Type-based heap and stack space analysis in Java

Emmanuel Hainry, Romain Péchoux

Type-based heap and stack space analysis in Java

Emmanuel Hainry* Romain Péchoux*

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. The 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 its use could allow the programmer to avoid memory errors. Second, type checking is decidable in linear time. Last but not least, it has a good expressivity since it analyzes most object oriented features like overload, inheritance, override and can also handle statements that alter the control flow like break or return. In particular, it improves previous analyses on the complexity of OO and imperative programs since it can deal with loops guarded by objects (list, trees, cyclic data).

1 Introduction

Motivations. 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 issue 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; features like recursion, while loops, inheritance, override, overload are handled by the presented analysis. Core Java can be seen as a language strictly more expressive than Featherweight Java [18] enriched with features like variable updates and while loops.

Contribution. The presented type system combines ideas coming from tiering discipline, used for complexity analysis of function algebra [3, 22], together with ideas coming from non-interference, used for secure information flow analysis [27]. It is inspired by two previous works: the seminal paper [23], initiating imperative programs type-based complexity analysis using secure information flow, which provides a characterization of polynomial time computable

*Université de Lorraine, Loria, UMR 7503, Nancy, France

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 the current work improves the following points:

- first, it is a non trivial extension to the object-oriented paradigm (although imperative features can be dealt with) allowing to study both the complexity of object instantiation (using the `new` operator) and of loops guarded by objects. The difficulty in such a study is that the underlying structure is no more a string or a stack but a graph of memory references. Moreover, our work characterizes the complexity of recursive and non-recursive method calls whereas previous works were restricted to while loops,

- second, it studies both program extensional and intensional properties (like heap and stack) whereas previous papers were focusing on the extensional part (characterizing function spaces). Consequently, it is a wider study that is closer to a programmer's expectations,

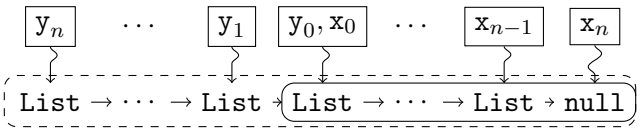
- third, it provides explicit big O polynomial upper bounds while the two aforementioned studies were only certifying algorithms to compute a function belonging to some fixed complexity class.

Intuition. In the introduced type system, the heap is represented by a directed graph where nodes are object addresses and arrows relate an object address to its field addresses. The type system splits variables into two universes: tier **0** universe and tier **1** universe. While tier **1** variables are pointers to nodes of the initial heap, tier **0** variables may point to newly created addresses. The information may flow from tier **1** to tier **0**, that is a tier **0** variable may depend on tier **1** variables. However our type system precludes flows from **0** to **1**. Indeed once a variable has stored a newly created instance, it can only be of tier **0**. Tier **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 **0** variables are just used as a storage for computed data. The polynomial upper bound is obtained as follows: if the input graph structure has size n then the number of distinct possible configurations for k tier **1** variables is at most $O(n^k)$.

Motivating example. Consider the following Java code doubling the length of a boolean List as an illustrating example:

```
y=x;
while (x!=null){
    y=new List(true,y);
    x=x.getTail();
}
```

The tier of variable `x` will be enforced to be **1** since it is used in a while loop guard. On the opposite, the tier of variable `y` will be enforced to be **0** since the `y=new List(true,y);` instruction enlarges its memory use. For each assignment, we will check that the tier of the variable assigned to is smaller than the tier of the assigned expression. Consequently, the assignment `y=x;` is typable in this code (since **0** is smaller than **1** ; a flow from tier **1** to **0** being admissible) whereas `x=y;` or `x=new List(true,x);` are not typable wrt to this code as they might lead to unbounded computations. As explained above, it means that `x` never points to newly created addresses while `y` may. This typing discipline is illustrated by the following figure:



$\mathtt{x}_i$ and $\mathtt{y}_i$ represent the object referred to by x and, respectively, y just before the i+1-th iteration of the while loop. The dashed rectangle represents the tier **0** universe whereas the plain rectangle represents tier **1** universe. As explained above, tier **0** variables may grow and can also access to the tier **1** memory, but not the contrary. This is the reason why $\mathtt{y}_0$ can point to the tier **1** memory at the beginning of the execution. To conclude, the loop guard in this small example could also been adapted to include circular structures like circular buffers and can be handled provided that the size of the guard is left unchanged. It is worth noticing that the studies mentioned below in the related works cannot deal with such programs.

