# Type-based heap and stack space analysis in Java

Emmanuel Hainry, Romain Péchoux


## ▶ To cite this version:

Emmanuel Hainry, Romain Péchoux. Type-based heap and stack space analysis in Java. 2013. hal-00773141v2

## HAL Id: hal-00773141
## https://hal.inria.fr/hal-00773141v2

Submitted on 13 Jan 2013 (v2), last revised 27 Nov 2013 (v4)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

hal-00773215, version 1 - 12 Jan 2013

# Type-based heap and stack space analysis in Java

Emmanuel Hainry
Loria and Université de Lorraine
Emmanuel.Hainry@loria.fr

Romain Péchoux
Loria and Université de Lorraine
Romain.Pechoux@loria.fr

***Abstract*—A type system is introduced for a strict but expressive subset of Java in order to infer resource upper bounds on both the heap-space and the stack-space requirements of typed programs. This type system is inspired by previous works on Implicit Computational Complexity, using tiering and non-interference techniques. The presented methodology has several advantages. First, it provides explicit polynomial upper bounds to the programmer, hence avoiding OutOfMemory and StackOverFlow errors. Second, type checking is decidable in linear time. Last, it has a good expressivity since it analyzes most object oriented features like overload, inheritance, override.**

***Index Terms*—OOP, Type system, Heap and stack upper bounds, Secure Information Flow.**

## I. Introduction

In the last decade, the development of embedded systems and mobile computing has led to a renewal of interest in predicting program resource consumption. This kind of problematic is highly challenging for popular object oriented programming languages which come equipped with environments for applications running on mobile and other embedded devices (e.g. Dalvik, Java Platform Micro Edition (Java ME), Java Card and Oracle Java ME Embedded).

The current paper tackles such an issue by introducing a type system for a compile-time analysis of both heap and stack space requirements of Java-like programs thus avoiding OutOfMemory and StackOverFlow errors, respectively. The set of analyzed programs is a strict but expressive subset of Java, named core Java, and features like recurrence, while loops, inheritance, override, overload are handled by the presented analysis. Core Java will be presented in a theoretically oriented manner in order to highlight the theoretical soundness of our results. It can be seen as a language strictly more expressive than Featherweight Java [18] enriched with features like variable updates and while loops.

The type system combines ideas coming from tiering discipline, used for complexity analysis of function algebra [3], [21], together with ideas coming from non-interference, used for secure information flow analysis [27]. It is inspired by two previous works:

- the seminal paper [22], initiating imperative programs type-based complexity analysis using secure information flow, which provides a characterization of polynomial time computable functions,
- and the paper [12], extending previous analysis to C processes with a fork/wait mechanism, which provides a characterization of polynomial space computable functions,

but this work differs on the following points:

- first, it is an extension to object-oriented paradigm (although imperative feature can be dealt with). In particular, it allows to study the complexity of recursive and non-recursive method calls whereas previous works where restricted to while loops,
- second, it studies program intensional properties (like heap and stack) whereas previous papers were focusing on the extensional part (characterizing function spaces). Consequently, it is closer to a programmer's expectations in term of analysis,
- third, it provides explicit big $O$ polynomial upper bounds while the two aforementioned studies were only certifying algorithms to compute a function belonging to some fixed complexity class.

The main intuition behind the type system is as follows. The heap is represented in term of a directed graph structure where nodes are object addresses and arrows relate an object address to its attribute addresses. The type system splits variables in two universes, tier $\mathbf{0}$ universe and tier $\mathbf{1}$ universe. Whereas tier $\mathbf{1}$ variables are pointers to nodes of the initial heap, tier $\mathbf{0}$ variables may point to newly created addresses. The information may flow from tier $\mathbf{1}$ to tier $\mathbf{0}$, that is a tier $\mathbf{0}$ variable may depend on tier $\mathbf{1}$ variables. However our type system precludes flows from $\mathbf{0}$ to $\mathbf{1}$. Indeed once a variable has stored a newly created instance, it can only be of tier $\mathbf{0}$. Naively, tier $\mathbf{1}$ variables are the ones that can be used either as guards of a while loop or as a recursive argument in a method call whereas tier $\mathbf{0}$ variables are just used as a storage for computed data.

The main idea of the polynomial upper bound is as follows. If the input graph structure has size $n$ then the number of distinct possible configurations for $k$ tier $\mathbf{1}$ variables is at most $O(n^k)$. Consequently, we know that a terminating program will stop in a polynomial number of steps, based on the assumption that loops and recursive calls are only controlled by tier $\mathbf{1}$ variables.

There are several related works on the complexity of imperative and object oriented languages. On imperative languages, the papers [25], [24], [19] study theoretically the heap-space complexity of core-languages using type systems based on a matrices calculus. On OO programming languages, the papers [14], [15] control the heap-space consumption using type systems based on amortized complexity introduced in previous works on functional languages [13], [20], [6]. Though

hal-00773215, version 1 - 12 Jan 2013

similar, our result differs on several points with this line of work. First, our analysis is not restricted to linear heap-space upper bounds. Second, it also applies to stack-space upper bounds. Last but not least, our language is not restricted to the expressive power of method calls and includes a `while` statement, controlling the interlacing of such a purely imperative feature with functional features like recurrence being a very hard task from a complexity perspective. Still on OO programs, the paper [23] was a first attempt to control the heap-space through the use of interpretation methods coming from rewriting. Another interesting line of research is based on the analysis of heap-space and time consumption of Java bytecode [1], [2], [7]. The results from [1], [2] make use of abstract interpretations to infer efficiently symbolic upper bounds on resource consumption of Java programs. A constraint-based static analysis is used in [7] and focuses on certifying memory bounds for Java Card. Our analysis can be seen as a complementary approach since we try to obtain practical upper bounds through a cleaner theoretically oriented treatment. Consequently, this approach allows us to deal with our typing discipline on the original Java code without considering the corresponding Java bytecode.

A complex type-system that allows the programmer to verify linear properties on heap-space is presented in [8]. Our result in contrast presents a very simple type system that however guarantees a polynomial bound.

Lastly, we would like to mention an interesting line of work [16], [17] aiming at characterizing complexity classes below polynomial time. This work is based on a particular programming language called PURPLE combining imperative statements together with pointers on a fixed graph structure. Although not directly related, our type system was inspired by such a work.

The presented work is independent from termination analysis but our main result relies on such analysis. Indeed, Theorem 1 providing polynomial upper bounds on both the stack and the heap space consumption of a typed program only holds for a terminating computation. Consequently, our analysis can be combined with termination analysis in order to certify the upper bounds on any input. Possible candidates for the imperative fragment are *Size Change Termination* [4], [5], tools like Terminator [9] based on *Transition predicate abstraction* [26] or symbolic complexity bound generation based on abstract interpretations, see [10], [11] for example.

The paper outline is as follows. In Section II, we introduce the syntax of core Java and the notion of well-formed program. Section III describes the semantics of core Java based on graph structures called pointer graphs. In Section IV, the type system, which is the main contribution of the paper, is presented and explained. Section V is devoted to prove intermediate lemmata and our main result, Theorem 1. This section ends with direct corollaries and a result on the decidability of type inference. Section VI consists in the direct possible extensions of our language including inheritance and override.

## II. Core Java syntax

In this section, we introduce the syntax of the considered core Java language a strict but expressive subset of Java.

### A. Syntax of classes

Expressions, instructions, constructors, methods and classes are defined by the grammar of Figure 1,

Expressions $\ni E ::= \mathtt{x} \mid \mathtt{null} \mid \mathtt{this}^{\mathrm{C}} \mid \mathtt{true} \mid \mathtt{false} \mid op(E_1,\ldots,E_n) \mid \mathtt{new}\ \mathrm{C}(E_1,\ldots,E_n) \mid E.m(E_1,\ldots,E_n)$

Instructions $\ni I ::= \ ; \mid [\tau]\ \mathtt{x}{:=}E; \mid I_1\ I_2 \mid \mathtt{while}(E)\{I\} \mid \mathtt{if}(E)\{I_1\}\mathtt{else}\{I_2\} \mid E.m(E_1,\ldots,E_n);$

Methods $\ni M_{\mathrm{C}} ::= \tau\ m(\tau_1\ \mathtt{x}_1,\ldots,\tau_n\ \mathtt{x}_n)\{I[\mathtt{return\ x;}]\}$

Cons $\ni K_{\mathrm{C}} ::= \mathrm{C}(\tau_1\ \mathtt{y}_1,\ldots,\tau_n\ \mathtt{y}_n)\{\mathtt{x}_1{:=}\mathtt{y}_1;\ldots\mathtt{x}_n{:=}\mathtt{y}_n;\}$

Classes $\ni \mathfrak{C} ::= \mathrm{C}\{\tau_1\ \mathtt{x}_1;\ldots;\tau_n\ \mathtt{x}_n;\ K_{\mathrm{C}}\ M_{\mathrm{C}}^1\ldots M_{\mathrm{C}}^k\}$

Fig. 1: Syntax of core Java

with $\mathtt{x} \in \mathbb{V}$, $op \in \mathbb{O}$, $\mathrm{C} \in \mathbb{C}$, $m \in \mathbb{M}$, $\mathbb{V}$ being the set of variables, $\mathbb{O}$ the set of operators, $\mathbb{M}$ the set of method names and $\mathbb{C}$ the set of class names. The $\tau$s are type annotations ranging over $\mathbb{C} \cup \{\mathtt{void}, \mathtt{boolean}\}$. As usual, let $[e]$ denote some optional element $e$. Moreover, as in Java ; denote the empty instruction. The core Java syntax does not include a `for` instruction based on the premise that, as in Java, a for statement $\mathtt{for}(\tau\ \mathtt{x}{:=}E;\mathtt{condition};\mathtt{Increment})\{\mathtt{Ins}\}$ can be simulated by the while statement $\tau\ \mathtt{x}{:=}E;$ $\mathtt{while}(\mathtt{condition})\ \{\mathtt{Ins\ Increment};\}$. Also notice that there is no attribute access in our syntax using the . operator. Getters will be needed. Consequently, all attributes are implicitly `private`. On the opposite, methods and classes are all `public`.

**Definition 1.** *A core Java* program *is a collection of classes together with exactly one executable:*

$$Exe\{\mathit{main}()\{\tau_1\ x_1 := E_1;\ldots;\tau_n\ x_n := E_n; I\}\}$$

*In an executable, the instruction* $\tau_1\ x_1 := E_1;\ldots;\tau_n\ x_n := E_n;$ *is called the* initialization instruction *whereas $I$ is called the* computational instruction.

We adopt Java conventions. In a class $\mathfrak{C} = \mathrm{C}\{\tau_1\ \mathtt{x}_1;\ldots;\tau_n\ \mathtt{x}_n;\ K_{\mathrm{C}}\ M_{\mathrm{C}}^1\ldots M_{\mathrm{C}}^k\}$, the $\mathtt{x}_i$s are called attributes. Moreover let $\mathrm{C}.\mathcal{A}$ denote the set of the attributes of $\mathfrak{C}$, i.e. $\mathrm{C}.\mathcal{A} = \{\mathtt{x}_1,\ldots,\mathtt{x}_n\}$. In a method or constructor, the arguments are called parameters. We write $m \in \mathrm{C}$ to denote that the method name $m$ corresponds to one of the methods declared in $\mathfrak{C}$, that is there exists $j \leq k$ such that $M_{\mathrm{C}}^j = \tau\ m(\ldots)\{\ldots\}$. Moreover, given a method $\tau\ m(\tau_1\ \mathtt{x}_1,\ldots,\tau_n\ \mathtt{x}_n)\{I\ [\mathtt{return\ x;}]\}$, we say that its signature is $\tau\ m^{\mathrm{C}}(\tau_1,\ldots,\tau_n)$, if $m \in \mathrm{C}$. Finally, each variable declared in an assignment of the shape $\tau\ \mathtt{x}{:=}E;$ is called a local variable.

hal-00773215, version 1 - 12 Jan 2013

For readability, we have restricted classes to have exactly one constructor initializing all the class attributes. Moreover the only considered primitive data are boolean values `true` and `false`. This is not a restriction since other primitive data types such as floats, integers and characters could be considered, as explained in Subsection IV-D. In the initial syntax, there is no inheritance and, consequently, no overrides. In order to simplify the discussion, the treatment of these features will be delayed to Section VI. However note that overload is possible in the initial core Java syntax.

