2008 IEEE International Conference on Research, Innovation and Vision for the Future in Computing & Communication Technologies
University of Science – Vietnam National University, Ho Chi Minh City, July 13-17, 2008

# Computing heap space cost of Java Card applets

Tuan-Hung Pham, Anh-Hoang Truong, Ninh-Thuan Truong
College of Technology
Vietnam National University
144 Xuan Thuy, Cau Giay, Hanoi
Emails: s0420192@coltech.vnu.vn, {hoangta, thuantn}@vnu.edu.vn

***Abstract*—We introduce an approach to find upper bounds of heap space for Java Card applets. Our approach analyzes and transforms bytecodes of Java Card applets into equivalent programs in a language that already has a type system for finding the sharp upper bounds of resource use. We then point out a linear-time algorithm to compute the maximum heap units that may be allocated during the lifetime of Java Card applets. We also have implemented a prototype tool and tested it on several examples and the results are good.**

## I. Introduction

Java Card is a technology used on smart cards or other small embedded devices whose memory and processing constraints are highly limited. An applet written in Java Card technology can be downloaded into a smart card and run completely inside the card. During runtime, the applet consumes the limited resources of the card. If the applet is not written well, unintentionally or intentionally, memory overflow may occur, and this may result in losing data or even destruction of the card. Therefore, it is crucial to know the maximum resources, in particular the heap space, which an applet may use before it is allowed to run on a smart card.

Heap space analysis is an active research area that has not only theoretical depth but also practical applications. Several analytical techniques were developed to address the problem in different ways. Many of these approaches analyze the heap memory use at source code level [1], [2], [3], [4]. Sometimes doing that at source code is not feasible since the source code is not always available so we also need to analyze the compiled bytecodes. To do this, some works [5], [6] propose the ideas of managing memory resources by analyzing the memory affected for each bytecode instruction that appears in compiled code, but they are not actually suitable for installing on cards.

This work introduces an approach to compute a maximum heap space used by an applet on smart cards. We develop a linear-time algorithm that takes all bytecodes of a method as input and returns a heap bound cost as output. The main steps of the computation process are as follows:

- First, we divide the list of bytecodes into blocks based on the branching instructions of Java Card. Then we build a so-called control flow graph (CFG) of the blocks. The nodes of the CFG are the bytecode blocks. For edges, if after the last instruction of a block the execution can continue to the first instruction of another block, then we connect an edge from the former block to the latter. Albert et al. [7] used this idea for finding memory cost of Java bytecode. Since Java Card instructions set is a subset of Java instructions set, we can reuse the technique and we will not restate how to construct the CFG here.
- Second, the CFG is heap-preserving transformed into a rooted, directed tree (RDT) that is equivalent to a syntax tree [8] of the abstract component language in the approach by Truong and Bezem [9]. Since the abstract component language already has a type system that can infer maximum heap space, we base on their type inference to compute the heap space cost of Java Card applets.
- Finally, our algorithm takes the RDT as input data and returns its maximum heap space. If loops and arrays whose number of iterations and lengths, respectively, are provided as constants in programs, our algorithm can deal with them. For simplicity, we also assume that there is no method invocation in the input applet.

The rest of this paper is organized as follows. Section II briefly introduces Java Card bytecode instruction set and a simplified version of the abstract component language. Section III presents our contribution, an approach to find upper bounds of heap space for Java Card applets. In Section IV, we show our experimental results on several sample applications, including ones in Sun's Java Card Development Kit. Related works are discussed in Section V. We conclude and give future directions in Section VI.

## II. Background

### A. Java Card virtual machine (JCVM) instruction set

Java Card bytecodes are the form of instructions that JCVM understands and executes. Each bytecode instruction consists of one opcode and zero or more operands. The opcode represents the action JCVM needs to perform, while the operands act as arguments of the action.

Among all Java Card bytecode instructions, there are only three instructions that can increase heap cost. They are `new`, `newarray` and `anewarray` for creating a new instance of an object, an array of a primitive type and an array of object references, respectively. All other instructions do not change heap space when they are executed.

This work was supported by the College of Technology, Vietnam National University, Hanoi, under the project CN.07.08.