Related works. On imperative languages, the papers [25, 24, 19] study theoretically the heap-space complexity of core-languages using type systems based on a matrix 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 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 recursion being a very hard task from a complexity perspective.

Another interesting line of research is based on the analysis of heap-space and time consumption of Java bytecode [1, 2, 21, 7]. The results from [1, 2, 21] 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. Moreover, our approach handles very elegantly while loops guarded by a variable of reference type whereas most of the aforementioned studies are based on invariants generation for primitive types only. 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.

In a similar vein, characterizing complexity classes below polynomial time is studied in [16, 17]. This work relies on a 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 this work.

The presented work is independent from termination analysis but our main result relies on such analysis. Indeed, the polynomial upper bounds on both the stack and the heap space consumption of a typed program provided by Theorem 1 only hold 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.

Outline. In Section 2, we introduce the syntax of core Java and the notion of well-formed

program. Section 3 describes the semantics of core Java based on graph structures called pointer graphs. In Section 4, the type system, which is the main contribution of the paper, is presented and explained. A full example is provided in Section 5 to explain the subtlety of the typing rules. Section 6 is devoted to present our main result, Theorem 1, direct corollaries as well as a decidability result on type inference. Finally, we show direct extensions in Section 7 to inheritance and to statements that break program control flow (break, return,...).

2 Core Java syntax

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

2.1 Syntax of classes

Expressions, instructions, constructors, methods and classes are defined by the grammar of Figure 1, with $\mathtt{x} \in \mathbb{V}$, $op \in \mathbb{O}$, $\mathrm{C} \in \mathbb{C}$, $\mathtt{m} \in \mathbb{M}$, $\mathbb{V}$ being the set of variables, $\mathbb{O}$ the set of operators,

---

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

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

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

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

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

Figure 1: Syntax of core Java

$\mathbb{M}$ the set of method names and $\mathbb{C}$ the set of class names. The τs 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 ; denotes 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 statement $\tau\ \mathtt{x}{:=}E;\ \mathtt{while}(\mathtt{condition})\ \{\mathtt{Ins}\ \mathtt{Increment};\}$. Also notice that there is no field access in our syntax using the "." operator. Consequently, all fields are implicitly `private`. In contrast, methods and classes are all `public`. This is not a huge restriction for a Java programmer since any field can be accessed and updated in an outter class by writing the corresponding getter and setter.

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

$$Exe\{\mathtt{main}()\{\tau_1\ \mathtt{x}_1 := E_1; \ldots; \tau_n\ \mathtt{x}_n := E_n; I\}\}$$

In an executable, the instruction $\tau_1\ \mathtt{x}_1 := E_1; \ldots; \tau_n\ \mathtt{x}_n := E_n;$ *is called the* initialization instruction *whereas* I *is called the* computational instruction.

In a class $\mathfrak{C} = \mathrm{C}\{\tau_1\ \mathtt{x}_1; \ldots; \tau_n\ \mathtt{x}_n;\ K_\mathrm{C}\ M^1_\mathrm{C} \ldots M^k_\mathrm{C}\}$, the $\mathtt{x}_i$s are called fields. Moreover let $\mathrm{C}.\mathcal{A}$ denote the set of the fields 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. Finally, each variable declared in an assignment of the shape $\tau\ \mathtt{x}{:=}E;$ is called a local variable. Given a method $\tau\ \mathtt{m}(\tau_1\ \mathtt{x}_1, \ldots, \tau_n\ \mathtt{x}_n)\{I\ [\mathtt{return}\ \mathtt{x};]\}$ of C, we say that its signature is $\tau\ \mathtt{m}^\mathrm{C}(\tau_1, \ldots, \tau_n)$; the notation $\mathtt{m}^\mathrm{C}$ denoting that $\mathtt{m}$ is declared in C.

For readability, we assume classes have exactly one constructor initializing all the class fields. Also, the only considered primitive data are boolean values `true` and `false`. Other primitive data types such as floats, integers and characters could be considered, as explained in Subsection 4.4.1. In the presented syntax, there is no inheritance and, consequently, no overrides even if those features can be handled by our type system. For simplicity and readability, their treatment will be delayed to Section 7. Notice however that overload is possible in the initial core Java syntax.

2.2 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, a field or a parameter. For simplicity, we assume that programs are statically transformed up-to α-conversion so that each variable (local variable, field or parameter) has a distinct name, hence preventing name clashes.
- Each local variable $\mathtt{x}$ is both declared and initialized exactly once by a $\tau\ \mathtt{x} := E;$ instruction before its first use.
- A method output type is `void` iff it has no `return` statement.
- Each method signature is unique.