### B. Well-formed programs

Throughout the paper, only well-formed programs satisfying the following conditions will be considered:

- each class name C appearing in the collection of classes corresponds to exactly one class of name C within the collection.
- a variable appearing in the collection of classes is either a local variable, or an attribute or a parameter. For simplicity, we will suppose that the considered programs are statically transformed up-to $\alpha$-conversion so that each variable (local variable, attribute or parameter) has a distinct name, i.e. there are no name clashes.
- each local variable $\mathtt{x}$ is both declared and initialized exactly once by a $\tau\ \mathtt{x} := E$ instruction for its first use.
- the use of self reference $\mathtt{this}^{\mathrm{C}}$ is restricted to the methods of the class C.
- a method output type is `void` if and only if it has no **return** statement.
- each method signature is unique. Moreover, in a method signature $\tau\ m^{\mathrm{C}}(\tau_1, \ldots, \tau_n)$, for each $i$ we have $\tau_i \in \mathbb{C}$. This restriction prevents the programmer from using methods with boolean parameters. The only reason for this restriction to hold is to simplify program semantics.

## III. Core Java pointer graph semantics

In this section, a pointer graph semantics of core Java programs is provided. A pointer graph is basically a graph structure representing the memory heap, whose nodes are references, together with a mapping associating a reference to a given variable. The pointer graph semantics is designed to work on such a structure together with a stack, for method calls, and a store, for primitive values. The semantics will be defined on meta-instructions, flattened instructions with stack operations.

### A. Pointer graph

**Definition 2.** *A* pointer graph $\mathcal{G}_{\mathcal{P}}$ *is a directed graph* $\mathcal{G} = (V, A)$ *together with a mapping* $\mathcal{P}$.

*The nodes in $V$ are references labeled by class names and the arrows in $A$ link one reference to a reference of its attributes and are labeled by the attribute name. In what follows, let $l$ be the node label mapping from $V$ to $\mathbb{C}$ and $i$ be the arrow label mapping from $A$ to $\cup_{C\in\mathbb{C}} C.\mathcal{A}$.*

*The partial mapping $\mathcal{P} : \mathbb{V} \cup \{\mathtt{this}\} \mapsto V$ associates a node of the graph in $V$ to some variable in $\mathbb{V}$ or to the current object (denoted `this`) and is called a* pointer mapping. *Let dom$(\mathcal{P})$ to be domain of $\mathcal{P}$.*

The memory used by a core Java program will be represented by a pointer graph. This graph explicits the arborescent nature of objects: each constructor call will create a new node of the graph and arrows to its attributes. It also respects the dynamic binding principle found in Object Oriented Languages. Those arrows are annotated by the attribute name. The semantics of an assignment $\mathtt{x} := E$ consists in updating the pointer mapping in such a way that $\mathcal{P}(\mathtt{x})$ will be the reference of the object computed by $E$.

The heap in which the objects are stored corresponds to the graph. Consequently, bounding the heap memory use consists in bounding the size of the computed graph, the size of a graph being the number of nodes.

Figure 2 illustrates the pointer graph associated to a sequence of object creations. The figure contains both the graph of labeled nodes and arrows together with the pointer mapping whose domain is represented by boxed variables and whose application is symbolized by snake arrows.

```
B b := new B(new A(), new A());
C c := new C(b);
D d := new D(c);
B e := new B(c, c);
```

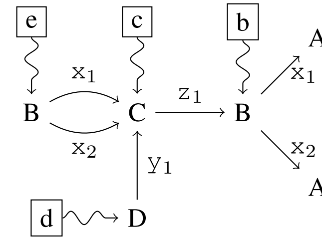


Fig. 2: Example of a pointer graph

### B. Pointer stack

The pointer stack of a program is used when calling a method: references to the parameters are pushed on the stack. In our context, the pointer stack will contain pointer mappings:

**Definition 3.** *A* pointer stack $\mathcal{S}_{\mathcal{G}}$ *is a LIFO structure of pointer mappings $\mathcal{S}$ corresponding to the same directed graph $\mathcal{G}$. Given a pointer stack $\mathcal{S}_{\mathcal{G}}$, define $\top\mathcal{S}$ to be the top pointer mapping of $\mathcal{S}$.*

Intuitively, the pointer mappings of a pointer stack $\mathcal{S}_{\mathcal{G}}$ map method parameters to the references of the arguments on which they are applied. Notice that all parameters can be mapped in such a way since they are of reference type by well-formedness assumption. Consequently, they are distinct from the pointer mapping in the pointer graph. For example, considering a method $m$ defined as $\tau\ m(\tau_1\ \mathtt{y})\{J; \mathtt{return}\ \mathtt{z}\}$ in a method call $\mathtt{x} := E.m(F)$; will push a new pointer graph $\mathcal{P}$ on pointer stack $\mathcal{S}_{\mathcal{G}}$ such that $\mathcal{P}(\mathtt{y})$ points to the node corresponding to the object computed by $F$. We will see in the next subsection that pop operation removing the top pointer

hal-00773215, version 1 - 12 Jan 2013

mapping from the pointer stack will correspond, as expected, to the evaluation of a return statement in a method body.

### C. Memory configuration

A *primitive store* $\sigma$ is a partial mapping $\sigma : \mathbb{V} \mapsto \{\texttt{true}, \texttt{false}\}$ associating a boolean value to some variable of primitive data type in $\mathbb{V}$. As usual, the domain of a primitive store $\sigma$ is denoted $dom(\sigma)$.

A memory configuration consists in a heap together with a stack and a store:

**Definition 4.** *A* memory configuration $\mathcal{C}$ *is a quadruple* $\langle \mathcal{G}, \mathcal{P}, \mathcal{S}, \sigma \rangle$ *such that* $\mathcal{G}_{\mathcal{P}}$ *is a pointer graph,* $\mathcal{S}_{\mathcal{G}}$ *is a pointer stack and* $\sigma$ *is a primitive store.*
*Among memory configurations, we distinguish the* initial configuration $\mathcal{C}_0$ *defined by* $\mathcal{C}_0 = \langle (\{\&\texttt{null}\}, \emptyset), \emptyset, [], \emptyset \rangle$ *where the notation* $\emptyset$ *is used both for empty set and empty mapping,* $[]$ *denotes the empty pointer stack, and* $\&\texttt{null}$ *is the reference of the* $\texttt{null}$ *object (i.e.* $l(\&\texttt{null}) = \texttt{null}$*).*

### D. Meta-language and flattening

The semantics of core Java programs will be defined on a meta-language of expressions and instructions. Meta-expressions are flat expressions. Meta-instructions consist in usual instructions flattened instructions and pop and push operations for managing method calls. Meta-expressions and meta-instructions are defined formally by the following grammar:

$$\begin{array}{lll} ME & ::= & \texttt{x} \mid \texttt{null} \mid \texttt{this}^{\texttt{C}} \mid \texttt{true} \mid op(\texttt{x}_1, \dots, \texttt{x}_n) \\ & & \mid \texttt{false} \mid \texttt{new C}(\texttt{x}_1, \dots, \texttt{x}_n) \mid \texttt{y}.m(\texttt{x}_1, \dots, \texttt{x}_n) \end{array}$$

$$\begin{array}{lll} MI & ::= & ; \mid [\tau]\ \texttt{x}{:=}ME; \mid MI_1\ MI_2 \mid \texttt{x}.m(\texttt{y}_1, \dots, \texttt{y}_n); \\ & & \mid \texttt{while}(\texttt{x})\{MI\} \mid \texttt{if}(\texttt{x})\{MI_1\}\texttt{else}\{MI_2\} \\ & & \mid \texttt{pop}; \mid \texttt{push}(\mathcal{P}); \mid \epsilon \end{array}$$

where $\epsilon$ denotes the empty meta-instruction.

Flattening an instruction $I$ into a meta-instruction $\overline{I}$ will consist in adding fresh intermediate variables for each complex parameter. This procedure is standard and defined in Figure 5 of Appendix C. The flattened meta-instruction will keep the semantics of the initial instruction unchanged. The main interest in such a program transformation is just that all the variables will be statically defined in a meta-instruction whereas they could be dynamically created by an instruction, hence allowing a cleaner semantic treatment of meta-instructions. We extend the flattening to methods and procedures by $\tau\ m(\tau_1\ \texttt{x}_1, \dots, \tau_n\ \texttt{x}_n)\{\overline{I}\ [\texttt{return x};]\}$ so that each instruction is flattened. A flattened program is the program obtained by flattening all the instructions in its methods. Notice that the flattening is a polynomially bounded program transformation.

**Lemma 1.** *Define the size of an instruction* $|I|$ *(respectively meta-instruction* $|MI|$*) to be the number of symbols in* $I$ *(resp.* $MI$*). For each instruction* $I$*, we have* $|\overline{I}| = O(|I|)$*.*

### E. Program semantics

Informally, the small step semantics $\to$ of core Java relates a pair $(\mathcal{C}, MI)$ of memory configuration $\mathcal{C}$ and meta-instruction $MI$ to another pair $(\mathcal{C}', MI')$ consisting of a new memory configuration $\mathcal{C}'$ and of the next meta-instruction $MI'$ to be executed. Let $\to^*$ (respectively $\to^+$) be its reflexive and transitive (respectively transitive) closure. Note that in special case where $(\mathcal{C}, MI) \to^* (\mathcal{C}', \epsilon)$ then we say that the meta-instruction $MI$ *terminates on memory configuration* $\mathcal{C}$.

**Definition 5.** *A core Java program of executable* $Exe\{\mathit{main}()\{\tau_1\ x_1 := E_1; \dots; \tau_n\ x_n := E_n; I\}\}$ *terminates if the following conditions hold:*

1) $(\mathcal{C}_0, \overline{\tau_1\ x_1 := E_1; \dots; \tau_n\ x_n := E_n;}) \to^* (\mathcal{C}, \epsilon)$
2) $(\mathcal{C}, \overline{I}) \to^* (\mathcal{C}', \epsilon)$

*The memory configuration* $\mathcal{C}$ *computed by the initialization instruction is called the* input.

Now we introduce some preliminary notations. Given a memory configuration $\mathcal{C} = \langle \mathcal{G}, \mathcal{P}, \mathcal{S}, \sigma \rangle$, let $\mathcal{C}(\texttt{x})$, intuitively the value of x, be defined by:

$$\mathcal{C}(\texttt{x}) = \begin{cases} \sigma(\texttt{x}) & \text{if } \texttt{x} \in dom(\sigma) \\ \top\mathcal{S}(\texttt{x}) & \text{if } \texttt{x} \in dom(\top\mathcal{S}) \\ \mathcal{P}(\texttt{x}) & \text{if } \texttt{x} \in dom(\mathcal{P}) \end{cases}$$

and let $\mathcal{C}[\mu : \texttt{x} \mapsto v]$, $\mu \in \{\sigma, \mathcal{P}, \top\mathcal{S}\}$, be a notation for the memory configuration $\mathcal{C}'$ that is equal to $\mathcal{C}$ but on $\mu$ where $\mathcal{C}'(\texttt{x}) = v$. Moreover let $\mathcal{C}[\mathcal{S} : \texttt{push}(\mathcal{P})]$ and $\mathcal{C}[\mathcal{S} : \texttt{pop}]$ be notations for the memory configuration where the pointer mapping $\mathcal{P}$ has been pushed to the top of the stack and where the top pointer mapping has been removed from the top of the stack, respectively. Finally, let $\mathcal{C}[V : v \mapsto \text{C}]$ denote a memory configuration $\mathcal{C}'$ whose graph contains the new node $v$ labeled by C (i.e. $l(v) = \text{C}$) and let $\mathcal{C}[A : (v, w) \mapsto \texttt{x}]$ denote a memory configuration $\mathcal{C}'$ whose graph contains the new arrow $(v, w)$ labeled by x (i.e. $i((v, w)) = \texttt{x}$. We defined $dom(\mathcal{C}) = dom(\mathcal{P}) \uplus dom(\top\mathcal{S}) \uplus dom(\sigma)$ (the domains are clearly disjoint by well-formedness. Hence $\mathcal{C}(\texttt{x})$ is clearly defined) and $[\![op]\!]$ to be the function computed by the language implementation of operator $op$.