ISSN 1242-5125 ENST S (Paris) (France 1983-9999)

For example, suppose that we have a Java Card method that returns the minimum value between two integers $a$ and $b$ as follows:

```
int min (int a, int b){
 if (a < b) return a;
 else return b;
}
```

JCVM compiles this source code to the following bytecode stream:

```
int min(int, int);
  Code:
   0:   iload_1
   1:   iload_2
   2:   if_icmpge 7
   5:   iload_1
   6:   ireturn
   7:   iload_2
   8:   ireturn
```

As we can see, every instruction begins with an offset (0, 1, 2, 5, 6, 7, and 8) followed by a mnemonic of an opcode and operand values (if exists).

For various reasons, JCVM does not support several features such as threads, cloning, and finalization of Java VM. The instruction set of JCVM is thus only a subset of that of normal Java VM. As a result, the first step of our approach in Section III-A is based on the result of Albert et al. [7].

### *B. A simplified component language*

Truong and Bezem [9] developed an abstract component language and a type system that can find the maximum number of instances for each component of a well-typed program. We found that if we can map bytecode programs to equivalent ones in this language, we can easily compute the maximum heap space used by Java Card applets (as the authors hinted in the end of Section 2.2). This section introduces a simplified version of the abstract component language of their approach [9]. We removed deallocation primitive, parallel composition and scope operators that Java Card bytecode does not use.

Component programs, declarations and expressions of our simplified component language (SCL) are defined in Fig. 1. We use extended Backus-Naur Form with infix | for choice and star for Kleene closure (zero or more iterations).

Component names, ranged over by $x, y, z$, are collected in a set $\mathbb{C}$. Component expressions, ranged over by $A, .., E$, can be empty expressions – used for startup, or primitive $\mathtt{new}\, x$ for creating a new instance of component $x$, or they can be assembled by two composition operators: choice, denoted by $+$, and sequencing denoted by juxtaposition.

| | | | |
|---|---|---|---|
| $Prog$ | $::=$ | $Decl^*; E$ | Program |
| $Decl$ | $::=$ | $x \prec E$ | Declaration |
| $E$ | $::=$ | | Expression |
| | | $\epsilon$ | Empty |
| | $\mid$ | $\mathtt{new}\, x$ | Instantiation |
| | $\mid$ | $(E + E)$ | Choice |
| | $\mid$ | $E\ E$ | Sequencing |

Fig. 1. Syntax of a simplified component language.

A declaration $x \prec E$ states that component $x$ deploys the component expression $E$. A primitive component is the one that does not depend on any other components, $x \prec \epsilon$, so it can be used to represent some specific resource such as a unit of memory. A component program is a list of component declarations followed by a main expression, which will be the startup expression when the program is executed.

## III. Our approach

### *A. Converting a CFG into an equivalent RDT*

As mentioned in Section I, Truong and Bezem [9] introduced an abstract component language and a type system that can count the maximum number of instances for each component. Therefore, if bytecodes of a Java Card applet can be transformed into the simplified version of the language - SCL, the maximum heap cost of the applet is easy to compute from their type inference algorithm. Since the SCL can be represented by a syntax tree (also called parse tree [8]), we only need to point out a way to convert the CFG [7] into an equivalent RDT of the SCL's syntax tree. One of the crucial requirements of the transformation is that it must preserve heap memory allocation.

Our CFG and RDT have two important properties. First, since every bytecode instruction must be reachable from the entry instruction of its method (otherwise bytecode verification of JVCM will fail), there always exists at least one path between two arbitrary nodes in these graphs if we ignore all directions of edges. Second, our main goal is the heap space consumption of Java Card applets, so every node may have only two states: $\mathtt{new}\, x$ if it increases the heap, $\epsilon$ in otherwise.

Now we will show how to transform a RDT into an equivalent syntax tree of SCL. Transformation rules are shown in Fig. 2 where:

- $E$ denotes an intermediate expression;
- $E(i)$ is the expression of all nodes reachable from node $i$;
- $e$ denotes an expression of a node;
- $v(i)$ is the value of node $i$, including $\mathtt{new}\, x$ and $\epsilon$;
- $\times$ and $+$ denote sequencing and choice operators, respectively.