3 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.

3.1 Pointer graph

Definition 2. *A* pointer graph $\mathcal{G}_\mathcal{P}$ *is a directed multigraph* $\mathcal{G} = (V, A)$ *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 fields and are labeled by the field 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 $\mathcal{P} : \mathbb{V} \cup \{\texttt{this}\} \mapsto V$ *associates a node of the graph in* V *to some variables in* $\mathbb{V}$ *or to the current object* `this` *and is called a* pointer mapping. *Let* $dom(\mathcal{P})$ *be the domain of* $\mathcal{P}$.

The memory used by a core Java program will be represented by a pointer graph. Each constructor call will create a new node of the graph and arrows to its fields. It also respects the dynamic binding principle found in Object Oriented Languages. Those arrows are annotated by the field name. The semantics of an assignment $\texttt{x} := E$; consists in updating the pointer mapping in such a way that $\mathcal{P}(\texttt{x})$ 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);
```

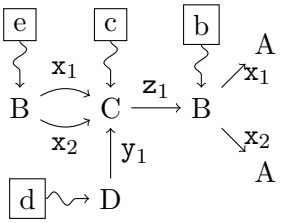


Figure 2: Example of a pointer graph

3.2 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* $\mathcal{S}$ *of pointer mappings corresponding to the same directed graph* $\mathcal{G}$. *Given a pointer stack* $\mathcal{S}_\mathcal{G}$, *define* $\top\mathcal{S}$ *to be the topmost 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 $\tau\ \texttt{m}(\tau_1\ \texttt{y})\{J; \texttt{return z}\}$ in a method call $\texttt{x} := E.\texttt{m}(F)$; will push a new pointer mapping $\mathcal{P}$ on pointer stack $\mathcal{S}_\mathcal{G}$ such that $\mathcal{P}(\texttt{y})$ 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 mapping from the pointer stack will correspond, as expected, to the evaluation of a return statement in a method body.

$$\overline{[\tau]\ \mathtt{x}{:=}E;} = [\tau]\ \mathtt{x}{:=}E; \quad \text{if } E \in \mathbb{V} \cup \{\mathtt{this}, \mathtt{null}, \mathtt{true}, \mathtt{false}\}$$

$$\overline{[\tau]\ \mathtt{x}{:=}op(E_1, \ldots, E_n);} = \overline{\tau_1\ \mathtt{x}_1{:=}E_1; \ldots \tau_n\ \mathtt{x}_n{:=}E_n;}\ [\tau]\ \mathtt{x} = op(\mathtt{x}_1, \ldots, \mathtt{x}_n);$$

$$\overline{[\tau]\ \mathtt{x}{:=}\mathtt{new}\ \mathrm{C}(E_1, \ldots, E_n);} = \overline{\tau_1\ \mathtt{x}_1{:=}E_1; \ldots \tau_n\ \mathtt{x}_n{:=}E_n;}\ [\tau]\ \mathtt{x}{:=}\mathtt{new}\ \mathrm{C}(\mathtt{x}_1, \ldots, \mathtt{x}_n);$$

$$\overline{[[\tau]\ \mathtt{x}{:=}]E.\mathtt{m}(E_1, \ldots, E_n);} = \overline{\tau_1\ \mathtt{x}_1{:=}E_1; \ldots \tau_n\ \mathtt{x}_n{:=}E_n; \tau_{n+1}\ \mathtt{x}_{n+1} = E;}$$
$$[[\tau]\ \mathtt{x}{:=}]\mathtt{x}_{n+1}.\mathtt{m}(\mathtt{x}_1, \ldots, \mathtt{x}_n);$$

$$\overline{I_1\ I_2} = \overline{I_1}\ \overline{I_2}$$

$$\overline{\mathtt{while}(E)\{I\}} = \overline{\mathtt{boolean}\ \mathtt{x}_1 := E;}\ \mathtt{while}(\mathtt{x}_1)\{\overline{I}\ \overline{\mathtt{x}_1 := E;}\}$$

$$\overline{\mathtt{if}(E)\{I_1\}\mathtt{else}\{I_2\}} = \overline{\mathtt{boolean}\ \mathtt{x}_1 := E;}\ \mathtt{if}(\mathtt{x}_1)\{\overline{I_1}\}\mathtt{else}\{\overline{I_2}\}$$

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

Figure 3: Instruction flattening

3.3 Memory configuration

A *primitive store* σ is a partial mapping $\sigma : \mathbb{V} \mapsto \{\mathtt{true}, \mathtt{false}\}$ associating a boolean value to some variable of primitive data type in $\mathbb{V}$. As usual, the domain of a primitive store σ 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* σ *is a primitive store.*
Among memory configurations, the initial configuration $\mathcal{C}_0$ *defined by* $\mathcal{C}_0 = \langle (\{\&\mathtt{null}\}, \emptyset), \emptyset, [], \emptyset \rangle$ *where the notation* $\emptyset$ *is used both for empty set and empty mapping,* $[]$ *denotes the empty pointer stack, and* $\&\mathtt{null}$ *is the reference of the* `null` *object (i.e.* $l(\&\mathtt{null}) = \mathtt{null}$*).*

3.4 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 flattened instructions and `pop` and `push` operations for managing method calls. Meta-expressions and meta-instructions are defined formally by the following grammar:

$$\begin{aligned} ME \quad ::= \quad & \mathtt{x} \mid \mathtt{null} \mid \mathtt{this} \mid \mathtt{true} \mid \mathtt{false} \mid op(\mathtt{x}_1, \ldots, \mathtt{x}_n) \\ & \mid \mathtt{new}\ \mathrm{C}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \mid \mathtt{y}.\mathtt{m}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \end{aligned}$$

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

where ϵ 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 3. 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,

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

Figure 4: Semantics of core Java

hence allowing a cleaner semantic treatment of meta-instructions. We extend the flattening to methods and procedures by $\tau\ \mathtt{m}(\tau_1\ \mathtt{x}_1,\ldots,\tau_n\ \mathtt{x}_n)\{\overline{I}\ [\mathtt{return}\ \mathtt{x};]\}$ 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 $|\overline{I}| = O(|I|)$.*

Proof. By induction on the definition of flattening. □

3.5 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. In the special case where $(\mathcal{C}, MI) \to^* (\mathcal{C}', \epsilon)$, we say that *$MI$ terminates on memory configuration $\mathcal{C}$.*

Definition 5. *A core Java program of executable*

$$Exe\{\mathtt{main}()\{\tau_1\ \mathtt{x}_1 := E_1;\ldots;\tau_n\ \mathtt{x}_n := E_n; I\}\}$$

terminates if the following conditions hold:

1. $(\mathcal{C}_0, \overline{\tau_1\ \mathtt{x}_1 := E_1; \ldots; \tau_n\ \mathtt{x}_n := E_n;}) \rightarrow^* (\mathcal{C}, \epsilon)$
2. $(\mathcal{C}, \overline{I}) \rightarrow^* (\mathcal{C}', \epsilon)$

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 $\mathcal{C}(\mathtt{x})$ be defined by:

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

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

The rules of $\rightarrow$ are defined formally in Figure 4.

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 &`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 σ. 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. $\top\mathcal{S}(\mathtt{this})$). 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`. Rules (5,7,9) are just structural rules introducing fresh local variables $\mathtt{x}_i$s in order to evaluate parameters in a operator evaluation, object creation or method call. Consequently, they need to enforce that the $\mathtt{x}_i$s are fresh variables, i.e. $\mathtt{x}_i \notin dom(\mathcal{C})$. 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 field name in the class C. Finally, this rule adds a link from the variable $\mathtt{x}$ 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.

4 Type system

4.1 Tiered types

The set of base types $\mathtt{T}$ is defined to be the set including a reference type C for each class name C, 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 $\alpha, \beta, \ldots$ denote tiers in $\{\mathbf{0}, \mathbf{1}\}$. Let the minimum $\bigwedge_{i \in S} \alpha_i$ of a finite set of tiers $\{\alpha_i \mid i \in S\}$, indexed by the finite set S, be defined in a standard way (setting $\bigwedge_{i \in \emptyset} \alpha_i = \mathbf{1}$).

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, the two projections π_1 and π_2 are defined by $\pi_1(\tau(\alpha)) = \tau$ and $\pi_2(\tau(\alpha)) = \alpha$.

Intuitively, tier $\mathbf{0}$ will be used to type variables whose 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 an argument of a recursive method call. The values stored in a tier $\mathbf{1}$ variable will not be allowed to increase during a computation.

4.2 Environments

A *field typing environment* $\delta_{m^{\mathrm{C}}}$ for a method m of class C maps each field $v \in \mathrm{C}.\mathcal{A}$ to a tiered type.

A *local variable and parameter typing environment* δ maps each non-field variable in $\mathbb{V}$ to a tiered type.

A *typing environment* Δ is a list containing the unions $\delta_{m^{\mathrm{C}}} \biguplus \delta$, for each field typing environment $\delta_{m^{\mathrm{C}}}$ and for the local variable and parameter typing environment δ. Let $\Delta(m^{\mathrm{C}})$ denote $\delta_{m^{\mathrm{C}}} \biguplus \delta$.

The main reason for defining typing environments this way is to allow the programmer to type a field with distinct tiers depending on the considered method. Indeed, a field can be used as a guard of a while loop (and will be of tier $\mathbf{1}$) in some method whereas it can store freshly created objects (and will be of tier $\mathbf{0}$) in some other method. In order to simplify our type system, one could consider all the field typing environments to be the same but this would impact negatively the expressivity of our type system.