The rules of $\to$ are defined formally in Figure 3. Let us explain the meaning of these rules. Rule (1) just consists in the evaluation of the empty instruction ';'.

Rules (2) to (8) are transitions for the distinct assignment of an expression to a variable. Rule (2) is the assignment of the null reference $\&\texttt{null}$ to a variable. Consequently, it updates the pointer mapping $\mathcal{P}$. Rule (3) is the assignment of a primitive boolean value to a variable. Consequently, it updates the primitive store $\sigma$. Rule (4) describes the assignment of a variable to another. It updates the primitive store if it is a primitive value, or updates the current pointer mapping or the top pointer mapping in the pointer stack, depending on whether the considered variable is a parameter or not. Rule (5) consists in the assignment of the self-reference. Consequently, it updates the pointer mapping $\mathcal{P}$ after searching the reference of the current object at the top of the pointer stack (i.e.

$$
\begin{array}{rcll}
(\mathcal{C},\ ;MI) & \to & (\mathcal{C}, MI) & (1)\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{null}; MI) & \to & (\mathcal{C}[\mathcal{P} : \mathtt{x} \mapsto \&\mathtt{null}], MI) & (2)\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}w; MI) & \to & (\mathcal{C}[\sigma : \mathtt{x} \mapsto w], MI) \quad w \in \{\mathtt{true}, \mathtt{false}\} & (3)\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{y}; MI) & \to & (\mathcal{C}[\mu : \mathtt{x} \mapsto \mathcal{C}(\mathtt{y})], MI) \quad \mu \in \{\sigma, \mathcal{P}, \top\mathcal{S}\} & (4)\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{this}^{\mathrm{C}}; MI) & \to & (\mathcal{C}[\mathcal{P} : \mathtt{x} \mapsto \top(\mathcal{S})(\mathtt{this})], MI) & (5)\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}op(\mathtt{y}_1, \dots, \mathtt{y}_n); MI) & \to & (\mathcal{C}[\sigma : \mathtt{x} \mapsto [\![op]\!](\mathcal{C}(\mathtt{y}_1), \dots, \mathcal{C}(\mathtt{y}_n))], MI) & (6)\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{new}\ \mathrm{C}(\mathtt{y}_1, \dots, \mathtt{y}_n); MI) & \to & (\mathcal{C}[V : v \mapsto \mathrm{C}][A : (v, \mathcal{C}(\mathtt{y}_i)) \mapsto \mathtt{z}_i][\mathcal{P} : \mathtt{x} \mapsto v], MI) & (7)\\
 & & \text{where } v \text{ is a fresh node and } \mathrm{C}.\mathcal{A} = \{\mathtt{z}_1, \dots, \mathtt{z}_n\} & \\
(\mathcal{C}, [[\tau]\ \mathtt{x}{:=}]\mathtt{y}_{n+1}.m(\mathtt{y}_1, \dots, \mathtt{y}_n); MI) & \to & (\mathcal{C}, \mathtt{push}(\{\mathtt{this} \mapsto \mathcal{C}(\mathtt{y}_{n+1}), \mathtt{z}_i \mapsto \mathcal{C}(\mathtt{y}_i)\}); MI'\ [\mathtt{x}{:=}\mathtt{z};]\ \mathtt{pop};\ MI) & (8)\\
 & & \text{if } m \text{ is a flattened method } \tau\ m(\tau_1\ \mathtt{z}_1, \dots \tau_n\ \mathtt{z}_n)\{MI'\ [\mathtt{return}\ \mathtt{z};]\} & \\
(\mathcal{C}, \mathtt{push}(\mathcal{P}); MI) & \to & (\mathcal{C}[\mathcal{S} : \mathtt{push}(\mathcal{P})], MI) & (9)\\
(\mathcal{C}, \mathtt{pop}; MI) & \to & (\mathcal{C}[\mathcal{S} : \mathtt{pop}], MI) & (10)\\
(\mathcal{C}, \mathtt{while}(\mathtt{x})\{MI'\}\ MI) & \to & (\mathcal{C},\ MI'\ \mathtt{while}(\mathtt{x})\{MI'\}\ MI) \quad \text{if } \mathcal{C}(\mathtt{x}) = \mathtt{true} & (11)\\
(\mathcal{C}, \mathtt{while}(\mathtt{x})\{MI'\}\ MI) & \to & (\mathcal{C},\ MI) \quad \text{if } \mathcal{C}(\mathtt{x}) = \mathtt{false} & (12)\\
(\mathcal{C}, \mathtt{if}(\mathtt{x})\{MI_{\mathtt{true}}\}\mathtt{else}\{MI_{\mathtt{false}}\}\ MI) & \to & (\mathcal{C}, MI_w\ MI) \quad \text{if } \mathcal{C}(\mathtt{x}) = w \in \{\mathtt{true}, \mathtt{false}\} & (13)
\end{array}
$$

Fig. 3: Semantics of core Java

$\top\mathcal{S}(\mathtt{this})$). Notice that such an assignment may only occur in a method body (because of well-formedness assumptions) and consequently the stack is non-empty and must contain a reference to `this`. Rule (6) consists in operator evaluation and updates the primitive store since operator outputs are restricted to be of `boolean` type. Rule (7) consists in the creation of a new instance. Consequently, this rule adds a new node $v$ of label C and the corresponding arrows $(v, \mathcal{C}(\mathtt{y}_i))$ of label $\mathtt{z}_i$ in the graph. $\mathcal{C}(\mathtt{y}_i)$ are the nodes of the graph corresponding to the parameters of the constructor call (or the boolean values if they are of type `boolean`) and $\mathtt{z}_i$ is the corresponding attribute name in the class C. Finally, this rule adds a link from the variable `x` to the new reference $v$ in the pointer mapping $\mathcal{P}$. Rule (8) consists in a call to method $m$. It adds a new instruction for pushing a new pointer mapping on the stack, containing references of the current object `this` on which $m$ is applied and references of the parameters. After adding the flattened body $MI'$ of $m$ to the evaluated instruction, it adds an assignment storing the returned value `z` in the assigned variable `x`, whenever the method is not a procedure, and a `pop;` instruction.

Rules (9) and (10) are standard rules for manipulating the pointer stack through the use of `pop` and `push` instructions.

Rules (11) to (13) are standard rules for control flow statements.

## IV. Type system

### A. Tiered types

The set of base types `T` is defined to be the set including a reference type C for each class name C and the special type `void` and the primitive type `boolean`. In other words, $\mathtt{T} = \{\mathtt{void}, \mathtt{boolean}\} \cup \mathbb{C}$.

*Tiers* are two elements of the lattice $(\{\mathbf{0}, \mathbf{1}\}, \vee, \wedge)$ where $\wedge$ and $\vee$ are the greatest lower bound operator and the least upper bound operator, respectively. The induced order, denoted $\preceq$, is such that $\mathbf{0} \preceq \mathbf{1}$. In what follows, let $\alpha, \beta, \dots$ denote tiers in $\{\mathbf{0}, \mathbf{1}\}$. Given a finite set of tiers indexed by the finite set $S$, $\{\alpha_i \mid i \in S\}$, let $\bigwedge_{i \in S} \alpha_i$ be defined inductively by:

$$
\bigwedge_{i \in S} \alpha_i = \begin{cases} \mathbf{1} \text{ if } S = \emptyset \\ \alpha_j \wedge (\bigwedge_{i \in S - \{j\}} \alpha_i), \text{ for some } j \in S, \text{ otherwise.} \end{cases}
$$

A *tiered type* is a pair $\tau(\alpha)$ consisting of a type $\tau \in \mathtt{T}$ together with a tier $\alpha \in \{\mathbf{0}, \mathbf{1}\}$. Given a tiered type, we define the two projections $\pi_1$ and $\pi_2$ as follows: $\pi_1(\tau(\alpha)) = \tau$ and $\pi_2(\tau(\alpha)) = \alpha$.

### B. Environments

A *variable typing environment* $\Gamma$ maps each variable in $\mathbb{V}$ to a tiered type. Intuitively, tier $\mathbf{0}$ will be used to type variables whose corresponding stored values might increase during a computation whereas tier $\mathbf{1}$ will be used to type variables used in the guard of a while loop or as a recursive argument of a method call. Consequently, the values stored in a tier $\mathbf{1}$ variable will not be allowed to increase during a computation. Given a variable typing environment $\Gamma$ and a tier $\alpha$, let $\Gamma_\alpha$ be the restriction of $\Gamma$ to variables `x` such that $\pi_2(\Gamma(\mathtt{x})) = \alpha$.

### C. Well-typed programs

*1) Operator signature:* The language is restricted to operators whose return type is `boolean`. An operator of arity $n$ comes equipped with a signature of the shape $\tau_1 \times \cdots \times \tau_n \to$ `boolean`, fixed by the language implementation. In the type

hal-00773215, version 1 - 12 Jan 2013

hal-00773215, version 1 - 12 Jan 2013

system, the notation $op :: \tau_1 \times \cdots \times \tau_n \to \texttt{boolean}$ denotes that $op$ has signature[1] $\tau_1 \times \cdots \times \tau_n \to \texttt{boolean}$.

*2) Judgments:* Expressions and instructions will be typed using tiered types whereas constructors and methods of arity $n$ have types of the shape: $\tau_1(\alpha_1) \times \ldots \times \tau_n(\alpha_n) \to \tau(\alpha)$. Given a variable typing environment $\Gamma$, there are four kinds of typing judgments:

- The judgment $\Gamma \vdash E : \tau(\alpha)$ means that expression $E$ corresponds to values of tiered type $\tau(\alpha)$.
- The judgment $\Gamma \vdash I : \texttt{void}(\alpha)$ is similar but the type is enforced to be `void`, meaning that instructions have no return value [2].
- The judgment $\Gamma \vdash K_{\mathrm{C}} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \mathrm{C}(\mathbf{0})$ enforces the output of a constructor to be of the correct type C and to be of tier $\mathbf{0}$, this important tiering restriction will prevent object instantiation in variables of tier $\mathbf{1}$.
- The last judgment $\Gamma \vdash M_{\mathrm{C}} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha)$ for methods is similar but unrestricted.

*3) Well-typedness:* Let us now introduce the notion of well-typed program. Intuitively, a well-typed program has an executable whose initialization instruction is only constrained by types and whose computational instruction is both constrained on types and tiers. The type system propagates these constraints on all the classes, methods and instructions used within these instructions.

**Definition 6** (Well-typed program)**.** *Given a program of executable Exe and a typing variable environment $\Gamma$, the judgment $\Gamma \vdash Exe : \diamond$ means that the program is well-typed wrt $\Gamma$.*

### D. *Typing rules*

*1) Expressions:* The typing rules for expressions are provided in Figure 4a.

