{r1} - 1$
$\texttt{size}(\texttt{ref}, sendSms, 5)$	$\leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, \_, \texttt{size}(\texttt{ref}, sendSms, 1), \_, \_)$
	$\leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})$
$\texttt{size}(\texttt{ref}, stf_1, 0)$	$\leq \texttt{size}(\texttt{ref}, gtf_2, 0) \leq s_{r1}$
$\texttt{size}(\texttt{ref}, stf_1, 2)$	$\leq \texttt{size}(\texttt{ref}, sendSms, 5) \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})$
$\texttt{size}(\texttt{ref}, stf_1, 3)$	$\leq \mathcal{A}^3_{stf}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf_1, 0), \_, \texttt{size}(\texttt{ref}, stf_1, 2))$
	$\leq s_{r1} + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})$
$\texttt{size}(\texttt{ref}, stf_2, 0)$	$\leq \texttt{size}(\texttt{ref}, stf_1, 3) \leq s_{r1} + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})$
$\texttt{size}(\texttt{ref}, stf_2, 2)$	$\leq \texttt{size}(\texttt{ref}, format, 2) \leq 6 \times (s_{r1} - 1)$
$\texttt{size}(\texttt{ref}, stf_2, 3)$	$\leq \mathcal{A}^3_{stf}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf_2, 0), \_, \texttt{size}(\texttt{ref}, stf_2, 2))$
	$\leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})$
$\texttt{size}(\texttt{ref}, asg, 0)$	$\leq \texttt{size}(\texttt{ref}, stf_2, 3)$
	$\leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})$
$\texttt{size}(\texttt{ref}, asg, 1)$	$\leq \mathcal{A}^1_{asg}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0))$
	$\leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})$
$\texttt{size}(\texttt{ref}, eq, 0)$	$\leq \texttt{size}(\texttt{ref}, sendSms, 1) \leq s_{r1}$
$\texttt{size}(\texttt{ref}, eq, 1)$	$\leq \texttt{val}(\texttt{ref}, eq, 1) \leq 0$
$\texttt{size}(\texttt{ref}, asg, 0)$	$\leq \texttt{val}(\texttt{ref}, asg, 0) \leq 0$
$\texttt{size}(\texttt{ref}, asg, 1)$	$\leq \mathcal{A}^1_{asg}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \leq 0$
Output parameter size functions for builtins (provided through annotations)	
$\mathcal{A}^2_{\texttt{gtf}}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf, 0), \_)$	$\leq \texttt{size}(\texttt{ref}, gtf, 0) - 1$
$\mathcal{A}^1_{\texttt{asg}}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0))$	$\leq \texttt{size}(\texttt{ref}, asg, 0)$
$\mathcal{A}^3_{\texttt{stf}}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf, 0), \_, \texttt{size}(\texttt{ref}, stf, 2))$	$\leq \texttt{size}(\texttt{ref}, stf, 0) + \texttt{size}(\texttt{ref}, stf, 2)$
Simplified size equations and closed form solution	
$\mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1}, s_{r2}, s_{r3})$	$\leq \begin{cases} 0 & \text{if } s_{r1} = 0 \\ 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) & \text{if } s_{r1} > 0 \end{cases}$
$\mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1}, s_{r2}, s_{r3})$	$\leq 3.5 \times s^2_{r1} - 2.5 \times s_{r1}$