Since a field can have distinct tiered types depending on the considered method, our type system has to keep information on the context (i.e. the method) while typing an expression or an instruction. In order to overcome this problem, we introduce the notion of contextual typing environment.

A *contextual typing environment* $\Gamma = (m^{\mathrm{C}}, \Delta)$ is a pair consisting in a method and a typing environment. The method $\mathtt{m}^{\mathrm{C}}$ in the contextual typing environment $(\mathtt{m}^{\mathrm{C}}, \Delta)$ indicates under which context the fields should be typed, in other words which field typing environment is considered. We will write (ϵ, Δ) when the context is empty. We assume that $\Delta(\epsilon)$ represents the local variable and parameter typing environment δ.

4.3 Well-typed programs

4.3.1 Operator signature

The language is restricted to operators whose return type is `boolean`[1]. An operator of arity n comes equipped with a signature of the shape $\tau_1 \times \cdots \times \tau_n \to \texttt{boolean}$, fixed by the language implementation. In the type system, the notation $op :: \tau_1 \times \cdots \times \tau_n \to \texttt{boolean}$ denotes that op has signature[2] $\tau_1 \times \cdots \times \tau_n \to \texttt{boolean}$.

4.3.2 Judgments

Expressions and instructions will be typed using tiered types whereas constructors and methods of arity n have types of the shape: $\tau_1(\alpha_1) \times \ldots \times \tau_n(\alpha_n) \to \tau(\alpha)$ and $\mathrm{C}(\beta) \times \tau_1(\alpha_1) \times \ldots \times \tau_n(\alpha_n) \to \tau(\alpha)$, respectively. Given a contextual typing environment Γ, 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. The type is enforced to be `void`, meaning that instructions have no return value [3].
- 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 $\Gamma \vdash M_{\mathrm{C}} : \mathrm{C}(\beta) \times \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha)$ for methods is similar but with unrestricted tiers. The first tiered type $\mathrm{C}(\beta)$ is the tiered type of the current object `this` and, consequently, its base type C is enforced to match the class of the considered method.

4.3.3 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 contextual typing environment* $\Gamma = (\epsilon, \Delta)$*, the judgment* $\Gamma \vdash Exe : \diamond$ *means that the program is well-typed wrt* Γ *and is defined in rule (Main) of Figure 5c.*

[1] It could be extended to other primitive data types as explained in Subsection 4.4.1.

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

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

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

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

$$\frac{\forall i\ (\mathtt{m}^{\mathrm{C}}, \Delta) \vdash E_i : \tau_i(\beta_i) \quad \alpha_i \preceq \beta_i \quad (\epsilon, \Delta) \vdash \mathrm{C}(\ldots \tau_i\ \mathtt{y}_i \ldots)\{\ldots \mathtt{x}_i := \mathtt{y}_i; \ldots\} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \mathrm{C}(\mathbf{0})}{(\mathtt{m}^{\mathrm{C}}, \Delta) \vdash \texttt{new}\ \mathrm{C}(E_1, \ldots, E_n) : \mathrm{C}(\mathbf{0})}\ \textit{(New)}$$

$$\frac{\forall i\ (\mathtt{n}^{\mathrm{D}}, \Delta) \vdash E_i : \tau_i(\beta_i) \quad \alpha_i \preceq \beta_i \quad (\epsilon, \Delta) \vdash \mathtt{m} : \mathrm{C}(\beta) \times \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha) \quad (\mathtt{n}^{\mathrm{D}}, \Delta) \vdash E : \mathrm{C}(\beta)}{(\mathtt{n}^{\mathrm{D}}, \Delta) \vdash E.\mathtt{m}(E_1, \ldots, E_n) : \tau(\alpha)}\ \textit{(Call)}$$

(a) Expressions

$$\frac{\Gamma \vdash \mathtt{x} : \tau(\alpha) \qquad \Gamma \vdash E : \tau(\beta) \quad \alpha \preceq \beta}{\Gamma \vdash [\tau]\ \mathtt{x}{:=}E; : \texttt{void}(\alpha)}\ \textit{(Ass)} \qquad \frac{\Gamma \vdash I : \texttt{void}(\alpha) \quad \alpha \preceq \beta}{\Gamma \vdash I : \texttt{void}(\beta)}\ \textit{(Sub)} \qquad \frac{\forall i,\ \Gamma \vdash I_i : \texttt{void}(\alpha_i)}{\Gamma \vdash I_1\ I_2 : \texttt{void}(\alpha_1 \vee \alpha_2)}\ \textit{(Seq)}$$