Rules *(True)* and *(False)* mean that boolean constants are of type `boolean` and tier $\mathbf{1}$ as they cannot increase. It is possible to add other Java primitive data types such as float, integer, char. As for booleans, they will be associated to tiered types of tier $\mathbf{1}$ since a value of primitive data type can be considered as a constant. Note that this is counter-intuitive since a while loop controlled by a guard of primitive data type will be treated as a constant time instruction but not surprising since all primitive data type values are stored on a constant number of bits.

Rule *(Null)* means that, as in Java and for polymorphic reasons, `null` can be considered of any class C and of tier $\mathbf{1}$ as it cannot increase.

Rule *(Var)* is standard. The *(Self)* rule explicits that the self reference $\texttt{this}^{\mathrm{C}}$ belongs to class C and does not have any constraint on its tier.

Rule *(Op)* describes how to type an expression consisting of an operator of a given signature applied to $n$ arguments. The $n$ arguments must be expressions of types corresponding to the operator signature. The expressions must be of the same tier $\alpha$ which will also be the tier of the whole expression. It prevents information to flow from tier $\mathbf{0}$ to tier $\mathbf{1}$. Note that flows from tier $\mathbf{1}$ to tier $\mathbf{0}$ are also prohibited in this rule. This is not a restriction since they are useless: operators only return booleans and, consequently, their computations cannot increase the memory.

Rule *(New)* describes the typing of object instantiation. It checks that the constructor arguments have tiered types $\tau_i(\beta_i)$ of the same types $\tau_i$ and of tier not lower than the admissible tiers $\alpha_i$ in the constructor typing judgment. Note that the new instance has type of the right class and tier $\mathbf{0}$ since its creation makes the memory grow (hence it cannot be of tier $\mathbf{1}$).

Rule *(Call)* represents how to type method calls of the shape $E.m(E_1, \ldots, E_n)$. First, we need to check that the method $m$ exists in the class of $E$ (denoted $m \in \mathrm{C}$, provided that $\Gamma \vdash E : \mathrm{C}(\beta)$). We then check that the arguments' types match the parameters' types in $m$'s signature and that the tiers of those arguments $\beta_i$ are not lower than the tiers $\alpha_i$ in the method typing judgment. Moreover, the tier $\beta$ of the object $E$ must not be bigger than the minimum of the attributes' tiers, denoted $\Gamma(\mathrm{C})$ and defined by:

$$\Gamma(\mathrm{C}) = \bigwedge_{\mathrm{x} \in \mathrm{C}.\mathcal{A}} (\pi_2(\Gamma(\mathrm{x}))).$$

This means that an object of tier $\mathbf{1}$ cannot have attributes of tier $\mathbf{0}$, in other words, no arrow will go from a node corresponding to tier $\mathbf{1}$ to a node of tier $\mathbf{0}$. A last and important point to stress is that the tier of the evaluated expression (or instruction) in a method call matches the tier of the return variable in the method, hence avoiding forbidden information flows.

*2) Instructions:* The typing rules for instructions are provided in Figure 4b.

Rule *(Ass)* explains how to type an assignment: it is an instruction, hence of type `void`. It is only possible to assign an expression $E$ to a variable $\mathtt{x}$ if both the types match and the tier $\beta$ of $E$ is higher than the tier $\alpha$ of $\mathtt{x}$. The tier of the instruction will be $\alpha$. This rule implies that information may flow from tier $\mathbf{1}$ to tier $\mathbf{0}$ but not the contrary. In other words, a tier $\mathbf{1}$ variable cannot be assigned to in a tier $\mathbf{0}$ instruction block whereas a tier $\mathbf{0}$ variable can be assigned to without any constraint, hence allowing an implicit sub-typing for expressions. This rule can be used both if the assignment is a declaration (the type $\tau$ is given) or not.

Rule *(Sub)* is a sub-typing rule. An instruction of tier $\alpha$ can also be tiered by $\beta$ with $\alpha \preceq \beta$. This means that a tier $\mathbf{0}$ instruction, where tier $\mathbf{1}$ variables cannot be modified, can be considered as a tier $\mathbf{1}$ instruction where tier $\mathbf{1}$ variables might be modified, thus relaxing confidentiality constraints.

Rule *(Seq)* types the sequence of two instructions $I_1$ and $I_2$. Once again, the type of instructions is `void`. The sequence's tier will be the maximum of the tiers of $I_1$ and $I_2$. The intuition for taking the maximum is the same the one in the *(Sub)* rule.

Rule *(If)* describes the typing discipline for a $\texttt{if}(E)\{I_1\}\texttt{else}\{I_2\}$ statement. $E$ needs to be a boolean expression of tier $\alpha$. $I_1$ and $I_2$ are instructions, hence of type

[1] For simplicity, each operator is supposed to have a single signature. This is a slight distinction with Java to simplify our treatment. Note however that multiple signatures could be handled by the complexity analysis.

[2] This is also a minor distinction with Java, where the assignment has return type of the evaluated expression, used in order to simplify the type system.

hal-00773215, version 1 - 12 Jan 2013

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{boolean}(\mathbf{1})}\ \textit{(True)} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{boolean}(\mathbf{1})}\ \textit{(False)} \qquad \frac{}{\Gamma \vdash \texttt{null} : \mathbf{C}(\mathbf{1})}\ \textit{(Null)}$$

$$\frac{\Gamma(\texttt{x}) = \tau(\alpha)}{\Gamma \vdash \texttt{x} : \tau(\alpha)}\ \textit{(Var)} \qquad \frac{}{\Gamma \vdash \texttt{this}^{\mathbf{C}} : \mathbf{C}(\alpha)}\ \textit{(Self)} \qquad \frac{\forall i,\ \Gamma \vdash E_i : \tau_i(\alpha) \qquad op :: \tau_1 \times \cdots \times \tau_n \to \texttt{boolean}}{\Gamma \vdash op(E_1, \ldots, E_n) : \texttt{boolean}(\alpha)}\ \textit{(Op)}$$

$$\frac{\forall i\ \Gamma \vdash E_i : \tau_i(\beta_i) \quad \alpha_i \preceq \beta_i \quad \Gamma \vdash \mathbf{C}(\tau_1\ \texttt{y}_1, \ldots, \tau_n\ \texttt{y}_n)\{\texttt{x}_1 := \texttt{y}_1; \ldots \texttt{x}_n := \texttt{y}_n\} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \mathbf{C}(\mathbf{0})}{\Gamma \vdash \texttt{new}\ \mathbf{C}(E_1, \ldots, E_n) : \mathbf{C}(\mathbf{0})}\ \textit{(New)}$$

$$\frac{\forall i\ \Gamma \vdash E_i : \tau_i(\beta_i) \quad \alpha_i \preceq \beta_i \quad \Gamma \vdash m : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha) \quad \Gamma \vdash E : \mathbf{C}(\beta) \quad \beta \preceq \Gamma(\mathbf{C}) \quad m \in \mathbf{C}}{\Gamma \vdash E.m(E_1, \ldots, E_n) : \tau(\alpha)}\ \textit{(Call)}$$

(a) Expressions

$$\frac{\Gamma \vdash \texttt{x} : \tau(\alpha) \qquad \Gamma \vdash E : \tau(\beta) \quad \alpha \preceq \beta}{\Gamma \vdash [\tau]\ \texttt{x}{:=}E; : \texttt{void}(\alpha)}\ \textit{(Ass)} \qquad \frac{\Gamma \vdash I : \texttt{void}(\alpha) \quad \alpha \preceq \beta}{\Gamma \vdash I : \texttt{void}(\beta)}\ \textit{(Sub)}$$

$$\frac{\forall i,\ \Gamma \vdash I_i : \texttt{void}(\alpha_i)}{\Gamma \vdash I_1\ I_2 : \texttt{void}(\alpha_1 \vee \alpha_2)}\ \textit{(Seq)} \qquad \frac{\Gamma \vdash E : \texttt{boolean}(\mathbf{1}) \quad \Gamma \vdash I : \texttt{void}(\mathbf{1})}{\Gamma \vdash \textbf{while}(E)\{I\} : \texttt{void}(\mathbf{1})}\ \textit{(Wh)}$$

$$\frac{}{\Gamma \vdash\ ;\ : \texttt{void}(\mathbf{0})}\ \textit{(Skip)} \qquad \frac{\Gamma \vdash E : \texttt{boolean}(\alpha) \quad \forall i,\ \Gamma \vdash I_i : \texttt{void}(\alpha)}{\Gamma \vdash \textbf{if}(E)\{I_1\}\textbf{else}\{I_2\} : \texttt{void}(\alpha)}\ \textit{(If)}$$

(b) Instructions

$$\frac{\forall i,\ \Gamma \vdash \texttt{y}_i : \tau_i(\alpha_i)}{\Gamma \vdash \mathbf{C}(\tau_1\ \texttt{y}_1, \ldots, \tau_n\ \texttt{y}_n)\{\texttt{x}_1 := \texttt{y}_1; \ldots \texttt{x}_n := \texttt{y}_n\} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \mathbf{C}(\mathbf{0})}\ (K_C)$$

$$\frac{\forall i,\ \Gamma \vdash \texttt{x}_i : \tau_i(\alpha_i) \qquad \Gamma \vdash I : \texttt{void}(\alpha)}{\Gamma \vdash \texttt{void}\ m(\tau_1\ \texttt{x}_1, \ldots, \tau_n\ \texttt{x}_n)\{I\} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \texttt{void}(\alpha)}\ (M_C^{void})$$

$$\frac{\forall i,\ \Gamma \vdash \texttt{x}_i : \tau_i(\alpha_i) \qquad \Gamma \vdash \texttt{x} : \tau(\alpha) \qquad \Gamma \vdash I : \texttt{void}(\alpha)}{\Gamma \vdash \tau\ m(\tau_1\ \texttt{x}_1, \ldots, \tau_n\ \texttt{x}_n)\{I\ \texttt{return}\ \texttt{x};\} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha)}\ (M_C)$$

$$\frac{\Gamma \vdash I : \texttt{void}(\mathbf{1}) \quad \forall i,\ \Gamma \vdash \texttt{x}_i : \tau_i(\alpha_i) \quad \forall i,\ \Gamma \vdash E_i : \tau_i(\beta_i)}{\Gamma \vdash \text{Exe}\{\texttt{main}()\{\tau_1\ \texttt{x}_1 := E_1; \ldots; \tau_n\ \texttt{x}_n := E_n; I\}\} : \diamond}\ \textit{(Main)}$$

(c) Constructors, methods and executable

Fig. 4: Type system for core Java

void, with the same tier $\alpha$. This prevents assignments of tier $\mathbf{1}$ variables in the instructions $I_1$ and $I_2$ to be controlled by a tier $\mathbf{0}$ expression.

Rule *(Skip)* is standard. ';' has type void and is of tier $\mathbf{0}$ since it has no complexity.

Rule *(Wh)* is the most important typing rule as it will constrain the use of while loops. In a statement $\textbf{while}(E)\{I\}$, the guard of the loop $E$ must be a boolean expression of tier $\mathbf{1}$ so that the guard is controlled. The instruction $I$, of type void, has to be of tier $\mathbf{1}$ since we expect the guard variables to be modified (i.e. assigned to). The whole statement is an instruction of type void and tier $\mathbf{1}$.

*3) Methods, constructors and executable:* The typing rules for constructors and methods are provided in Figure 4c.

Rule *$(K_C)$* describes the typing of a constructor definition. Constructors are of fixed form, so the only thing to check is that the parameters are of the desired tiered types. As explained in Rule *(New)* the output tier can only be $\mathbf{0}$.

Rules *$(M_C^{void})$* and *$(M_C)$* show how to type method definitions in the case where the method is a procedure (i.e. there is no return statement) and in the case where it returns a value. A procedure is defined with the void return type. If a method has a return type $\tau$, its body must finish by a **return** x statement with x of tiered-type $\tau(\alpha)$. In this case, the output type of the method will also be $\tau(\alpha)$. In both cases, the types and number of parameters need to match the method signature, the instruction $I$ in the body of the method needs to be of type $\texttt{void}(\alpha)$, i.e. the tier matches the output tier so

hal-00773215, version 1 - 12 Jan 2013

that there is no forbidden information flow.