Fig. 5. Size equations example

spaces (thus the output is at most as bigger as the input), whereas `UnicodeEncoder.format` converts any special character into its Unicode equivalent (thus the output is at most six times the size of the input), a safe upper bound for the output parameter position size is given by the second annotation.

```
genResourceUsageEqs(SCC, res, mt, CFG) {
  Eqs← $\emptyset^{|SCC|}$
  for (sig:SCC)
    Eqs[sig]←genSigRU(sig, res, mt, SCC, CFG)
  Sols←recEqsSolver(simplifyEqs(Eqs))
  for (sig:SCC)
    insert(mt, cost, max(Sols[sig]))
  return mt
}

genSigRU(sig, res, mt, SCC, CFG) {
  Eqs← $\emptyset$
  BMs←getBlocks(CFG, sig)
  for (bm:BMs)
    body←bm.body
    $Cost_{body}$ ← 0
    for (stmt:body)
      $Cost_{stmt}$ ←genStmtRU(stmt, res, mt, SCC)
      $Cost_{body}$ ←$Cost_{body}$ + $Cost_{stmt}$
    $Cost_{bm}$ ←genBlockRU(bm, res, mt)
    Eqs←Eqs ∪ {$Cost_{bm}$ ≤$Cost_{body}$}
}

genStmtRU(stmt, res, mt, SCC) {
  $\{i_1, \ldots, i_k\}$ ← stmt input parameter positions
  $\{s_{i_1}, \ldots, s_{i_k}\}$ ←
     {max(lookup(mt, size, stmt.sig, $i_1$)), ...,
      max(lookup(mt, size, stmt.sig, $i_k$))}
  if (stmt.sig∉ SCC)
    $Cost_{user}$ ← $\mathcal{A}_{stmt.sig}$(res, $s_{i_1}$, ..., $s_{i_k}$)
    $Cost_{alg'}$ ←lookup(mt, cost, res, stmt.sig)
    $Cost_{alg}$ ←$Cost_{alg'}$($s_{i_1}$, ..., $s_{i_k}$)
    return min($Cost_{alg}$, $Cost_{user}$)
  else return Cost(stmt.sig, res, $s_{i_1}$, ..., $s_{i_k}$)
}

genBlockRU(bm, res, mt) {
  $\{i_1, \ldots, i_l\}$ ← bm input formal parameter positions
  $\{s_{i_1}, \ldots, s_{i_l}\}$ ←
     {lookup(mt, size, bm.id, $i_1$), ...,
      lookup(mt, size, bm.id, $i_l$)}
  return Cost(bm.id, res, $s_{i_1}$, ..., $s_{i_l}$)
}
```

Fig. 6. The resource usage analysis algorithm

In the particular case of builtins and methods for which we do not have the code, size relationships are not computed but rather taken from the user `@Size` annotations. These functions are illustrated in the third block of rows. Finally, in the fourth block of rows we show the recurrence equations built for the output parameter sizes in the block method and in the final row the closed form solution obtained.

4.2 Resource usage analysis

The core of our framework is the resource usage analysis, whose pseudo code is shown in Fig 6. It takes a strongly-connected component of the CFG, including the set of annotations which provide the application programmer-provided resources and cost functions, and calculates an resource usage function which is an upper bound on the usage made by the program of those resources. The algorithm manipulates the same memo table described in Sec. 4.1 in order to avoid recomputations and access the size relationships already inferred.

The algorithm is structured in a very similar way to the size analysis (which also allows us to draw from it to keep the explanation within space limits): for each element of the strongly-connected component the algorithm will construct an equation for each block method that shares the same signature representing the resource usage of that block. To do this, the algorithm will visit each invoke statement. There are three possible scenarios, covered by the genStmtRU function. If the signatures of caller and callee(s) belong to the same strongly-connected component, we are analyzing a recursive invoke statement. Then, we add to the body resource usage a symbolic resource usage function, in an analogous fashion to the case of output parameters in recursive invocations during the size analysis.

The other scenarios occur when the invoke statement is non-recursive. Either a resource usage function $Cost_{alg}$ for the callee has been previously computed, or there is a user annotation $Cost_{usr}$ that matches the given signature, or both. In the latter case, the minimum between these two functions is chosen (i.e., the most precise safe upper bound assigned by the analysis to the resource usage of the non-recursive invoke statement).

Example 4 The call (sixth statement) in the upper-most `CellPhone.sendSms` block method matches the signature of the block method itself and thus it is recursive. The first four parameter positions are of input type. The upper-bound expression returned by genStmtRU is $\underset{sendSms}{Cost}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3})$. Note that the input size relationships were already normalized during the size analysis. Now consider the invocation of `Stream.send`. The resource usage expression for the statement is defined by the function $\mathcal{A}_{send}(\$, \_, 6\times(s_{r1}-1))$ since the input parameter at position one is at most six times the size of the second input formal parameter, as calculated by the size analysis in Fig. 5. Note also that there is a resource annotation `@Cost({"cents","2*size(r1)"})` attached to the block method describing the behavior of $\mathcal{A}_{send}$ and yielding the expression $Cost_{user} = 12\times(s_{r1}-1)$. On the other hand, the absence of any callee code to analyze –the original method is native– results in $Cost_{alg} = \infty$. Then, the upper bound obtained by the analysis for the statement is $\min(Cost_{alg}, Cost_{user}) = Cost_{user}$.

