An improved type system for counting logs of transactional multi-threaded programs

Vu Xuan Tung[1] and Truong Anh Hoang[2]

[1] Japan Advanced Institute of Science and Technology
tungvx@jaist.ac.jp
[2] University of Engineering and Technology, Vietnam National University, Hanoi
hoangta@vnu.edu.vn

Abstract. This paper presents an improved type system to estimate an upper bound for the resource consumption of programs with nested and multi-threaded transactions. This work improves our previous type system to which in comparison the new one is more precise. In addition, the soundness of the system is sketched by the help of rewriting techniques.

1 Introduction

Software Transactional Memory [1] is an alternative to locked-based synchronization for shared memory concurrency. One of various transactional models supporting advanced features such as nested and multi-threaded transactions is described in [2]. In this model, a nested transaction is the one inside which some other transaction starts. The former and the latter are called the parent transaction and the child one respectively. A child transaction must commit (terminate) before their parent. Furthermore, a transaction is multi-threaded when there are a number of threads which run inside that transaction. In order to independently manipulate shared variables, a thread spawned inside a transaction will make a copy of the memory (variables) being used by the parent thread executing the transaction. As a result, for assuring the memory consistency, when the parent thread commits a transaction, all its child threads must join the commit. This point of commit is referred as a *joint commit point*. In a quintessential implementation, each transaction maintains a local copy of memory called log to record memory accesses during its execution. Because each thread may execute a number of nested transactions, the number of logs coexisting at the same time may be many. In particular, a child thread will further store a copy of its parent's logs so that the it can be executed independently with the parents until a joint commit point. At joint commit points, their own logs and the copies are consulted to check for consistency and potentially perform a roll-back when conflicts are detected. A major complication for the static analysis is that the memory allocation (cloned logs) and the synchronized memory deallocation (joint commits) are implicit. As a consequence, estimating the resources used by a transactional programs is challenging.

In our previous works [3, 4], we proposed type systems to statically estimate the memory consumption in terms of the maximum number of logs co-existing

at the same time. In this work, we improved our system in [3] and sketch the proof with the help of rewriting techniques.

A motivating example An example is described in Figure 1. Whilst the statement onacid starts a new transaction, commit closes the corresponding transaction. The spawn(e) statement starts a new thread to execute e. The program

```
1  onacid; // thread 0
2      onacid;
3          spawn(e1; commit; commit); // thread 1
4          onacid;
5              e2;
6              spawn(e3; commit; commit; commit) // thread 2
7          commit;
8          e4;
9      commit;
10     e5;
11 commit;
```

Fig. 1. A nested, multi-threaded program.

is graphically illustrated in Figure 2. The commands onacid and commit are denoted by [and], resp. Figure 2 further illustrates that the parallel threads must commit a transaction at the same time. The vertical alignments of commits illustrate these synchronizations. Suppose that e_1 opens and closes one transaction,

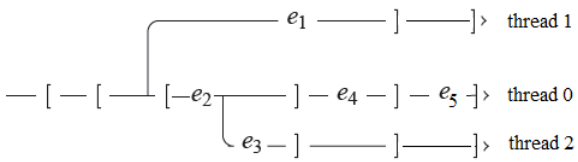

Fig. 2. Illustration of the running example

e_2 four, e_3 two, e_4 four and e_5 five. The maximum resource consumption occurs after spawning e_2. At that time, thread 1 contributes 3 transactions (2 from the main thread, and 1 of e_1), thread 2 contributes 6 transactions (3 from the main thread, and 3 of e_3), and the main thread (thread 0) contributes 3 transactions. And the total will be $3 + 6 + 3 = 12$ transactions.

As mentioned, child threads join with their parent via joint commits at a time which is an implicit synchronization point. The difficulty for the analysis is

that it must capture those implicit synchronization points at compile time which are not explicitly represented by the syntax. Furthermore, the analysis needs to contain enough information in order to analyze the resource consumption *compositionally*. The utilized language is a variant of Featherweight Java extended by transactional constructs known as Transactional Featherweight Java [2].