Finally, typing an executable is done through the rule *(Main)* and consists in verifying that the initialization instruction respects types and that the computational instruction (denoted by instruction $I$) is of tier $\mathbf{1}$. Notice that no tier constraints are checked in the initialization instruction: this means that we do not control the complexity of this latter instruction ; the main reason for this choice is that this instruction is considered to be building the program input. In opposition, the computational instruction $I$ is considered to be the computational part of the program and has to respect the tiering discipline.

### E. Type preservation under flattening

We show that the flattening of a typable instruction has a type preservation property. A direct consequence is that the flattened program can be considered instead of the initial program.

**Proposition 1.** *Given an instruction $I$ and a typing variable environment $\Gamma$ such that $\Gamma \vdash I : \mathtt{void}(\alpha)$ holds, there is a typing variable environment $\Gamma'$ such that the following holds:*

- $\forall x \in dom(\Gamma),\ \Gamma(x) = \Gamma'(x)$
- $\Gamma' \vdash \overline{I} : \mathtt{void}(\alpha)$

*Conversely, if $\Gamma' \vdash \overline{I} : \mathtt{void}(\alpha)$, then $\Gamma' \vdash I : \mathtt{void}(\alpha)$.*

## V. Upper bound on the stack size and the heap size

### A. Definitions

In this section, we will state our main result showing that well-formed and typed programs have both pointer stack size and pointer graph size bounded polynomially by the input size under termination and safety assumptions. Moreover, for a given core Java program, a precise upper bound can be extracted. For that purpose, we need to define the notion of size for pointer stack, pointer graph and memory configuration.

**Definition 7** (Sizes)**.**

- *The size of a pointer graph $\mathcal{G}_\mathcal{P}$ is defined to be the number of nodes in $\mathcal{G}$ and denoted $|\mathcal{G}_\mathcal{P}|$.*
- *The size of a pointer stack $\mathcal{S}_\mathcal{G}$ is defined to be the number of pointer mappings in the stack $\mathcal{S}$ and denoted $|\mathcal{S}_\mathcal{G}|$.*
- *The size of a memory configuration $\langle \mathcal{G}, \mathcal{P}, \mathcal{S}, \sigma\rangle$ is equal to $|\mathcal{G}_\mathcal{P}| + |\mathcal{S}_\mathcal{G}| + |dom(\mathcal{P})| + |dom(\sigma)|$.*

Since a pointer graph contains both references to the objects (the nodes) and references to the attribute instances (the arrows), it would make sense to bound both the number of nodes and the number of arrows in order to control the heap-space, for a practical application. Notice that the outdegree of a node is bounded by a constant of the program (the maximum number of attributes in a class) and, consequently, bounding the number of nodes is sufficient to obtain a big $O$ bound. The size of a pointer stack is very close to the size of the Java Virtual Machine stack since it counts the number of nested method calls.

Given two methods $M_\mathrm{C}$ and $M'_{\mathrm{C}'}$ of respective signatures $s$ and $s'$ and respective names $m$ and $m'$, define the relation $\sqsubset$ on method signatures by $s \sqsubset s'$ if $m'$ is called in $M_\mathrm{C}$, i.e. in the body of $M_\mathrm{C}$ (this check is fully static as long as we do not consider inheritance). Let $\sqsubset^+$ be its transitive closure. A method of signature $s$ is *recursive* if $s \sqsubset^+ s$ holds. Given two method signatures $s$ and $s'$, $s \equiv s'$ holds if both $s \sqsubset^+ s'$ and $s' \sqsubset^+ s$ hold. Given a signature $s$, the equivalence class $[s]$ is defined as usual by $[s] = \{s' \mid s' \equiv s\}$. When the signature $s$ of a given method $M_\mathrm{C}$ of name $m$ is clear from the context, we will write $[m]$ as an abuse of notation for $[s]$ and say that $M_\mathrm{C}$ is a recursive method. Finally, we write $s \sqsubsetneq^+ s'$ if $s \sqsubset^+ s'$ holds and $s' \sqsubset^+ s$ does not hold.

The notion of level of a meta-instruction is introduced to compute an upper bound on the number of recursive steps for a method call evaluation.

**Definition 8** (Level)**.** *Let the level $\lambda$ of a method signature be defined as follows:*

- $\lambda(s) = 1$ *if* $s \notin [s]$
- $\lambda(s) = \max\{1 + \lambda(s') \mid s \sqsubsetneq^+ s'\}$ *otherwise.*

*As usual, we will write by abuse of notation $\lambda(m)$ whenever the signature of $m$ is clear from the context. Moreover, let $\lambda$ be the maximal level of a method within a given program.*

The notion of intricacy corresponds to the number of nested `while` loops in a meta-instruction and will be used to compute the requested upper bounds.

**Definition 9** (Intricacy)**.** *Let the intricacy $\nu$ of a meta-instruction be defined as follows:*

- $\nu(;) = \nu(pop;) = \nu(push(\mathcal{P});) = \nu(x{:=}ME;) = 0$
- $\nu(MI\ \ MI') = \max(\nu(MI), \nu(MI'))$
- $\nu(\mathtt{if}(x)\{MI\}\mathtt{else}\{MI'\}) = \max(\nu(MI), \nu(MI'))$
- $\nu(\mathtt{while}(x)\{MI\}) = 1 + \nu(MI)$

*Moreover, let $\nu$ be the maximal intricacy of a meta-instruction within a given program.*

Notice that both intricacy $\nu$ and level $\lambda$ are bounded by the size of their corresponding program.

### B. Safety restriction on recursive methods

Now we put some aside restrictions on recursive methods to ensure that their computations remain polynomially bounded. Recursive methods will be restricted to have only one recursive call and no while loop in their body (to prevent exponential growth) and must have tier $\mathbf{1}$ input (as the guard of a `while`) and output (to prevent a recursive dependence on a tier $\mathbf{0}$ variable).

**Definition 10** (Safety)**.** *A well-typed program with respect to a variable typing environment $\Gamma$ is* safe *if for each recursive method $M_C = \tau\ m(\ldots)\{MI\ [\mathtt{return}\ x;]\}$:*

- *there is exactly one call to some $m' \in [m]$ in MI,*
- *there is no while loop inside MI, i.e. $\nu(MI) = 0$,*
- *and the following judgment can be derived:*

$$\Gamma \vdash M_C : \tau_1(\mathbf{1}) \times \cdots \times \tau_n(\mathbf{1}) \to \tau(\mathbf{1}).$$

**Remark 1.** *A program is safe with respect to a variable typing environment $\Gamma$ iff its flattened version is safe with respect to the variable environment of Proposition 1.*

hal-00773215, version 1 - 12 Jan 2013

### C. Intermediate lemmata

In this section, we introduce intermediate lemmata that allow us to prove the main result. Given a memory configuration $\mathcal{C}$ and a variable typing environment $\Gamma$, define the *tier* $\mathbf{1}$ *memory configuration* $\mathcal{C}_{\Gamma_\mathbf{1}}$ by:

$$\mathcal{C}_{\Gamma_\mathbf{1}}(\mathrm{x}) = \begin{cases} \mathcal{C}(\mathrm{x}) & \text{if } \mathrm{x} \in dom(\Gamma_\mathbf{1}) \\ \bot & \text{otherwise} \end{cases}$$

where the symbol $\bot$ means that $\mathcal{C}_{\Gamma_\mathbf{1}}$ is undefined on the given input. Given a configuration $\mathcal{C}$ and a meta-instruction $MI$, the *distinct tier* $\mathbf{1}$ *configuration sequence* $\delta_{\Gamma_\mathbf{1}}(\mathcal{C}, MI)$ wrt variable typing environment $\Gamma$, is defined by:

- If $(\mathcal{C}, MI) \to (\mathcal{C}', MI')$ then:

$$\delta_{\Gamma_\mathbf{1}}(\mathcal{C}, MI) = \begin{cases} \mathcal{C}_{\Gamma_\mathbf{1}}.\delta_{\Gamma_\mathbf{1}}(\mathcal{C}', MI') & \text{if } \mathcal{C}_{\Gamma_\mathbf{1}} \neq \mathcal{C}'_{\Gamma_\mathbf{1}} \\ \delta_{\Gamma_\mathbf{1}}(\mathcal{C}', MI') & \text{otherwise} \end{cases}$$

- If $MI = \epsilon$ then $\delta_{\Gamma_\mathbf{1}}(\mathcal{C}, MI) = \mathcal{C}_{\Gamma_\mathbf{1}}$.

As usual, the size $|s|$ of a sequence $s$ (respectively the cardinal $\#S$ of a set $S$) is the number of elements in $s$ (resp. $S$).

Informally, $\delta_{\Gamma_\mathbf{1}}(\mathcal{C}, MI)$ is a record of the distinct tier $\mathbf{1}$ memory configurations encountered during the evaluation of $(\mathcal{C}, MI)$. Now we can show a non-interference property à la Volpano et al. [27] stating that given a safe program, there is no information flow from tier $\mathbf{0}$ variables to $\mathbf{1}$ variables.

**Lemma 2** (Non-interference)**.** *Given a meta-instruction $MI$ of a safe program with respect to typing variable environment $\Gamma$, let $\mathcal{C}$ and $\mathcal{C}'$ be two memory configurations, if $\mathcal{C}_{\Gamma_\mathbf{1}} = \mathcal{C}'_{\Gamma_\mathbf{1}}$ then $\delta_{\Gamma_\mathbf{1}}(\mathcal{C}, MI) = \delta_{\Gamma_\mathbf{1}}(\mathcal{C}', MI)$. In other words, tier $\mathbf{1}$ variables do not depend on tier $\mathbf{0}$ variables.*

Using Lemma 2, if a safe program evaluation encounters twice the same meta-instruction under two configurations equal on tier $\mathbf{1}$ variables then the considered meta-instruction does not terminate on both configurations.

**Lemma 3.** *Given a memory configuration $\mathcal{C}$ and a meta-instruction $MI$ of a safe program with respect to typing variable environment $\Gamma$, if $(\mathcal{C}, MI) \to^+ (\mathcal{C}', MI)$ and $\mathcal{C}_{\Gamma_\mathbf{1}} = \mathcal{C}'_{\Gamma_\mathbf{1}}$, then the meta-instruction $MI$ does not terminate on memory configuration $\mathcal{C}$.*

Lemma 3 permits to demonstrate that the number of distinct tier $\mathbf{1}$ memory configurations encountered during the evaluation of a terminating and safe program is polynomially bounded in the input size.

**Lemma 4.** *Given an input $\mathcal{C}$ and a meta-instruction $MI$ of a safe program with respect to variable typing environment $\Gamma$, the following holds:*

$$\#\left\{\mathcal{C}'_{\Gamma_1} \mid (\mathcal{C}, MI) \to^* (\mathcal{C}', MI')\right\} \leq |\mathcal{C}|^{\#dom(\Gamma_1)}.$$

### D. Main result

Now we can prove the polynomial upper bounds on the stack and the heap using intermediate Lemmata.

**Theorem 1.** *If a core Java program of computational instruction $I$ is safe wrt to variable typing environment $\Gamma$ and terminates on input $\mathcal{C}$ then for each memory configuration $\mathcal{C}'$ and meta-instruction $MI$ s.t. $(\mathcal{C}, \overline{I}) \to^* (\mathcal{C}', MI)$ we have:*

$$|\mathcal{C}'| = O(|\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times\lambda)}).$$

*In other words, if $\mathcal{C}' = \langle \mathcal{G}, \mathcal{P}, \mathcal{S}, \sigma\rangle$ then both $|\mathcal{G}_\mathcal{P}|$ and $|\mathcal{S}_\mathcal{G}|$ are in $O(|\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times\lambda)})$.*

As a corollary, if the program terminates on all input configurations, then we may infer a polynomial time upper bound on its execution time.