At this point, the analysis has built a resource usage function (denoted by $Cost_{body}$) that reflects the resource usage of the statements within the block. Finally, it yields a resource usage equation of the form $Cost_{block} \leq Cost_{body}$ where $Cost_{block}$ is again a symbolic resource usage function built by replacing each input formal parameter position with its size relations in that block method. These resource usage equations are simplified by calling simplifyEqs and, finally, they are solved calling recEqsSolver, both already defined in Sec. 4.1. This process yields an (in general, approximate, but always safe) closed form upper bound on the resource usage of the block methods in each strongly-connected component. Note that given a signature the analysis constructs a closed form solution for every block method that shares that signature. These solutions approximate the resource usage consumed in or provided by each block method. In order to compute the total resource usage of the signature the analysis returns the maximum of these solutions yielding a safe global upper bound.

Example 5 The resource usage equations generated by our algorithm for the two `sendSms` block methods and the "\$" resource (monetary cost of sending the SMSs) are listed in Fig. 7. The computation is partially based on the size relations in Fig. 5. The resource usage of each block method is calculated by building an equation such that the left part is a symbolic function constructed by replacing each parameter position with its size (i.e., $\underset{sendSms}{Cost}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3})$ and $\underset{sendSms}{Cost}(\$, s_{r0}, 0, s_{r2}, s_{r3})$), and the rest of the equation consists of adding the resource usage of the invoke statements in the block method. These are calculated by computing the minimum between the resource usage function inferred by the analysis and the function provided by the user. The equations corresponding to the recursive and non-recursive block methods are in the first and second row, respectively. They can be simplified (third row) and expressed in closed form (fourth row), obtaining a final upper bound for the charge incurred by sending the list of text messages of $6 \times s_{r1}^2 - 6 \times s_{r1}$.

5 Experimental results

We have completed an implementation of our framework, and tested it for a representative set of benchmarks and resources. Our experimental results are summarized in Table 1. Column

Resource usage equations

$$\begin{aligned}
\underset{sendSms}{Cost}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq\ & \mathsf{min}(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, ne)}^{\infty},\ \overbrace{\mathcal{A}_{ne}(\$, s_{r1}, \_)}^{\texttt{@Cost("cents","0")=0}})\\
& +\mathsf{min}(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, gtf)}^{\infty},\ \overbrace{\mathcal{A}_{gtf}(\$, s_{r1}, \_)}^{\texttt{@Cost("cents","0")=0}}\ )\\
& +\mathsf{min}(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, format)(\_, s_{r1}-1)}^{0}, \overbrace{\mathcal{A}_{format}(\$, \_, s_{r1}-1)}^{\infty})\\
& +\mathsf{min}(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, send)}^{\infty},\ \overbrace{\mathcal{A}_{send}(\$, \_, 6 \times (s_{r1}-1))}^{\texttt{@Cost("cents","2*size(r1)")=}12\times(s_{r1}-1)})\\
& +\mathsf{min}(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, gtf)}^{\infty},\ \overbrace{\mathcal{A}_{gtf}(\$, s_{r1}, \_)}^{\texttt{@Cost("cents","0")=0}}\ ) + \underset{sendSms}{Cost}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3})\\
& +\mathsf{min}(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, stf)}^{\infty},\ \overbrace{\mathcal{A}_{stf}(\$, s_{r1}, \_, \_)}^{\texttt{@Cost("cents","0")=0}}\ )\\
& +\mathsf{min}(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, stf)}^{\infty},\ \overbrace{\mathcal{A}_{stf}(\$, s_{r1}, \_, \_)}^{\texttt{@Cost("cents","0")=0}}\ )\\
& +\mathsf{min}(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, asg)}^{\infty},\ \overbrace{\mathcal{A}_{asg}(\$, \_)}^{\texttt{@Cost("cents","0")=0}})\\
& \leq 12 \times (s_{r1}-1) + \underset{sendSms}{Cost}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3})
\end{aligned}$$