$$\frac{\Gamma \vdash E : \texttt{boolean}(\mathbf{1}) \quad \Gamma \vdash I : \texttt{void}(\mathbf{1})}{\Gamma \vdash \texttt{while}(E)\{I\} : \texttt{void}(\mathbf{1})}\ \textit{(Wh)} \qquad \frac{}{\Gamma \vdash\ ; : \texttt{void}(\mathbf{0})}\ \textit{(Skip)} \qquad \frac{\Gamma \vdash E : \texttt{boolean}(\alpha) \quad \forall i,\ \Gamma \vdash I_i : \texttt{void}(\alpha)}{\Gamma \vdash \texttt{if}(E)\{I_1\}\texttt{else}\{I_2\} : \texttt{void}(\alpha)}\ \textit{(If)}$$

(b) Instructions

$$\frac{\forall i,\ (\epsilon, \Delta) \vdash \mathtt{y}_i : \tau_i(\alpha_i)}{(\epsilon, \Delta) \vdash \mathrm{C}(\tau_1\ \mathtt{y}_1, \ldots, \tau_n\ \mathtt{y}_n)\{\mathtt{x}_1 := \mathtt{y}_1; \ldots \mathtt{x}_n := \mathtt{y}_n\} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \mathrm{C}(\mathbf{0})}\ \textit{(K}_C\textit{)}$$

$$\frac{(\mathtt{m}^{\mathrm{C}}, \Delta) \vdash \texttt{this} : \mathrm{C}(\beta) \qquad \forall i,\ (\mathtt{m}^{\mathrm{C}}, \Delta) \vdash \mathtt{x}_i : \tau_i(\alpha_i) \qquad (\mathtt{m}^{\mathrm{C}}, \Delta) \vdash \mathtt{x} : \tau(\alpha) \qquad (\mathtt{m}^{\mathrm{C}}, \Delta) \vdash I : \texttt{void}(\alpha)}{(\epsilon, \Delta) \vdash \tau\ \mathtt{m}(\tau_1\ \mathtt{x}_1, \ldots, \tau_n\ \mathtt{x}_n)\{I\ \texttt{return}\ \mathtt{x};\} : \mathrm{C}(\beta) \times \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha)}\ \textit{(M}_C\textit{)}$$

$$\frac{(\mathtt{m}^{\mathrm{C}}, \Delta) \vdash \texttt{this} : \mathrm{C}(\beta) \qquad \forall i,\ (\mathtt{m}^{\mathrm{C}}, \Delta) \vdash \mathtt{x}_i : \tau_i(\alpha_i) \qquad (\mathtt{m}^{\mathrm{C}}, \Delta) \vdash I : \texttt{void}(\alpha)}{(\epsilon, \Delta) \vdash \texttt{void}\ \mathtt{m}(\tau_1\ \mathtt{x}_1, \ldots, \tau_n\ \mathtt{x}_n)\{I\} : \mathrm{C}(\beta) \times \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \texttt{void}(\alpha)}\ \textit{(M}_C^{\texttt{void}}\textit{)}$$

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

(c) Constructors, methods and executable

Figure 5: Type system for core Java

4.4 Typing rules

4.4.1 Expressions

The typing rules for expressions are provided in Figure 5a.

Rules *(True)* and *(False)* mean that boolean constants are of type `boolean` and tier **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 **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 **1** as it cannot increase.

The *(Self)* rule explicits that the self reference `this` belongs to class C (the class of the context) and has a tier lower than the minimum of the fields' tiers under the typing environment $\Delta(\mathtt{m}^{\mathrm{C}})$, denoted by $\bigwedge \Delta(\mathtt{m}^{\mathrm{C}})$ and defined formally by:

$$\bigwedge \Delta(\mathtt{m}^{\mathrm{C}}) = \bigwedge_{\mathtt{x} \in \mathrm{C}.\mathcal{A}} (\pi_2(\Delta(\mathtt{m}^{\mathrm{C}})(\mathtt{x}))).$$

This entails that an object of tier **1** has no field of tier **0**. In other words, no arrow will go from a node corresponding to a tier **1** variable to a field node of tier **0**.

Rule *(Var)* is standard.

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 α which will also be the tier of the whole expression. It prevents information to flow from tier **0** to tier **1**. Note that flows from tier **1** to tier **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 τ_i and of tier not lower than the admissible tiers α_i in the constructor typing judgment. Note that the new instance has type of the right class and tier **0** since its creation makes the memory grow (hence it cannot be of tier **1**).

Rule *(Call)* represents how to type method calls of the shape $E.\mathtt{m}(E_1, \ldots, E_n)$. First, we check that the tiered type $\mathrm{C}(\beta)$ of the self reference in the method `m` is equal to the tiered type of the instance E. While checking the method's type, the current method in the contextual typing environment is updated to $\mathtt{m}^{\mathrm{C}}$. We check that the arguments' types match the parameters' types in `m`'s signature and that the tiers of those arguments β_i are not lower than the tiers α_i in the method typing judgment. An 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.

4.4.2 Instructions

The typing rules for instructions are provided in Figure 5b.

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 `x` if both the types match and the tier β of E is higher than the tier α of `x`. The tier of the instruction will be α. This rule implies