Another corollary of interest is that tier $\mathbf{1}$ variables remain polynomially bounded even if the program does not terminate. This is particularly interesting in the sense that we can guarantee security properties on the data stored in such variables even if we are unable to prove program termination.

A last and direct result is that our characterization is complete with respect to the class of functions computable in polynomial time as a direct consequence of Marion's result [22] since both our language and type system can be viewed as an extension of the considered imperative language. This means that our type system has a good expressivity.

### E. Type inference

**Proposition 2** (Type inference)**.** *Deciding if there exists a variable typing environment $\Gamma$ such that typing rules are satisfied can be done in time linear in the size of the program.*

## VI. Extension to inheritance

In this section, we present an extension of the language with inheritance and provide some adjustments needed in our analysis in order to preserve the stack and heap-space upper bounds of Theorem 1. Inheritance is a major trait of object oriented programming languages that has to be treated by any reasonable static analysis tool on Java like programs. In order to simplify the discussion, we have made the choice to hide this feature for a while since it does not add extra complexity in terms of heap and stack size. We explain in this section how the exposed methodology can deal with inheritance.

### A. Syntax

We extend the class grammar by class declarations of the shape: D `extends` C$\{\tau_1\ \mathrm{x}_1; \ldots; \tau_n\ \mathrm{x}_n;\ K_\mathrm{D}\ M^1_\mathrm{D} \ldots M^k_\mathrm{D}\}$ with $\mathrm{D} \in \mathbb{C}$ and $K_\mathrm{D}$ being a constructor initializing both the attributes of C and the attributes of D with respect to the parameters given as input. As in Java multiple inheritance is prohibited. Inheritance defines a partial order on classes denoted by $\mathrm{D} \trianglelefteq \mathrm{C}$. Considering this extended syntax makes method overriding, subtyping and polymorphism possible.

### B. Semantics

The semantics can be extended by creating a new node of label D in the graph each time a `new` D$(\ldots)$ expression is evaluated. The only difficulty to face is the semantics of method calls. As in Java, the method to be executed can only be chosen dynamically as it can be overridden in the subclass

hal-00773215, version 1 - 12 Jan 2013

to which the object belongs. Consequently, a check on the current object type has to be done before evaluating the method call. Once the type D is known the evaluation search for the method signature in the corresponding class and evaluates its body once founded. In the particular, case where this signature does not exist, the search is extended to the super class C and, so on. This is the main reason for single inheritance and it allows polymorphic programming style.

### C. Type system

The corresponding type system has to be extended in two ways. First, it must allows polymorphism but must also keep the tiers unchanged to prevent information flows. This can be done by adding the following rule:

$$\frac{\Gamma \vdash E : \mathrm{D}(\alpha) \quad \mathrm{D} \trianglelefteq \mathrm{C}}{\Gamma \vdash E : \mathrm{C}(\alpha)} \textit{(Pol)}$$

Second, it must check the tiered types of overridden methods in such a way that they are at least as liberal on their arguments tiers and as constrained on the output tier:

$$\frac{\forall i,\ \beta_i \preceq \alpha_i \qquad \alpha \preceq \beta \qquad \mathrm{D} \trianglelefteq \mathrm{C} \qquad \Gamma \vdash \tau\ M_{\mathrm{C}} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha)}{\Gamma \vdash \tau\ M_{\mathrm{D}} : \tau_1(\beta_1) \times \cdots \times \tau_n(\beta_n) \to \tau(\beta)} \textit{(OverR)}$$

provided that $M_{\mathrm{D}}$ overrides $M_{\mathrm{C}}$. Finally, the constructor of the subclass has to follow a similar rule on its inherited attributes.

### D. Safety

Now a method call can be performed dynamically depending on the current object type during program evaluation. This can lead to the creation of unexpected recursive calls. Hence the safety notion has to be changed in order to capture this behavior. For that purpose, it just suffices to extend the notion of recursive method signature by the following rule:

$$\tau\ m^{\mathrm{C}}(\tau_1, \ldots, \tau_n) \sqsubset \tau\ m^{\mathrm{D}}(\tau_1, \ldots, \tau_n),\ \text{if } \mathrm{D} \trianglelefteq \mathrm{C}$$

That is, a method $m^{\mathrm{C}}$ is considered to call its override $m^{\mathrm{D}}$ by dynamic binding.

## VII. Conclusion

This work presents a simple type-system (it can be checked in linear time) that provides explicit polynomial upper bounds on the heap and stack size of an object oriented program allowing method calls (including recursive) and inheritance. As the system is purely static, the bounds are not as tight as may be desirable. It would indeed be possible to refine the framework to obtain a better exponent at the price of a non-uniform formula (for example not considering all tier **1** variables but only those modified in each while loop or recursive method would reduce the computed complexity). OO features, such as abstract classes, interfaces and static attributes and methods, were not considered here, but we claim that they can also be treated by our analysis. Also note that the safety condition can be alleviated on recursive methods by ensuring that only one recursive call is reachable in the execution of the method body. To conclude, the presented work has some inherent limits on Java constructs that break control flows (like exception handlers, `break` or `return` statements). This is the reason why `return` statement uses are restricted in a method body. A more liberal use would break the non-interference property of Lemma 2 (e.g., a `break` statement depending on a conditional of tier **0** inside a while loop). We let the analysis of such statements as an open issue.

## References


[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, “Costa: Design and implementation of a cost and termination analyzer for java bytecode,” in *FMCO*, ser. LNCS, vol. 5382, 2007, pp. 113–132.

[2] ——, “Cost analysis of object-oriented bytecode programs,” *Theor. Comput. Sci.*, vol. 413, no. 1, pp. 142–159, 2012.

[3] S. Bellantoni and S. Cook, “A new recursion-theoretic characterization of the poly-time functions,” *Comput. Complex.*, vol. 2, pp. 97–110, 1992.

[4] A. M. Ben-Amram, “Size-change termination, monotonicity constraints and ranking functions,” *Log. Meth. Comput. Sci.*, vol. 6, no. 3, 2010.

[5] A. M. Ben-Amram, S. Genaim, and A. N. Masud, “On the termination of integer loops,” in *VMCAI*, ser. LNCS, vol. 7148, 2012, pp. 72–87.

[6] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska, “Automatic certification of heap consumption,” in *LPAR*, ser. Lecture Notes in Computer Science, vol. 3452, 2004, pp. 347–362.

[7] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider, “Certified memory usage analysis,” in *FM 2005: Formal Methods*, ser. LNCS, vol. 3582, 2005, pp. 91–106.

[8] W. Chin, H. Nguyen, S. Qin, and M. Rinard, “Memory usage verification for OO programs,” in *Static Analysis, SAS 2005*, 2005, pp. 70–86.

[9] B. Cook, A. Podelski, and A. Rybalchenko, “Terminator: Beyond safety,” in *CAV*, ser. LNCS, vol. 4144, 2006, pp. 415–426.

[10] S. Gulwani, “Speed: Symbolic complexity bound analysis,” in *CAV*, ser. LNCS, vol. 5643, 2009, pp. 51–62.

[11] S. Gulwani, K. K. Mehra, and T. M. Chilimbi, “Speed: precise and efficient static estimation of program computational complexity,” in *POPL*. ACM, 2009, pp. 127–139.

[12] E. Hainry, J.-Y. Marion, and R. Péchoux, “Type-based complexity analysis for fork processes,” in *FOSSACS*, ser. LNCS, 2013, to appear.

[13] M. Hofmann and S. Jost, “Static prediction of heap space usage for first-order functional programs,” in *POPL*. ACM, 2003, pp. 185–197.

[14] ——, “Type-based amortised heap-space analysis,” in *ESOP*, ser. LNCS, vol. 3924, 2006, pp. 22–37.

[15] M. Hofmann and D. Rodriguez, “Efficient type-checking for amortised heap-space analysis,” in *CSL*, ser. LNCS, vol. 5771, 2009, pp. 317–331.

[16] M. Hofmann and U. Schöpp, “Pointer programs and undirected reachability,” in *LICS*. IEEE Computer Society, 2009, pp. 133–142.

[17] ——, “Pure pointer programs with iteration,” *ACM Trans. Comput. Log.*, vol. 11, no. 4, 2010.

[18] A. Igarashi, B. C. Pierce, and P. Wadler, “Featherweight java: a minimal core calculus for java and gj,” *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 3, pp. 396–450, 2001.

[19] N. D. Jones and L. Kristiansen, “A flow calculus of *wp*-bounds for complexity analysis,” *ACM Trans. Comput. Log.*, vol. 10, no. 4, 2009.

[20] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann, “Static determination of quantitative resource usage for higher-order programs,” in *POPL*, 2010, pp. 223–236.

[21] D. Leivant and J.-Y. Marion, “Lambda calculus characterizations of poly-time,” *Fundam. Inform.*, vol. 19, no. 1/2, pp. 167–184, 1993.

[22] J.-Y. Marion, “A type system for complexity flow analysis,” in *LICS*, 2011, pp. 123–132.

[23] J.-Y. Marion and R. Péchoux, “Analyzing the implicit computational complexity of object-oriented programs,” in *FSTTCS*, ser. LIPIcs, vol. 2, 2008, pp. 316–327.

[24] J.-Y. Moyen, “Resource control graphs,” *ACM Trans. Comput. Logic*, vol. 10, no. 4, pp. 29:1–29:44, 2009.

[25] K.-H. Niggl and H. Wunderlich, “Certifying polynomial time and linear/polynomial space for imperative programs,” *SIAM J. Comput.*, vol. 35, no. 5, pp. 1122–1147, 2006.

[26] A. Podelski and A. Rybalchenko, “Transition predicate abstraction and fair termination,” in *POPL*. ACM, 2005, pp. 132–144.

[27] D. Volpano, C. Irvine, and G. Smith, “A sound type system for secure flow analysis,” *J. Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996.

hal-00773215, version 1 - 12 Jan 2013

## APPENDIX A
## EXAMPLE

Let us apply our framework to a simple list sorting program. We define two classes : `BList` for encoding binary integers as binary lists (with the least significant bit in head) and `IList` for encoding list of integers. Tiers are made explicit for the important functions, instructions and variables: The notation $\mathtt{x}^\alpha$ means that `x` has tier $\alpha$ under the considered typing variable environment $\Gamma$, i.e. $\Gamma \vdash \mathtt{x} : \tau(\alpha)$, for some $\tau$, whereas $I : \alpha$ means that $\Gamma \vdash I : \mathtt{void}(\alpha)$.

```
BList { //Binary integers: boolean list
  boolean value;
  BList queue;

  BList(boolean v, BList q) {
    value = v;
    queue = q;
  }

  //getQueue :: BList(1) or
  //getQueue::BList(0)
  BList getQueue() {
    return queue;
  }

  //setQueue :: BList(1) → void(1),
  //setQueue :: BList(0) → void(0) or
  //setQueue :: BList(1) → void(0)
  void setQueue(BList q) {
    queue = q;
  }

  //getValue :: boolean(1)
  //or getValue::boolean(0)
  boolean getValue() {
    return value;
  }

  //double::BList(0)
  BList double() {
    BList n^0 = new BList(false, this);
    return n^0;
  }

  // recursive method
  // decrement :: void(1)
  void decrement() {
    if (value^1 == true or value^1 == null) {
      value^1 = false; :1
    } else {
      if (queue^1 != null) {
        value = true;
        queue.decrement(); :1
      } else {
        value = false; :1
      }
    }
  } :1 //mandatory by safety

//concat :: BList(1) → BList(1)
void concat(BList other) {
  BList o = this;
  while (o.getQueue() != null) {
    o = o.getQueue();
  }
  o.setQueue(other);
}

// isEqual :: BList(1) → boolean(1)
boolean isEqual(BList other) {
  boolean res = true;
  BList b1 = this;
  BList b2 = other;
  while (b1 != null && b2 != null) {
    if (b1.getValue() != b2.getValue()) {
      res = false;
    }
    b1 = b1.getQueue();
    b2 = b2.getQueue();
  }
  if (b1 != null || b2 != null) {
    res =false;
  }
  return res;
}

//lessOrEqualTo :: BList(1) → boolean(1)
boolean lessOrEqualTo(BList other^1) {
  BList b1^1 = this^1;
  BList b2^1 = other^1;
  boolean res^1 = true;
  while (b1 != null &&
         b2 != null) {
    if (!b1.getValue() &&
         b2.getValue()) {
      res = true;
    } else {;}
    if (b1.getValue() &&
       !b2.getValue()) {
      res = false;
    } else {;}
    if (b1.getQueue() == null &&
        b2.getQueue() != null) {
      res = true;
    } else {;}
    if (b2.getQueue() == null &&
        b1.getQueue() != null) {
      res = false;
    } else {;}
    b1 = b1.getQueue();
    b2 = b2.getQueue();
```