$$\begin{aligned}
\underset{sendSms}{Cost}(\$, s_{r0}, 0, s_{r2}, s_{r3}) \leq\ & \mathsf{min}(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, eq)}^{\infty}\ ,\ \overbrace{\mathcal{A}_{eq}(\$, 0, \_)}^{\texttt{@Cost("cents","0")=0}})\\
& + \mathsf{min}(\underbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, asg)}_{\infty},\ \underbrace{\mathcal{A}_{asg}(\$, 0)}_{\texttt{@Cost("cents","0")=0}}) \quad \leq 0
\end{aligned}$$

Simplified resource usage equations and closed form solution

$$\underset{sendSms}{Cost}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq \begin{cases} 0 & \text{if } s_{r1} = 0 \\ 12 * s_{r1} - 12 + \underset{sendSms}{Cost}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3}) & \text{if } s_{r1} > 0 \end{cases}$$

$$\underset{sendSms}{Cost}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq 6 \times s_{r1}^2 - 6 \times s_{r1}$$

Fig. 7. Resource equations example

Program provides the name of the main class to be analyzed. Column **Resource(s)** shows the resource(s) defined and tracked. Column t_s shows the time (in milliseconds) required by the size analysis to construct the size relations (including the data dependency analysis and class hierarchy analysis) and obtain the closed form. Column t_r lists the time taken to build the resource usage expressions for all method blocks and obtain their closed form solutions. t provides the total times for the whole analysis process. Finally, column **Resource Usage Func.** provides the upper bound functions inferred for the resource usage. For space reasons, we only show the most important (asymptotic) component of these functions, but the analysis yields concrete functions with constants.

Regarding the benchmarks we have covered a reasonable set of data-structures used in object-oriented programming and also standard Java libraries used in real applications. We have also covered an ample set of application-dependent resources which we believe can be relevant in those applications. In particular, not only have we represented high-level resources such as cost of SMS, bytes received (including a coarse measure of bandwidth, as

Program	Resource(s)	t_s	t_r	t	Resource Usage Func.	
BST	Heap usage	250	22	367	$O(2^n)$	$n \equiv$ tree depth
CellPhone	SMS monetary cost	271	17	386	$O(n^2)$	$n \equiv$ packets length
Client	Bytes received and bandwidth required	391	38	527	$O(n)$ $O(1)$	$n \equiv$ stream length —
Dhrystone	Energy consumption	602	47	759	$O(n)$	$n \equiv$ int value
Divbytwo	Stack usage	142	13	219	$O(log_2(n))$	$n \equiv$ int value
Files	Files left open and Data stored	508	53	649	$O(n)$ $O(n \times m)$	$n \equiv$ number of files $m \equiv$ stream length
Join	DB accesses	334	19	460	$O(n \times m)$	$n, m \equiv$ records in tables
Screen	Screen width	388	38	536	$O(n)$	$n \equiv$ stream length

Table 1. Times of different phases of the resource analysis and resource usage functions.

a ratio of data per program step), and files left open, but also other low-level (i.e., bytecode level) resources such as stack usage or energy consumption. The resource usage functions obtained can be used for several purposes. In program *Files* (a fragment characteristic of operating system kernel code) we kept track of the number of file descriptors left open. The data inferred for this resource can be clearly useful, e.g., for debugging: the resource usage function inferred in this case ($O(n)$) denotes that the programmer did not close $O(n)$ file descriptors previously opened. In program *Join* (a database transaction which carries out accesses to different tables) we decided to measure the number of accesses to such external tables. This information can be used, e.g., for resource-oriented specialization in order to perform optimized checkpoints in transactional systems. The rest of the benchmarks include other definitions of resources which are also typically useful for verifying application-specific properties: *BST* (a generic binary search tree, used in [2] where a heap space analysis for Java bytecode is presented), *CellPhone* (extended version of program in Figure 1), *Client* (a socket-based client application), *Dhrystone* (a modified version of a program from [20] where a general framework is defined for estimating the energy consumption of embedded JVM applications; the complete table with the energy consumption costs that we used can be found there), *DivByTwo* (a simple arithmetic operation), and *Screen* (a MIDP application for a cellphone, where the analysis is used to make sure that message lines do not exceed the phone screen width). The benchmarks also cover a good range of complexity functions ($O(1), O(log(n), O(n), O(n^2) \ldots, O(2^n), \ldots$) and different types of structural recursion such as simple, indirect, and mutual. The code for these benchmarks and a demonstrator are available at `http://www.cs.unm.edu/~jorge/RUA`.