Related work Estimating resource usage has been studied in various settings. Hughes and Pareto [5] introduce a strict, first order functional language with a type system such that well typed programs run within the space specified by the programmer. The paper [6] uses inference system to describe a memory management for programs that perform dynamic memory allocation and de-allocation. Hofmann and Jost [7] compute the linear bounds on heap space for a first-order functional language. In terms of imperative and object-oriented languages, Wei-Ngan Chin et al. [8] verify memory usages for object-oriented programs. Programmers are required to annotate the memory usage and size relations for methods as well as explicit de-allocation. In [9] the authors statically compute upper bounds of resource consumption of a method using a non-linear function of method's parameters. The bounds are not precise and their work is not type-based. Braberman et al. [10] calculate non-linear symbolic approximation of memory bounds for Java programs involving both data structures and loops. In [11] the authors propose type systems for component languages with parallel composition but the threads run independently. In [12], Albert et al. compute the heap consumption of a program as a function of its data size. [13] proposes a fast algorithm to statically find the upper bounds of heap memory for a class of JavaCard programs.

Our analysis not only takes care of multi-threading —many of the cited works are restricted to sequential languages — but also of the complex and implicit synchronization (by joint commits) structure entailed by the transactional model.

Content outlines The rest of the paper is structured as follows. Section 2 introduces syntax and operational semantics of the calculus. Section 3 presents the improved type system. The soundness of the analysis is sketched in Section 4. We conclude and give the future work in Section 5.

2 The language TFJ

Transactional Featherweight Java (TFJ) is an object calculus featuring threads and imperative constructs needed to model transactions, supporting a quite expressive transactional concurrency model.

Syntax The syntax of TFJ is shown in Table 1 which is similar to the the syntax of Featherweight Java except for the first and the last line. These two lines make the language multi-threaded and transactional. The explanation of the syntax was given in [4].

$P ::= \mathbf{0} \mid P \parallel P \mid p(e)$	processes
$L ::= \mathsf{class}\ C\{\boldsymbol{f}, \boldsymbol{M}\}$	class definition
$M ::= m(\boldsymbol{x})\{e\}$	methods
$v ::= r \mid x \mid \mathsf{null}$	values
$e ::= v \mid v.f \mid v.f := v \mid \mathsf{if}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e$	
$\mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid v.m(\boldsymbol{v})$	expressions
$\mid \mathsf{new}\ C() \mid \mathsf{spawn}(e) \mid \mathsf{onacid} \mid \mathsf{commit}$	

Table 1. TFJ syntax

2.1 Semantics

The semantics of TFJ are given by two-level set of operational rules, the local and the global semantics .

Local semantics The local semantics deal with the evaluation of *one* single *thread* and local transitions are of the form $E, e \rightarrow E', e'$. E and E' here are *local environments*, while e and e' are expressions being executed by the thread. We start by defining a local environment.

Definition 1 (Local environment). *A local environment E is a set of the form H; V where H is a finite sequence $l_1{:}log_1$; ...; $l_k{:}log_k$, i.e., sequence of pairs of transaction labels l_i and the corresponding* log *log_i, V is the set of variables. We define the size of E is the number of pairs $l{:}log$, denoted $|E|$.*

The sequence of transaction names and logs is used to represent the nesting structure. The number $|E|$ further specifies the nesting depth of the thread. The local level concerns a single thread and consists of rules for the commands for reading, writing, method calls and for creating new objects (Table 2 of [4]).

Global semantics At the global level, the reduction is of the form: $\Gamma, P \Rightarrow \Gamma', P'$ or $\Gamma, P \Rightarrow error$, where Γ is the global environment and P is a process.

Definition 2 (Global environment). *A global environment Γ is a finite set of maps, written as $p_1{:}E_1$; ...; $p_k{:}E_k$, from thread names p_i to local environments E_i. We write $|\Gamma|$ for denoting the size of Γ, and $|\Gamma| = \sum_{i=0}^{k} |E_i|$.*

Definition 3 (Resource consumption). *Given a program in the state of the global environment Γ, the actual resource consumption of the program at that moment is $|\Gamma|$.*

The global steps make use of functions which access and change the global environment: $reflect(p_i, E_i', \Gamma)$, $spawn(p, p', \Gamma)$, $start(l, p_i, \Gamma)$, $intranse(\Gamma, l)$, $commit(\boldsymbol{p}, \boldsymbol{E}, \Gamma)$ which are further clarified in [4]. Five reduction rules of the *global* semantics are given in Table 3 of [4] .

Definition 4. *Given two global environments Γ_1, Γ_2. The function $common(\Gamma_1, \Gamma_2)$ returns the number of common transactions of Γ_1 and Γ_2.*

3 Type system

Types in our system are computed from what we call sequence of *tagged* numbers, which abstracts the transactional behaviors of terms.