We build a corresponding syntax tree from a RDT by traversing the RDT from its root to its leaf nodes. First the syntax tree has only one node $E(root)$. At node $i$ in RDT we count its outdegree and apply the corresponding rule in Fig. 2. That is we replace $E(i)$ by the right hand side of $\Rightarrow$ in the rule. Since the RDT has a finite number of nodes, this process will stop after the last leaf node of the RDT is processed and we obtain the corresponding full syntax tree of the RDT.

Fig. 3 shows a RDT and its equivalent syntax tree after applying the algorithm. As it is not difficult, we do not give all details of the process.

We have shown that bytecodes of an applet has an equivalent CFG and a RDT can be transformed into an equivalent syntax

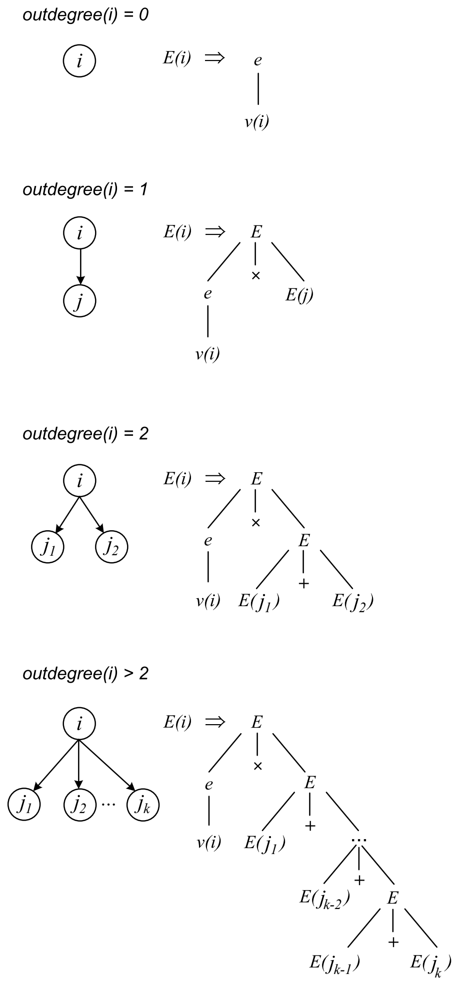


Fig. 2. Rules to transform a RDT into a syntax tree.

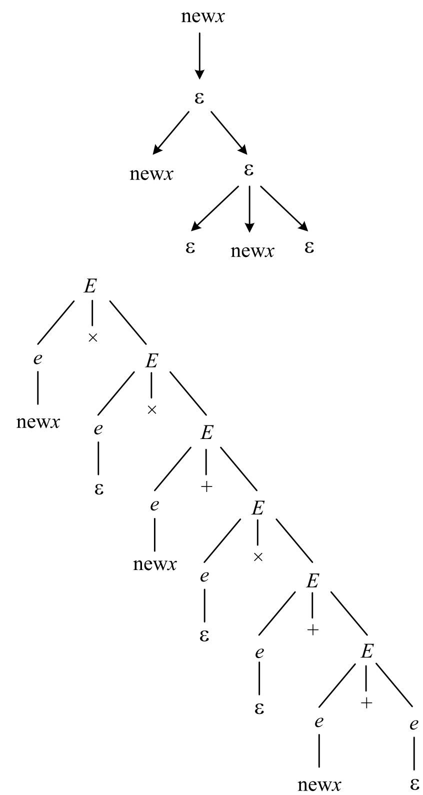

Fig. 3. An example of a RDT and its corresponding syntax tree.

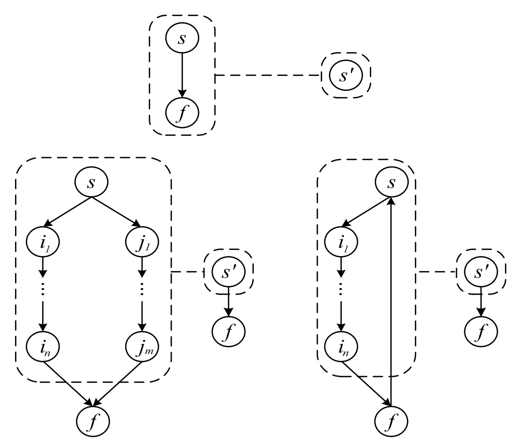

Fig. 4. Transformation (Shrink) rules from CFG into RDT.

tree of SCL. To complete the picture, we need to transform the CFG into an equivalent RDT in the sense that their heap allocations are the same. After that, we can find the maximum heap cost of Java Card applets by applying the type inference algorithm in the approach by Truong and Bezem [9].