that information may flow from tier **1** to tier **0** but not the contrary. In other words, a tier **1** variable cannot be assigned to in a tier **0** instruction block whereas a tier **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 τ is given) or not.

Rule *(Sub)* is a sub-typing rule. An instruction of tier α can also be tiered by β with $\alpha \preceq \beta$. This means that a tier **0** instruction, where tier **1** variables cannot be modified, can be considered as a tier **1** instruction where tier **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 is the same as for the *(Sub)* rule.

Rule *(Wh)* is the most important typing rule as it will constrain the use of while loops. In a statement $\texttt{while}(E)\{I\}$, the guard of the loop E must be a boolean expression of tier **1** so that the guard is controlled. The instruction I, of type `void`, has to be of tier **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 **1**.

Rule *(Skip)* is standard.

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 α. I_1 and I_2 are instructions, hence of type `void`, with the same tier α. Combined with rule *(Seq)*, it prevents assignments of tier **1** variables in the instructions I_1 and I_2 to be controlled by a tier **0** expression.

4.4.3 Methods, constructors and executable

The typing rules for constructors and methods are provided in Figure 5c.

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 **0**.

Rules *($M_C^{\texttt{void}}$)* 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 τ, 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 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 **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 contrast, the computational instruction I is considered to be the computational part of the program and has to respect the tiering discipline.

4.5 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 contextual typing environment $\Gamma = (\mathtt{m}^C, \Delta)$ such that $\Gamma \vdash I : \mathtt{void}(\alpha)$ holds, there is a contextual typing environment $\Gamma' = (\mathtt{m}^C, \Delta')$ such that the following holds:*

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

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 $I = \tau\ \mathtt{x} = E.\mathtt{m}'(E_1, \ldots, E_n);$ such that $\Gamma \vdash I : \mathtt{void}(\alpha)$ and $\Gamma = (m^{\mathrm{C}}, \Delta)$. This means that $\Gamma \vdash E_i : \tau_i(\alpha_i)$, $\Gamma \vdash \mathtt{x} : \tau(\alpha)$ and $\Gamma \vdash E : \mathrm{C}'(\beta)$ hold, for some α_i, β and C'. The flattening of I is of the shape $\overline{J}\ [\tau]\ \mathtt{x} = \mathtt{x}_{n+1}.\mathtt{m}'(\mathtt{x}_1, \ldots, \mathtt{x}_n)$; with $J = \tau_1\ \mathtt{x}_1{:=}E_1; \ldots \tau_n\ \mathtt{x}_n{:=}E_n; \tau_{n+1}\ \mathtt{x}_{n+1}{:=}E;$. Let $\Gamma' = (m^{\mathrm{C}}, \Delta')$ for the environment Δ' defined, for each method n^{D}, by:

$$\Delta'(n^{\mathrm{D}})(\mathtt{y}) = \begin{cases} \tau(\alpha) \text{ if } \mathtt{y} = \mathtt{x} \\ \tau_i(\alpha_i) \text{ if } \mathtt{y} \in \{\mathtt{x}_1, \ldots, \mathtt{x}_n\} \\ \mathrm{C}'(\beta) \text{ if } \mathtt{y} = \mathtt{x}_{n+1} \\ \Delta(n^{\mathrm{D}})(\mathtt{y}) \text{ otherwise} \end{cases}$$

We have $\Gamma' \vdash J\ [\tau]\ \mathtt{x} = \mathtt{x}_{n+1}.\mathtt{m}'(\mathtt{x}_1, \ldots, \mathtt{x}_n);: \mathtt{void}(\alpha)$ and $\Gamma' \vdash J : \mathtt{void}(\alpha)$ (sub-typing might be used). By induction hypothesis, there is a contextual typing environment $\Gamma'' = (\mathtt{m}^{\mathrm{C}}, \Delta'')$ such that $\Gamma'' \vdash \overline{J} : \mathtt{void}(\alpha)$ and $\forall n^{\mathrm{D}},\ \forall \mathtt{x} \in dom(\Delta'(n^D)),\ \Delta''(n^D)(\mathtt{x}) = \Delta'(n^{\mathrm{D}})(\mathtt{x})$ and, consequently, $\Gamma'' \vdash \overline{I} : \mathtt{void}(\alpha)$. All the other cases are treated similarly. □

5 Illustrating examples

5.1 Sorting lists

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 lists of integers. Tiers are made explicit in the code: The notation $\mathtt{x}^\alpha$ means that `x` has tier α under the considered contextual typing environment Γ, i.e. $\Gamma \vdash \mathtt{x} : \tau(\alpha)$, for some τ, whereas $I : \alpha$ means that $\Gamma \vdash I : \mathtt{void}(\alpha)$.