3.1 Types

To represent local transactional behavior of a term, we use the set of four tags or signs $\{+, -, \#, \neg\}$ to abstractly represent, respectively, the opening transactions, closing transactions, (local) maximum and joint commit behavior of the term. These tags are paired with positive natural numbers. More specifically, our sequences of tagged numbers are sequences over the set

$$\mathbf{N^T} = \{^{+}1\,, ^{-}1\,\} \cup \{^{\#}n \mid n \in \mathbf{N}^+\} \cup \{^{\neg}n \mid n \in \mathbf{N}^+\}$$

where $\mathbf{N}^+$ is the set of all positive natural numbers.

Definition 5. $S = s_1 s_2 s_3 \dots s_k$ *is a* sequence of tagged numbers *iff.* $s_i \in \mathbf{N^T}$ *for all* $i \in \{1, \dots, k\}$.

Let ϵ be the empty sequence and $|s_i|$ stand for the natural number of s_i. The latter notion is abused to represent the "zero" behavior of ϵ: $|\epsilon| = 0$.

3.2 Typing rules

Types T is defined by the following syntax where S is a sequence of tagged numbers, TT is the concatenation of two types. Operations $T \otimes T$, T^{ρ}, $T \odot T$, and $T \parallel^k T$ appear to type different kinds of expressions.

$$T = S \mid TT \mid T \otimes T \mid T^{\rho} \mid T \odot T \mid T \parallel^k T$$

Local level On the local level, the typing judgment T of an expression e has the form: $e : T$. The local derivation rules for expressions by our type system are shown in Table 2.

Global level In order to prove the soundness of the type system, we need to type expressions during the running time. Table 3 presents the derivation rules for a process which can be either a thread or a composition of parallel processes.

3.3 A Rewriting System for Types

We define a rewriting system $(\Sigma, \mathbf{R})$ for simplifying types, which consists of a set $\Sigma = \{concatenation, \otimes, \rho, \odot, ^{+}1\,, ^{-}1\,\} \cup \{\parallel^k \mid k \in \mathbf{N}\} \cup \{^{\#}n \mid n \in \mathbf{N}^+\}$ $\cup \{^{\neg}n \mid n \in \mathbf{N}^+\}$ of function symbols (*concatenation* means the usual concatenation function over strings), and a set $\mathbf{R}$ of rewriting rules which will be given later. Let $s^{\#}$ be either an empty sequence or a single element $^{\#}n$: $s^{\#}_k \in \{\epsilon\} \cup \{^{\#}n \mid$

$$\frac{}{\mathsf{onacid} : {}^{+}1}\ \text{T-ONACID} \qquad \frac{}{\mathsf{commit} : {}^{-}1}\ \text{T-COMMIT}$$

$$\frac{e_1 : T_1 \quad e_2 : T_2}{\mathsf{let}\ x = e_1 \mathsf{in}\ e_2 : T_1 T_2}\ \text{T-LET} \qquad \frac{e : T}{\mathsf{spawn}(e) : (T \otimes \epsilon)^{\rho}}\ \text{T-SPAWN}$$

$$\frac{e_1 : T_1 \quad e_2 : T_2}{\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_1 \odot T_2}\ \text{T-COND}_1 \qquad \frac{mtype(C,m) : E \to T}{v.m(\boldsymbol{v}) : T}\ \text{T-CALL}$$

$$\frac{}{v : \epsilon}\ \text{T-VAL} \qquad \frac{}{v.f : \epsilon}\ \text{T-ACCESS}$$

$$\frac{}{v.f := v' : \epsilon}\ \text{T-UPDATE} \qquad \frac{}{\mathsf{new}\ C() : \epsilon}\ \text{T-NEW}$$

Table 2. Type system (local level)

$$\frac{e : T}{p(e) : T}\ \text{T-THREAD}$$

$$\frac{\Gamma_1, P_1 \quad P_1 : T_1 \quad \Gamma_2, P_2 \quad P_2 : T_2 \quad \Gamma = \Gamma_1, \Gamma_2 \quad common(\Gamma_1, \Gamma_2) = k}{P_1 \| P_2 : T_1 \|^k T_2}\ \text{T-PAR}$$

Table 3. Type system (global level)