To match the two graphs, it is necessary to recognize the differences between them first. There are two main differences.

- The CFG can have cycles while the RDT cannot. Cycles are caused by **`for`**, **`while`**, and **`do-while`** statements which produce `goto` bytecode instructions jumping to smaller offsets.
- The CFG can have join nodes while the RDT cannot. This is caused by `if` and `goto` instructions jumping to higher offsets.

From these above observations, we use several rules that do not affect heap allocations to shrink the CFG so that the number of cycles and join nodes are gradually reduced. This process is repeated until we receive a RDT. By repeating these rules, the CFG will be transformed into a RDT without affecting the maximum heap cost of the CFG.

Concretely, Fig. 4 shows three rules to transform the CFG into a RDT: If we have a directed edge $(s, f)$ that $outdegree(f) = 0$ and $indegree(f) = 1$, we remove node $f$ and add its heap cost to node $s$. If there are multiple distinct paths from node $s$ to node $f$ in which every node $i_1, .., i_n, j_1, .., j_m$ has one in both indegree and outdegree, we remove all these paths and add the maximum heap cost of the paths into node $s$. Similarly, if there is a cycle formed by $s, i_1, .., i_n, f, s$ and every node $i_1, .., i_n$ has one in both indegree and outdegree, we remove $i_1, ..., i_n$ and increase heap cost of node $s$ by the result of multiplication of total heap consumption of $i_1, ..., i_n$ and the cycle's repeat number provided as a constant in program's source code.

### B. An algorithm for finding heap bound of Java Card applets

We have shown that it is feasible to compute heap space of a bytecode program. However, we still need an algorithm that has very small memory footprint and runs fast enough to be able to run on cards. We have following observations:

- After obtaining a RDT, the heap bound calculation of each node is based on its preceding ones. That means if there is a path from node $i$ to node $j$ then calculation of node $i$ must be carried out before that of node $j$. In other words, we need to know one topological order of heap costs of vertices.
- Regarding the data structures used in our algorithm, the most space consuming one is cost of storing the CFG. We need three one-dimensional arrays to do this including $next[]$, $goto\_jump[]$, and $if\_jump[]$ for storing the next sequential node, the set of `goto` jump destinations and the target of `if` jump of each node, respectively.
- Instead of considering a node of CFG formed by a block of adjacent instructions as proposed by Albert et al. [7], an instruction acts as a node in our algorithm to avoid complexity.
- Since a cycle is only formed when an instruction performs `goto` to a smaller offset, we can detect cycles easily by comparing the offset of two bytecodes.

From the above notes, we propose a new algorithm for inferring heap bound in Algorithm 1. Before that, we present some notations.

- $M[i]$: maximum heap consumption when the node $i$ of the method is executed.
- $visited[i]$: **`true`** if node $i$ has already visited, **`false`** in otherwise.
- $indegree[i]$: the indegree of node $i$.
- $next[i]$: the instruction follows node $i$ and can run immediately after node $i$ runs.
- $if\_jump[i]$: the target of conditional branch of node $i$.
- $goto\_jump[i]$: the set of destinations where node $i$ performs `goto` to.
- $HeapConsumption(i)$: heap consumption used by the corresponding instruction of node $i$.
- $FirstInstruction()$: the first instruction in bytecode stream.
- $LastInstruction()$: the set of `return` instructions that can exit a method.

Since depth-first search takes $O(|V|+|E|)$ time and it takes $O(1)$ time to update heap bound of an instruction, the time complexity of the algorithm is $O(|V|+|E|)$.