6 Conclusions

We have presented a fully-automated analysis for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. Our analysis derives a vector of functions, one for each defined resource. Each of these functions returns, for each given set of input data sizes, an upper bound on the usage that the whole program (and each individual method) make of the corresponding resource. Important novel aspects of our approach are the fact that it allows the application programmer to define the resources to be tracked by writing simple resource descriptions via source-level annotations, as well as the fact that we have provided a concrete analysis algorithm and report on an implementation. The current results show that the proposed analysis can obtain non-trivial bounds on a wide range of interesting resources in reasonable time. Another important aspect of our work, because of its impact on the scalability, precision, and automation of the analysis, is that our approach allows using the annotations also for a number of other purposes such as stating the resource usage of external methods, which is instrumental in allowing modular composition and thus scalability. In addition, our annotations allow stating the resource usage of any method for which the automatic analysis infers a value that is not accurate enough to prevent inaccuracies in the automatic inference from propagating. Annotations are also used by the size and resource usage analysis to express their output. Finally, the annotation language can also be used to state specifications related to resource usage, which can then be proved or disproved based on the results of analysis following, e.g., the scheme of [19] thus finding bugs or verifying (the resource usage of) the program.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.
2. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.
3. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.
4. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, LNCS 3362, pages 1–27. Springer-Verlag, 2005.
5. David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *Proc. of OOPSLA'96, SIGPLAN Notices*, 31(10):324–341, October 1996.
6. R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo. Purrs: The Parma University's Recurrence Relation Solver. `http://www.cs.unipr.it/purrs`.
7. I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *5th IEEE Int'l. Symp. on Object-oriented Real-time Distributed Computing*, Apr. 2002.
8. R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.
9. Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symposium on Programming (ESOP)*, number 3444 in LNCS, pages 311–325. Springer-Verlag, 2005.
10. S.J. Craig and M. Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.
11. K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.
12. S. K. Debray and N. W. Lin. Cost analysis of logic programs. *TOPLAS*, 15(5), 1993.
13. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. PLDI'90*, pages 174–188. ACM, June 1990.
14. S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *ILPS'97*. MIT Press, 1997.
15. J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proc. of DDECS*. IEEE Computer Society, 2006.

16. G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 2002.
17. B. Grobauer. Cost recurrences for DML programs. In *Int'l. Conf. on Functional Programming*, pages 253–264, 2001.
18. M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *PPDP*. ACM Press, 2005.
19. M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
20. Sébastien Lafond and Johan Lilius. Energy consumption analysis for two embedded java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007.
21. D. Le Metayer. ACE: An Automatic Complexity Evaluator. *TOPLAS*, 10(2), 1988.
22. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.
23. P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21:715–734, 1996.
24. M. Méndez-Lojo and M. Hermenegildo. Precise Set Sharing Analysis for Java-style Programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.
25. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.
26. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP*, LNCS, 2007.
27. G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of PPDP'06*, pages 261–271. ACM Press, 2006.
28. M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.
29. D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.
30. Lothar Thiele and Reinhard Wilhelm. Design for time-predictability. In *Perspectives Workshop: Design of Systems with Predictable Behaviour*, 2004.
31. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.
32. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.
33. H. S. Wilf. *Algorithms and Complexity.* A.K. Peters Ltd, 2002.
34. R. Wilhelm. Timing analysis and timing predictability. In *Proc. FMCO*, LNCS. Springer-Verlag, 2004.