$n \in \mathbf{N}^+\}$, $s^{\mathbf{n}}$ (bold **n** intuitively means **negation**) be a single element of either ${}^{-}1$ or ${}^{\neg}n$: $s^{\mathbf{n}}_k \in \{{}^{-}1\} \cup \{{}^{\neg}n \mid n \in \mathbf{N}^+\}$, and $s^{\mathbf{c}}$ (bold **c** means a **commit** point) either ${}^{-}1$, ${}^{\neg}n$ or $T_1 \|^0 \cdots \|^0 T_k$: $s^{\mathbf{c}} \in \{{}^{-}1\} \cup \{{}^{\neg}n \mid n \in \mathbf{N}^+\} \cup \{T_1 \|^0 \cdots \|^0 T_k \mid k > 2\}$. These notation can be used possibly with a subscript. We group the rewriting rules by relevance of function symbols.

1. Because ${}^{\#}n$ describes n nested transactions,

$$ {}^{+}1\, s^{\#}\ {}^{-}1 \ \rightarrow\ {}^{\#}(\left|s^{\#}\right| + 1)$$

2. Since ${}^{\#}m\,{}^{\#}n$ represents that m nested transactions and n ones are sequential, the expression with larger value will be selected when rewriting.

$$ {}^{\#}m\,{}^{\#}n \ \rightarrow\ {}^{\#}max(m, n)$$

3. Intuitively, because the ρ notion is used to mark the expression will be run in parallel with the main thread, it should be idempotent.

$$(T^{\rho})^{\rho} \rightarrow T^{\rho}$$

   Because the role of ρ is to mark the commits to be join with the parent thread, it can be removed when no commit exists.

$$({}^{\#}n\,)^{\rho} \rightarrow\ {}^{\#}n$$

Expressions of the form $T_1^\rho T_2$ represent that T_2 is the remaining part of the parent thread after spawning the thread corresponding to T_1. Because T_2 contains commits to join with those of T_1, we concatenate T_1 and T_2 to let commits ready for joining.

$$T_1^\rho T_2 \to (T_1 T_2)^\rho$$

4. The operation $\otimes$ signals the need for joining commits in types. The following rules "push" commits into ready for joining. The first rule of this group generally make the system compositional.

$$(T_1 \otimes T_2)T_3 \to T_1 \otimes (T_2 T_3)$$
$$T_1 \otimes s^\# T_2^\rho \to T_1 \otimes s^\# T_2$$
$$T_1^\rho \otimes T_2 \to T_1 \otimes T_2$$

5. The rules below do actual joining commits between threads where from left to right of types, we find the first *negation* components and combine them. This way is very nature to the semantics of joint commits in TFJ.

$$\begin{aligned}
\epsilon \otimes \epsilon &\to \epsilon \\
{}^\#m \otimes {}^\#n &\to {}^\#(m+n) \\
{}^\#m \otimes \epsilon &\to {}^\#m \\
\epsilon \otimes {}^\#m &\to {}^\#m \\
s_1^\# s_1^{\mathbf{n}} T_1 \otimes s_2^\# s_2^{\mathbf{n}} T_2 &\to (s_1^\# \otimes s_2^\#)\,{}^\neg(|s_1^{\mathbf{n}}| + |s_2^{\mathbf{n}}|)\,(T_1 \otimes T_2)
\end{aligned}$$

6. Because $\odot$ is used to type alternatives of if_then_else, between the corresponding $s^\#$ components, we take one with the larger value.

$$\begin{aligned}
{}^\#m \odot {}^\#n &\to {}^\#max(m,n) \\
{}^\#m \odot \emptyset &\to {}^\#m \\
\epsilon \odot {}^\#n &\to {}^\#n \\
s_1^\#\,{}^+1\,T_1 \odot s_2^\#\,{}^+1\,T_2 &\to (s_1^\# \odot s_2^\#)\,{}^+1\,(T_1 \odot T_2) \\
s_1^\#\,{}^-1\,T_1 \odot s_2^\#\,{}^-1\,T_2 &\to (s_1^\# \odot s_2^\#)\,{}^-1\,(T_1 \odot T_2) \\
s_1^\#\,{}^\neg n_1\,T_1 \odot s_2^\#\,{}^\neg n_2\,T_2 &\to (s_1^\# \odot s_2^\#)\,{}^\neg\max(n_1,n_2)\,(T_1 \odot T_2) \\
T_1^\rho \odot T_2^\rho &\to (T_1 \odot T_2)^\rho
\end{aligned}$$