We briefly explain our algorithm by running the sample program in Fig. 5. Its bytecode stream and CFG are represented in Fig. 6 and Fig. 7, respectively. The algorithm starts from instruction 0 whose heap bound is initially set to zero. There is no heap bound changed when going to the node 1, 3, 5 and 6. At node 6, a choice between the next node 9 and the conditional branch 70 appears, we go to node 9 first since the priority of process of our DFS method is `goto`, `next` and `if`, respectively. When we reach node 11, there is a $goto-jump$ list which contains four elements 36, 47, 54, 64.

**Algorithm 1** Heap bounds algorithm for Java Card applets

*Input:* The bytecode stream of a method
*Output:* The heap bound value

**HeapBound**():
DFS($FirstInstruction()$);
**return** $max(M[i] \mid i \in LastInstruction())$

**DFS**($i$):
**if** $indegree[i] = 0$ **then**
  $visited[i] \leftarrow$ **true**
  **if** $goto\_jump[i] \neq \emptyset$ **then**
    **for all** $j \in goto\_jump[i]$ **do**
      UpdateHeapBound($i$, $j$)
      DFS($j$)
    **end for**
  **else**
    **if** $\exists next[i]$ **then**
      UpdateHeapBound($i$, $next[i]$);
      DFS($next[i]$);
    **end if**
    **if** $\exists if\_jump[i]$ **then**
      UpdateHeapBound($i$, $if\_jump[i]$);
      DFS($if\_jump[i]$);
    **end if**
  **end if**
**end if**

**UpdateHeapBound**($i$, $j$):
**if** $visited[j] =$ **true and** $offset[i] > offset[j]$ **then**
  $M[j] \leftarrow M[j] + cycle's\ repeat\ number \times (M[i] - M[j])$ // A cycle is detected
  $temp \leftarrow j$ // Move forward two nodes to escape cycle
  **repeat**
    $temp \leftarrow next[temp]$
    $M[temp] \leftarrow M[j]$
  **until** $\exists if\_jump[temp]$
  $M[if\_jump[temp]] \leftarrow M[j]$
**else**
  $M[j] \leftarrow max(M[j], M[i] + HeapConsumption(j))$
  $indegree[j] \leftarrow indegree[j] - 1$;
**end if**

At node 36, we have an instruction that allocates memory for `AClass`. It makes the heap bound of this node changed from 0 to $Size$(`AClass`). The heap bound is preserved through node 39, 40, 43, and 44. Similarly, the paths from node 11 to nodes 47, 48, 50, 51 and to nodes 54, 56, 59, 61 set the heap bound of node 51 and node 61 to $Size$(`int[5]`) and $Size$(`AClass[10]`), respectively.

```
public void sample_method(int x) {
  AClass aclass;
  int[] prim_arr;
  AClass[] class_arr;

  for (int i = 0; i < 3; i++) {
   switch (x) {
   case 0:
     aclass = new AClass();
     break;
   case 1:
     prim_arr = new int[5];
     break;
   case 2:
     class_arr = new AClass[10];
     break;
   }
  }
}
```

Fig. 5. An example method named `sample_method`.

```
public void sample_method(int);
Code:
 0:   iconst_0
 1:   istore 5
 3:   iload 5
 5:   iconst_3
 6:   if_icmpge 70
 9:   iload 1
 11:  tableswitch{ //0 to 2
      0: 36;
      1: 47;
      2: 54;
      default: 64}
 36:  new #5; //class AClass
 39:  dup
 40:  invokespecial #6;
         //Method AClass."<init>":()V
 43:  astore_2
 44:  goto 64
 47:  iconst_5
 48:  newarray int
 50:  astore_3
 51:  goto 64
 54:  bipush 10
 56:  anewarray #5; //class AClass
 59:  astore 4
 61:  goto 64
 64:  iinc 5, 1
 67:  goto 3
 70:  return
```

Fig. 6. The bytecode stream of `sample_method`.

Considering node 64, its heap bound is the maximum of heap bounds of its ingoing nodes and equal to $max$(0, $Size$(`AClass`), $Size$(`int[5]`), $Size$(`AClass[10]`)) = $max$($Size$(`int[5]`), $Size$(`AClass[10]`)). This maximum heap cost is still kept for node 67.