hal-00773215, version 1 - 12 Jan 2013

```
    }
    return res^1;
  }
}

IList {// List of Integers
  BList value^0;
  IList queue^0;

  IList(BList v, IList q) {
    value = v;
    queue = q;
  }

  BList getValue() {
    return value^0;
  }

  IList getQueue() {
    return queue^0;
  }

  void setQueue(IList q^0) {
    queue = q; :0
  }

  // max ::  → BList(1)
  // the object needs to be of tier 1
  BList max() {
    BList currentMax^1 = null;
    IList o^1 = this^1;
    while (o^1 != null) {
      BList v^1 = o.getValue();
      if (currentMax^1.lessOrEqualTo(v^1)) {
        currentMax^1 = v^1;
      }
      o^1 = o.getQueue();
    }
    return currentMax^1;
  }

  // remove :: IList(1)
  void remove(BList element^1) {
    IList o^1 = this^1;
    IList p^1 = null;
    while (!o.getValue().isEqual(element^1)) {
      p^1 = o;
      o^1 = o.getQueue();
    }
    if (p^1 != null) {
      p.setQueue(o.getQueue());
    } else {
      this^1 = o.getQueue();
    }
  }

  /* Selection sort */
  // sort :: → IList(0)
  IList sort() {
    IList o^1 = this;
    IList s^0 = null;
    while (o^1 != null) {
      m^1 = o^1.max();
      s^0 = new IList(m^1, s^0);
      o^1.remove(m^1);
    }
    return s^0;
  }
}

Exe {
  main() {
    // Initialization
    BList i1^1 = new BList(true,
                   new BList(true, null));
    BList i2^1 = new BList(true, null);
    BList i3^1 = new BList(false,
                   new BList(true, null));
    IList l^1 = new IList(i1,
                 new IList(i2,
                 new IList(i3, null)));

    // Computation
    IList s^0 = l^1.sort();
    i3.decrement(); :1
  }
}
```

## Appendix B
## Proofs

**Proposition 1.** *Given an instruction $I$ and a typing variable environment $\Gamma$ such that $\Gamma \vdash I : \mathtt{void}(\alpha)$ holds, there is a typing variable environment $\Gamma'$ such that the following holds:*

- $\forall x \in dom(\Gamma),\ \Gamma(x) = \Gamma'(x)$
- $\Gamma' \vdash \overline{I} : \mathtt{void}(\alpha)$

*Conversely, if $\Gamma' \vdash \overline{I} : \mathtt{void}(\alpha)$, then $\Gamma' \vdash I : \mathtt{void}(\alpha)$.*

*Proof:* By induction on program flattening on instructions. Consider a method call $I = \tau\ \mathtt{x} = E.m(E_1, \dots, E_n)$ such that $\Gamma \vdash I : \mathtt{void}(\alpha)$ and $m \in$ C. This means that $\Gamma \vdash E_i : \tau_i(\alpha_i)$ and $\Gamma \vdash E : \mathrm{C}(\alpha)$ hold, for some $\alpha_i$ and $\alpha$. The flattening of $I$ is of the shape $\overline{J}\ [\tau]\ \mathtt{x} = \mathtt{x}_{n+1}.m(\mathtt{x}_1, \dots, \mathtt{x}_n)$; with $J = \tau_1\ \mathtt{x}_1{:=}E_1; \dots \tau_n\ \mathtt{x}_n{:=}E_n; \tau_{n+1}\ \mathtt{x}_{n+1}{:=}E$;. Now define $\Gamma'$ by:

$$\Gamma'(\mathtt{y}) = \begin{cases} \mathrm{C}(\alpha) \text{ if } \mathtt{y} = \mathtt{x} \\ \tau_i(\alpha_i) \text{ if } \mathtt{y} \in \{\mathtt{x}_1, \dots, \mathtt{x}_n\} \\ \Gamma(\mathtt{y}) \text{ otherwise} \end{cases}$$

We have $\Gamma' \vdash J\ [\tau]\ \mathtt{x} = \mathtt{x}_{n+1}.m(\mathtt{x}_1, \dots, \mathtt{x}_n); : \mathtt{void}(\alpha)$ and $\Gamma' \vdash J : \mathtt{void}(\alpha)$ (sub-typing might be used). By induction

$$\overline{[\tau]\ \mathtt{x}{:=}E;} = [\tau]\ \mathtt{x}{:=}E; \quad \text{if } E \in \mathbb{V} \cup \{\mathtt{this}^{\mathsf{C}}, \mathtt{null}, \mathtt{true}, \mathtt{false}\}$$
$$\overline{[\tau]\ \mathtt{x}{:=}op(E_1,\dots,E_n);} = \overline{\tau_1\ \mathtt{x}_1{:=}E_1;\dots\tau_n\ \mathtt{x}_n{:=}E_n;}\ [\tau]\ \mathtt{x} = op(\mathtt{x}_1,\dots,\mathtt{x}_n);$$
$$\overline{[\tau]\ \mathtt{x}{:=}\mathtt{new}\ \mathsf{C}(E_1,\dots,E_n);} = \overline{\tau_1\ \mathtt{x}_1{:=}E_1;\dots\tau_n\ \mathtt{x}_n{:=}E_n;}\ [\tau]\ \mathtt{x}{:=}\mathtt{new}\ \mathsf{C}(\mathtt{x}_1,\dots,\mathtt{x}_n);$$
$$\overline{[[\tau]\ \mathtt{x}{:=}]E.m(E_1,\dots,E_n);} = \overline{\tau_1\ \mathtt{x}_1{:=}E_1;\dots\tau_n\ \mathtt{x}_n{:=}E_n;\tau_{n+1}\ \mathtt{x}_{n+1} = E;}\ [[\tau]\ \mathtt{x}{:=}]\mathtt{x}_{n+1}.m(\mathtt{x}_1,\dots,\mathtt{x}_n);$$
$$\overline{I_1\ I_2} = \overline{I_1}\ \overline{I_2}$$
$$\overline{\mathtt{while}(E)\{I\}} = \overline{\mathtt{boolean}\ \mathtt{x}_1 := E;}\ \mathtt{while}(\mathtt{x}_1)\{\overline{I}\ \overline{\mathtt{x}_1 := E;}\}$$
$$\overline{\mathtt{if}(E)\{I_1\}\mathtt{else}\{I_2\}} = \overline{\mathtt{boolean}\ \mathtt{x}_1 := E;}\ \mathtt{if}(\mathtt{x}_1)\{\overline{I_1}\}\mathtt{else}\{\overline{I_2}\}$$

All $\mathtt{x}_i$ represent fresh variables and the types $\tau_i$ match the expressions $E_i$ types

Fig. 5: Instruction flattening

hypothesis, there is a variable typing environment $\Gamma''$ such that $\Gamma'' \vdash \overline{J} : \mathtt{void}(\alpha)$ and $\forall \mathtt{x} \in \mathit{dom}(\Gamma'),\ \Gamma''(\mathtt{x}) = \Gamma'(\mathtt{x})$ and, consequently, $\Gamma'' \vdash \overline{I} : \mathtt{void}(\alpha)$. All the other cases are treated similarly. ■

**Lemma 2** (Non-interference)**.** *Given a meta-instruction MI of a safe program with respect to typing variable environment $\Gamma$, let $\mathcal{C}$ and $\mathcal{C}'$ be two memory configurations, if $\mathcal{C}_{\Gamma_\mathbf{1}} = \mathcal{C}'_{\Gamma_\mathbf{1}}$ then $\delta_{\Gamma_\mathbf{1}}(\mathcal{C}, \mathit{MI}) = \delta_{\Gamma_\mathbf{1}}(\mathcal{C}', \mathit{MI})$. In other words, tier $\mathbf{1}$ variables do not depend on tier $\mathbf{0}$ variables.*

*Proof:* First, note that Rule *(Wh)* of Figure 4b and the definition of safe programs enforce all the guards of a safe program (in a while loop and in a recursive call) to be of tier $\mathbf{1}$. Applying Proposition 1, tier $\mathbf{1}$ meta-instructions do not depend on loops controlled by tier $\mathbf{0}$ expressions.
Second, in a `if` meta-instruction of tier $\mathbf{0}$ guard, all the commands are of tier $\mathbf{0}$ by Rule *(If)*. Consequently, no tier $\mathbf{1}$ variable is updated in these commands. Indeed a tier $\mathbf{1}$ variable assignment enforces the containing command to be of tier $\mathbf{1}$ using Rule *(Ass)* and Rule *(Seq)*.
Finally, the rule (Ass) in Figure 4b enforces that tier $\mathbf{1}$ variables of a safe program are only updated by assignments of the shape $\Gamma \vdash \mathtt{x}{:=}E; : \mathtt{void}(\mathbf{1})$. All the variables contained in $E$ are enforced to be of tier $\mathbf{1}$ (except the current object or the parameters in the special case of non recursive methods) by the type system. Consequently, if $\mathcal{C}_{\Gamma_\mathbf{1}} = \mathcal{C}'_{\Gamma_\mathbf{1}}$ and $(\mathcal{C}, \mathtt{x}{:=}E;\ \mathit{MI}') \to (\mathcal{D}, \mathit{MI}')$ then $(\mathcal{C}', \mathtt{x}{:=}E;\ \mathit{MI}') \to (\mathcal{D}', \mathit{MI}')$ and $\mathcal{D}_{\Gamma_\mathbf{1}} = \mathcal{D}'_{\Gamma_\mathbf{1}}$. Now the case of a non recursive method is trivial since its code can be inlined while still being typed under $\Gamma$. And so is the case where the current object variable is of tier $\mathbf{0}$ since the method return variable tier is enforced to be $\mathbf{1}$ by Rules *(Ass)*, *(Call)* and *($M_C$)*. Consequently, there is no information flow from the current object to the return variable in the method body in this particular case. ■

**Lemma 3.** *Given a memory configuration $\mathcal{C}$ and a meta-instruction MI of a safe program with respect to typing variable environment $\Gamma$, if $(\mathcal{C}, \mathit{MI}) \to^+ (\mathcal{C}', \mathit{MI})$ and $\mathcal{C}_{\Gamma_\mathbf{1}} = \mathcal{C}'_{\Gamma_\mathbf{1}}$, then the meta-instruction MI does not terminate on memory configuration $\mathcal{C}$.*

*Proof:* Assume that during the transition $(\mathcal{C}, \mathit{MI}) \to^+ (\mathcal{C}', \mathit{MI})$ there is a $\mathcal{C}''$ such that $\mathcal{C}''_{\Gamma_\mathbf{1}} \neq \mathcal{C}_{\Gamma_\mathbf{1}}$, then the distinct tier $\mathbf{1}$ configuration sequence $\delta_{\Gamma_\mathbf{1}}(\mathcal{C}, \mathit{MI})$ contains this $\mathcal{C}''_{\Gamma_\mathbf{1}}$ before $\mathcal{C}'_{\Gamma_\mathbf{1}}$. From the construction of the sequence, we deduce that $\delta_{\Gamma_\mathbf{1}}(\mathcal{C}, \mathit{MI})$ is of the shape $\dots\mathcal{C}''_{\Gamma_\mathbf{1}}\dots\delta_{\Gamma_\mathbf{1}}(\mathcal{C}', \mathit{MI})$. However from Lemma 2, $\delta_{\Gamma_\mathbf{1}}(\mathcal{C}, \mathit{MI}) = \delta_{\Gamma_\mathbf{1}}(\mathcal{C}', \mathit{MI})$, hence it is infinite and the meta-instruction $\mathit{MI}$ does not terminate on memory configuration $\mathcal{C}$.