7. The $\neg$ components represent the number of threads inside the latest opened transaction. Such transaction thus when closing will contribute as much as its number of threads to the local maximal.

$${}^+1\,s^\#\,{}^\neg n \to {}^\#(\left|s^\#\right| + n)$$

The following rule is similar to the above but it takes care of which nested transactions are copied. It is the rule that make our system more precise than the one in our previous work [3].

$$^{+}1\ s_1^{\#}\ (s_2^{\#}\ ^{\neg}n\ S)^{\rho} \rightarrow\ ^{\#}(\left|s_1^{\#}\right| + 1)\ (^{\#}(\left|s_2^{\#}\right| + n)\ \ S)^{\rho}$$

Applying the rewriting rules into above example we will get $^{\#}12$ instead of $^{\#}13$ as in [3].

8. Those below rules are used to take care of joint commits between threads in running time.

$$\begin{aligned}
\epsilon\ \|^0\ T \quad &\rightarrow\ T \\
T\ \|^0\ \epsilon \quad &\rightarrow\ T \\
^{\#}m\ \|^0\ ^{\#}n \quad &\rightarrow\ ^{\#}(m+n) \\
T_1 s_1^{\mathbf{c}} s_1^{\#}\ \|^k\ T_2 s_2^{\mathbf{c}} s_2^{\#} \ &\rightarrow\ (T_1\ \|^{k-1}\ T_2)(s_1^{\mathbf{c}}\ \|^0\ s_2^{\mathbf{c}})(s_1^{\#}\ \|^0\ s_2^{\#}) \\
&\quad \text{if } k > 0 \text{ and } T_1 s_1^{\mathbf{c}} s_1^{\#}, T_2 s_2^{\mathbf{c}} s_2^{\#} \text{ are normal forms}
\end{aligned}$$

9. The following rules for global level are similar to those in group 7.

$$^{+}1\ s^{\#}(^{\#}n\ \|^0\ T)\ \rightarrow\ (^{\#}(\left|s^{\#}\right| + n)\ \|^0\ \ ^{+}1\ T) \text{ if } ^{\#}n\ \|^0\ T \text{ is a normal form}$$

$$^{+}1\ \ s^{\#}(S\ \|^0\ T)\ \rightarrow\ s^{\#}(^{+}1\ S\ \|^0\ T)\ \text{ if } S\ \|^0\ T \text{ is a normal form and } S \neq\ ^{\#}n$$

Proposition 1 *The rewriting system* $(\Sigma, \mathbf{R})$ *is terminating and confluent.*

Definition 6. *A program is* well-typed *if it has type* T *and* $T \rightarrow^{!}\ s^{\#}$.

Theorem 1. *If* $ST \rightarrow^{!}\ s^{\#}$, *then* $\left|s^{\#}\right| \geq |S|$ *where* $S =\ ^{+}1 \cdots\ ^{+}1$

Proof. By induction on the rewriting sequence of ST.

4 Soundness

This section establishes the soundness of our type system that the static estimation over-approximates the actual potential resource consumption of a program. First, we define a function for estimating maximum resource consumption during executing a program.

Definition 7. *If* Γ, P *is a running state of program and* $P : T$, *then the maximum resource consumption during executing* P *is estimated as*

$$max(\Gamma, P) = \begin{cases} \left|s^{\#}\right| & \text{if } ST \rightarrow^{!} s^{\#} \\ error & \text{otherwise.} \end{cases}$$

where $S =\ ^{+}1 \cdots^{+}1$ *and* $|S| = |\Gamma|$

Lemma 1 (Subject reduction (local)). *For any local transition* $E, e \rightarrow E', e'$*, if* $S(T_1 \parallel^k T) \rightarrow^! s^\#$ *then* $S(T_1 \parallel^k T') \rightarrow^! (s')^\#$ *and* $\left|s^\#\right| \geq \left|(s')^\#\right|$ *where* T *and* T' *are types of* e *and* e' *respectively, and* $S = {}^+1 \cdots {}^+1$ *,* T_1 *is any type.*

Proof. By considering all transitions of Table **??**.