A cycle is detected when node 67 performs a `goto` instruction back to node 3. The cycle's repeat number is the number of iterations that the for-loop of `sample_method` in Fig. 5 has. The heap bound of node 3 is hence updated from 0 to 3 $\times$ $max$($Size$(`int[5]`), $Size$(`AClass[10]`)). Next, it escapes the cycle by finding the nearest node that has a conditional jump to outside. The heap bound of node 5 and 6 are thus changed from 0 to 3 $\times$ $max$($Size$(`int[5]`), $Size$(`AClass[10]`)).

From node 6, an `if` jump is performed to go to the last instruction of this method and update its heap bound which indicates the maximum heap used in the entire method.

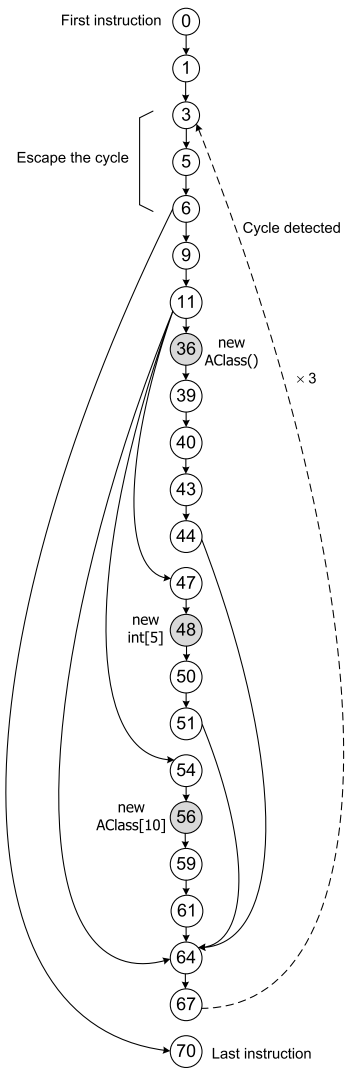


Fig. 7. Corresponding CFG of `sample_method`.

Suppose that $Size$(`AClass[10]`) = 30 (bytes), we can compute the heap bound of this method as follows:

Heap Bound Value = Heap bound of the last instruction = 3 $\times$ $max$($Size$(`int[5]`), $Size$(`AClass[10]`)) = 3 $\times$ $max$(4 $\times$ 5, 30) = 90 (bytes).

## IV. Experimental results

To illustrate our approach, we have implemented a prototype tool that compute the heap bounds of Java Card applets using

TABLE I
EXPERIMENTAL RESULTS.

| Sample | Size | NInstrs | NEdges | Time |
|---|---|---|---|---|
| NullApp | 916 | 21 | 18 | 15 |
| Wallet | 3277 | 64 | 68 | 15 |
| RMIDemo | 3067 | 111 | 107 | 15 |
| ServiceDemo | 2755 | 114 | 111 | 15 |
| JavaLoyalty | 2312 | 125 | 128 | 15 |
| ChannelsDemo | 7227 | 191 | 190 | 15 |
| Transit | 7765 | 269 | 276 | 15 |
| Biometry | 5419 | 271 | 259 | 15 |
| Photocard | 5383 | 330 | 340 | 15 |
| Utilitydemo | 10580 | 405 | 414 | 15 |
| SecureRMIDemo | 7411 | 417 | 424 | 30 |
| JavaPurse | 15489 | 465 | 479 | 30 |
| JavaPurseCrypto | 16713 | 524 | 538 | 30 |
| SigMsgRec | 4606 | 654 | 656 | 30 |
| Total | 92920 | 3961 | 4008 | 270 |

our proposed algorithm. The input of this tool is a bytecode stream of a class file and the output is the heap bound of all methods in the file. We tested our tool on sample applets presented in Java Card Development Kit 2.2.2. The obtained results are shown in Table I.

In the table, the first column is the name of the sample that we use. Each sample consists of several class files and the total size (in bytes) of these classes is represented in the **Size**. The columns **NInstrs** and **NEdges** are the numbers of bytecode instructions and CFG's directed edges of each sample, respectively. The last column is the execution time in milliseconds. By doing a manual calculation, we found that all heap bounds of these applets our tool returned are accurate.