Otherwise, we are in a state $(\mathcal{C}, \mathit{MI})$ from which the set of variables of tier $\mathbf{1}$ will never change. If $(\mathcal{C}, \mathit{MI}) \to^+ (\mathcal{C}', \mathit{MI})$ then this means that the meta-instruction $\mathit{MI}$ contains either a while loop or a recursive call (otherwise the meta-instruction $\mathit{MI}$ cannot be the same). Since while loops and recursive call parameters are of tier $\mathbf{1}$, by definition of safe programs, this means that they remain unchanged and consequently the meta-instruction $\mathit{MI}$ does not terminate on $\mathcal{C}$. ■

**Lemma 4.** *Given an input $\mathcal{C}$ and a meta-instruction MI of a safe program with respect to variable typing environment $\Gamma$, the following holds:*

$$\#\left\{\mathcal{C}'_{\Gamma_1} \mid (\mathcal{C}, \mathit{MI}) \to^* (\mathcal{C}', \mathit{MI}')\right\} \leq |\mathcal{C}|^{\#\mathit{dom}(\Gamma_1)}.$$

*Proof:* By Lemma 2, there is no information flow from tier $\mathbf{0}$ to tier $\mathbf{1}$. Moreover, Rule *(New)* of Figure 4a enforces that tier $\mathbf{1}$ expressions cannot correspond to the creation of a new instance. Indeed in an assignment of the shape $\mathtt{x}{:=}\mathtt{new}\ \mathsf{C}(\mathtt{y}_1,\dots,\mathtt{y}_n)$, the judgment $\Gamma \vdash \mathtt{new}\ \mathsf{C}(\mathtt{y}_1,\dots,\mathtt{y}_n) : \mathsf{C}(\mathbf{0})$ holds and, consequently, the judgment $\Gamma \vdash \mathtt{x} : \mathsf{C}(\mathbf{1})$ cannot hold because of Rule *(Ass)*. Consequently, variables of tiered type $\mathsf{C}(\mathbf{1})$ may only point to nodes of the initial pointer graph corresponding to input $\mathcal{C}$. The number of such nodes is bounded by $|\mathcal{C}|$. A boolean variable $\mathtt{x}$ of tier $\mathbf{1}$ has only two possible distinct values and clearly $2 \leq |\mathcal{C}|$, since the graph of $\mathcal{C}$ has at least one node (the `null` reference) and $\mathtt{x}$ is in the domain of the primitive store of $\mathcal{C}$. The number of tier $\mathbf{1}$ variables being equal to $\#\mathit{dom}(\Gamma_\mathbf{1})$, by definition of $\Gamma_\mathbf{1}$, the number of distinct configurations is bounded by $|\mathcal{C}|^{\#\mathit{dom}(\Gamma_\mathbf{1})}$. ■

It follows from Lemma 4 that the while loops of a safe and terminating program are polynomial time instructions.

**Lemma 5.** *Given a meta-instruction MI of a safe program*

hal-00773215, version 1 - 12 Jan 2013

*with respect to variable typing environment* $\Gamma$ *such that* $MI$ *terminates on input* $\mathcal{C}$*. Each* `while` *loop in* $MI$ *can be executed at most* $|\mathcal{C}|^{\#dom(\Gamma_1)\times\nu(MI)}$ *times.*

*Proof:* By induction on the intricacy. First, notice that a while loop meta-instruction has an intricacy strictly greater than 0. Now consider a meta-instruction of the shape $MI$ such that $\nu(MI) = 1$ then clearly $MI = MI_1$ $\texttt{while}(\texttt{x})\{MI'\}$ $MI_2$, for some meta-instructions $MI_i$ and $MI'$ such that $\nu(MI_i) \leq 1$ and $\nu(MI') = 0$, since the while loop cannot be nested. From the semantics, we trivially infer that either the guard x is evaluated to false and we will never encounter this `while` statement again, either it evaluates to `true` and the next time the while is encountered, the meta-instruction will be the same. From Lemma 3, we know that if we encounter a meta-instruction twice with configurations that match on tier **1** variables, the meta-instruction does not terminate on said configuration. That means that the while loop can only be executed once per distinct tier **1** configuration, which is $|\mathcal{C}|^{\#dom(\Gamma_1)}$, by Lemma 4. Now suppose that it holds for a meta-instruction of intricacy $k$ and consider a meta-instruction $MI$ of intricacy $k+1$, then clearly $MI = MI_1$ $\texttt{while}(\texttt{x})\{MI'\}$ $MI_2$ with $\nu(MI) \geq \nu(MI') + 1$. Using the same argument than for the base case, this meta-instruction can be transformed into the following equivalent meta-instruction $MI_1 \underbrace{MI' \ldots MI'}_{k \text{ times}} MI_2$ for some $k$ such that $k \leq |\mathcal{C}|^{\#dom(\Gamma_1)}$. By induction hypothesis, any while loop within $MI'$ can be executed at most $|\mathcal{C}|^{\#dom(\Gamma_1)\times\nu(MI')}$ and, consequently, it can be executed at most $k \times |\mathcal{C}|^{\#dom(\Gamma_1)\times\nu(MI')} \leq |\mathcal{C}|^{\#dom(\Gamma_1)\times(\nu(MI')+1)} \leq |\mathcal{C}|^{\#dom(\Gamma_1)\times\nu(MI)}$ in $MI$. If $MI_1$ or $MI_2$ also has intricacy $k+1$, the same argument can be applied. ■

Now we show a bound similar to the bound of Lemma 5 on method calls wrt the level:

**Lemma 6.** *Given a meta-instruction* $MI = [[\tau]x{:=}]y.m(y_1, \ldots, y_n);$ *of a safe program with respect to variable typing environment* $\Gamma$*. If* $(\mathcal{C}, MI) \to^k (\mathcal{C}', \epsilon)$ *(i.e.* $MI$ *terminates on input* $\mathcal{C}$*) then* $k = O(|\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times\lambda(m))})$*.*

*Proof:* By induction on the level. Consider a method of the shape $\tau\ m(\ldots)\{MI'\ [\texttt{return z;}]\}$.

If $\lambda(m) = 1$. By definition of the level, this means $m \notin [m]$, i.e. the method $m$ is not recursive. Hence, by Lemma 5, each meta-instruction can be executed at most $|\mathcal{C}|^{\#dom(\Gamma_1)\times(\nu(MI')+1)}$ (The constant 1 comes from the fact that an instruction is executed at least once even if it is not located within a while statement). Consequently, there are at most $|MI'| \times |\mathcal{C}|^{\#dom(\Gamma_1)\times(\nu(MI')+1)}$ executed meta-instruction before the program terminates. i.e. $k = O(|\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times 1)})$.

Assume $\lambda(m) = i + 1$. Either m is recursive, this means $m \in [m]$. Hence, by safety assumption and by Lemma 4, we know that there are at most $|\mathcal{C}|^{\#dom(\Gamma_1)}$ nested recursive calls to $m$ in the evaluation of $MI$ since all the arguments are of tier **1**, there is at most one recursive call in the method body (and no while loop) and the meta-instruction terminates. Consequently, the number of meta-instructions unfolded by a method call on $m$ is at most $|MI'| \times |\mathcal{C}|^{\#dom(\Gamma_1)}$. Consequently, the number of meta-instructions unfolded by method calls of $[m]$ is at most $O(|\mathcal{C}|^{\#dom(\Gamma_1)})$ (indeed just take the finite sum of $|MI'| \times |\mathcal{C}|^{\#dom(\Gamma_1)}$, for each method of the equivalence class). In the worst case, all the other meta-instructions correspond to method calls of level $i$. Applying the induction hypothesis, we know each of these calls will generate at most $O(|\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times i)})$. Putting all together, it generates at most $O(|\mathcal{C}|^{\#dom(\Gamma_1)} \times |\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times i)}) = O(|\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times\lambda(m))})$ meta-instructions.

Or $m$ is not recursive, this means that it may contain `while` meta-instructions. In the worst case, the calls to methods of level $i$ can be in a `while` that is nested $\nu$ times. This means that the $|MI'|$ meta-instructions of level $i$ can be unfolded $|\mathcal{C}|^{\#dom(\Gamma_1)\nu}$ times from Lemma 4. Since by induction hypothesis, each can yield $O(|\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times i)})$ meta-instructions, it finally gives $O\left(|\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times(i+1))}\right)$ meta-instructions. ■

**Theorem 1.** *If a core Java program of computational instruction* $I$ *is safe wrt to variable typing environment* $\Gamma$ *and terminates on input* $\mathcal{C}$ *then for each memory configuration* $\mathcal{C}'$ *and meta-instruction* $MI$ *s.t.* $(\mathcal{C}, \overline{I}) \to^* (\mathcal{C}', MI)$ *we have*

$$|\mathcal{C}'| = O(|\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times\lambda)}).$$

*In other words, if* $\mathcal{C}' = \langle \mathcal{G}, \mathcal{P}, \mathcal{S}, \sigma \rangle$ *then both* $|\mathcal{G}_\mathcal{P}|$ *and* $|\mathcal{S}_\mathcal{G}|$ *are in* $O(|\mathcal{C}|^{\#dom(\Gamma_1)\times((\nu+1)\times\lambda)})$*.*

*Proof:* Proposition 1 guarantees that types remain stable under flattening. Safety is also preserved by Remark 1. Moreover, the size of a flattened program remains linear in the size of the initial program, by Lemma 1. Consequently, we can consider the flattened program instead of the initial program. The heap-space upper bound is a consequence of Lemmata 4 and 6 that bound the number of assignments executed in a terminating and safe program. Since each assignment creates a bounded number of new nodes in the graph, we obtain the requested upper bound. The stack upper bound is a direct consequence of Lemma 6 since the maximal size of the stack is bounded by the number of executed `push` instructions, also bounded by the number of executed instructions (i.e. the reduction depth). ■

**Proposition 2** (Type inference)**.** *Deciding if there exists a variable typing environment* $\Gamma$ *such that typing rules are satisfied can be done in time linear in the size of the program.*

*Proof:* Types can be checked in linear time in the size of the program as typing mainly consists in checking type annotations with respect to method signatures, operator signatures and attributes declarations.
For tiers, we encode the tier of each variable x by a boolean variable $x$ that will be true if the variable is of tier **1**, false if it is of tier **0**. Each instruction generates some constraints. For example, in the case of an assignment $\texttt{x} := \texttt{y}$, we have to check $\pi_2(\Gamma(\texttt{x})) \preceq \pi_2(\Gamma(\texttt{y}))$, which can be represented as

hal-00773215, version 1 - 12 Jan 2013

$(y \vee \neg x)$. Verifying all such constraints generates a conjonction of such clauses which are in number linear in the size of the program. As a result, the type inference problem is reduced to 2-SAT and can be solved in linear time. ■

## APPENDIX C
## FLATTENING

The formal description of how to flatten an instruction is given in Figure 5.

hal-00773215, version 1 - 12 Jan 2013