Lemma 2 (Subject reduction (global)). *For any transition* $\Gamma, P \rightarrow \Gamma', P'$ *if* $max(\Gamma, P) = n$ *then* $max(\Gamma', P') = n'$ *and* $n \geq n'$*.*

Proof. By considering all transitions of Table **??** with the help of Lemma 1.

Lemma 3. *Given a well-typed program* P_0 *and* T *is its type, or* $T \rightarrow^! s^\#$*. For any state* Γ, P *of the program, we have:* $\max(\Gamma, P) \leq \left|s^\#\right|$

Proof. By inductions on transitions of the semantics.

- Initial state: the assertion trivially holds because $\max(\emptyset, P) = \left|s^\#\right| \leq \left|s^\#\right|$.
- For any transition $\Gamma, P \rightarrow \Gamma', P'$, assume that $\max(\Gamma, P) \leq \left|s^\#\right|$, then $\max(\Gamma', P') \leq \max(\Gamma, P)$ (Lemma 2). As a result $\max(\Gamma', P') \leq \left|s^\#\right|$.

We ends the section by the theorem about the soundness of our system.

Theorem 2. *Given a well-typed program* P_0 *and* T *is its type, or* $T \rightarrow^! s^\#$*. Then the resource consumption of the program during running cannot exceed* $\left|s^\#\right|$*.*

Proof. Consider any state of the program Γ, P. By Lemma 3, we have $\max(\Gamma, P) \leq \left|s^\#\right|$ In addition, by definition of $\max(\Gamma, P)$ and Theorem 1, the following inequality is inferred: $|\Gamma| \leq \left|s^\#\right|$

5 Conclusion

We improved a type system for statically estimating the resource upper bound for a transactional model supporting nested and multi-threaded transactions with synchronization commits. In addition, the soundness of the system is further formally proved in the paper. For future work, we plan to implement a type inference tool based on the proposed type system and extend the language with exceptions handling. We also want to make our analysis more fine-grained to infer the number of copies of each variables or the actual memory in bytes.

References

[1] N. Shavit, D. Touitou, Software transactional memory, in: Symposium on Principles of Distributed Computing, 1995, pp. 204–213.
[2] S. Jagannathan, J. Vitek, A. Welc, A. Hosking, A transactional object calculus, Sci. Comput. Program. 57 (2) (2005) 164–186.

[3] X.-T. Vu, M. T. Tran, A.-H. Truong, M. Steffen, A type system for finding upper resource bounds of multi-threaded programs with nested transactions, SoICT '12, ACM, New York, NY, USA, 2012, pp. 21–30.
[4] T. M. T. Tran, M. Steffen, H. Truong, Estimating Resource Bounds for Software Transactions, Technical report 414, University of Oslo, Dept. of Informatics (Dec. 2011).
[5] J. Hughes, L. Pareto, Recursion and dynamic data-structures in bounded space: Towards embedded ML programming, SIGPLAN Notices 34 (9).
[6] M. Tofte, J.-P. Talpin, Region-based memory management, Information and Computation 132 (2).
[7] M. Hofmann, S. Jost, Static prediction of heap space usage for first-order functional programs, in: Proceedings of POPL '03, ACM, 2003.
[8] W.-N. Chin, H. H. Nguyen, S. Qin, M. C. Rinard, Memory usage verification for OO programs, in: Proceedings of SAS '05, Vol. 3672 of Lecture Notes in Computer Science, Springer-Verlag, 2005.
[9] V. Braberman, D. Garbervetsky, S. Yovine, A static analysis for synthesizing parametric specifications of dynamic memory consumption, Journal of Object Technology 5 (5).
[10] D. Aspinall, R. Atkey, K. MacKenzie, D. Sannella, Symbolic and analytic techniques for resource analysis of Java bytecode, in: M. Wirsing, M. Hofmann, A. Rauschmayer (Eds.), TGC'10, no. 6084 in Lecture Notes in Computer Science, Springer-Verlag, 2010.
[11] M. Bezem, D. Hovland, H. Truong, A type system for counting instances of software components, Technical report, University of Bergen, Norway (2007).
[12] E. Albert, S. Genaim, M. Gomez-Zamalloa, Heap space analysis for Java bytecode, in: ISMM '07, ACM, New York, NY, USA, 2007.
[13] T.-H. Pham, A.-H. Truong, N.-T. Truong, W.-N. Chin, A fast algorithm to compute heap memory bounds of Java Card applets, in: SEFM'08, 2008.