We observe that the number of instructions and that of directed edges of are not substantially difference and a directed edge is formed by either two adjacent instructions or jumping (`if` and `goto`) from an instruction to its targets. Since `if` and `goto` rarely appear in bytecode streams, the values shown in the **NInstrs** and **NEdges** columns are almost the same. Since the number of vertices $|V|$ is almost the same as the number of egdes $|E|$, the complexity of our algorithm is linear $O(|V|)$. The experiments were performed on an Intel P4 2.4 GHz with 1GB of RAM.

## V. Related works

The most relevant works to ours seem to be [1], [2], [4], [10], [5], [7]. Unnikrishnan et al. [1] propose a analysis model for inferring maximum size of live heap space of programs written in garbage-collected languages. Similarly, Hofmann and Jost [4] points out a method to count the memory used for object allocation and deallocation in Java-like languages by extending the ideas of the static prediction in [2]. Since these techniques are performed at source-level, it is difficult to apply them to the context of Java Card environment where source code is not available.

Giambiagi and Schneider [10] analyze memory consumption by an algorithm that finds all potential loops and (mutually) recursive methods of Java Card applets. In contrast to our linear-time algorithm, the their complexity is still high. It thus needs more optimizations to be actually fit into smart cards.

The most closely related work to our approach is [5] that relies on heap space cost relations generated at compile-time to infer heap bound of an input Java bytecode program. Both this work and ours use the same method to construct a CFG from bytecode stream described in [7], but for different purposes. Unlike ours, the technique takes longer time to run, especially for the abstract-interpretation based size analysis. Therefore, it is not suitable for Java Card applets.

## VI. Conclusion

We have presented an approach to compute an upper bound of heap consumption of Java Card applets. Our approach constructs a resource-preserving map from a stream of Java Card bytecodes to the language proposed in the approach by Truong and Bezem [9]. We have implemented a prototype tool that takes a compiled Java Card applet as input and prints out heap memory consumption of its methods. The tool has been tested on several sample applications and the results are good.

In the future, we plan to take into account method calls and loops whose numbers of iterations are not only constants but also method arguments. By doing that, the heap bound is a function of method arguments and thus we can find heap bound for a larger set of Java Card applets.

## VII. Acknowledgement

This work was supported by the College of Technology, Vietnam National University, Hanoi, under the project CN.07.08. We also thank Van-Hung Dang, Viet-Ha Nguyen and anonymous reviewers for their comments on the earlier version of the paper.

## References

[1] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu, "Optimized live heap bound analysis," in *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*. London, UK: Springer-Verlag, 2003, pp. 70–85.

[2] M. Hofmann and S. Jost, "Static prediction of heap space usage for first-order functional programs," in *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2003, pp. 185–197.

[3] W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard, "Memory usage verification for OO programs," in *SAS*, 2005, pp. 70–86.

[4] M. Hofmann and S. Jost, "Type-based amortised heap-space analysis (for an object-oriented language)," in *Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, ser. LNCS, P. Sestoft, Ed., vol. 3924. Springer, 2006, pp. 22–37.

[5] E. Albert, S. Genaim, and M. Gomez-Zamalloa, "Heap space analysis for java bytecode," in *ISMM '07: Proceedings of the 6th international symposium on Memory management*. New York, NY, USA: ACM, 2007, pp. 105–116.

[6] W. G. Schneider, "Memory consumption analysis of java smart cards," in *XXXI Latin American Informatics Conference (CLEI 2005)*, 2005.

[7] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Cost analysis of java bytecode," in *ESOP*, ser. Lecture Notes in Computer Science, R. D. Nicola, Ed., vol. 4421. Springer, 2007, pp. 157–172.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[9] H. Truong and M. Bezem, "Finding resource bounds in the presence of explicit deallocation," in *Proceedings ICTAC*, ser. Lecture Notes in Computer Science, D. V. Hung and M. Wirsing, Eds., vol. 3722. Springer, 2005, pp. 227–241.

[10] P. Giambiagi and G. Schneider, "Memory consumption analysis of java smart cards," in *Proceedings of CLEI'05*, Cali, Colombia, October 2005.