```
1 BList {  //List of booleans
2   boolean value;
3   BList queue;
4
5   BList(boolean v, BList q) {
6     value = v;
7     queue = q;
8   }
```

The constructor `BList` can be typed by $\mathrm{boolean}(\alpha) \times \mathrm{Blist}(\beta) \to \mathrm{BList}(\mathbf{0})$, with $\alpha,\ \beta \in \{\mathbf{0}, \mathbf{1}\}$, depending on the local variable and parameter typing environment, by rule (K_{C}).

```
9   BList getQueue() {
10    return queue;
11  }
```

The method `getQueue` can be typed by BList(**1**) → BList(**1**) or BList(**0**) → BList(**1**), if $\Delta(\text{getQueue}^{\text{BList}})(\text{queue})=\mathbf{1}$, and by BList(**0**) → BList(**0**), if $\Delta(\text{getQueue}^{\text{BList}})(\text{queue})=\mathbf{0}$, by rules ($M_{\text{C}}$) and (Self).

The type BList(**1**) → BList(**0**) is prohibited by rule (Self) since the tier of the current object **1** has to be lower than the minimum tier of its arguments **0**.

```
12   void setQueue(BList q) {
13     queue = q;
14   }
```

The method setQueue can be given the types BList(**1**) × BList(**1**) → `void`(**1**), BList(**0**) × BList(**1**) → `void`(α), or BList(**0**) × BList(**0**) → `void`(α), $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

The input type BList(**1**) × BList(**0**) is prohibited since the tier of parameter `q` (**0**) has to be greater than the tier of the field `queue`, by rule (Ass); but this latter tier has to be greater than the tier of the current object (**1**), by rule (Self). Finally, the output tier corresponds to the tier of the typed instruction by rule (M_{C}). Consequently, it also corresponds to the tier of the field `queue` by rule (Ass) and is enforced to be **1** when the current object has type **1**, by the rule (Self). Notice that it can be **1** whenever the object's current tier is **0** using the subtyping rule (Sub).

```
15   boolean getValue() {
16     return value;
17   }
```

`getValue` can be given the types BList(**1**) → boolean(**1**), BList(**0**) → boolean(**1**) or BList(**0**) → boolean(**0**) (the explanations are the same than for the `getQueue` getter).

```
18   BList double() {
19     BList n⁰ = new BList(false, this);
20     return n⁰;
21   }
```

The method `double` can be typed by BList(**0**) → BList(**0**) or BList(**1**) → BList(**0**). Indeed the local variable `n` is enforced to be of tier **0** by a combination of rules (K_{C}) and (Ass). Consequently, the method output type is BList(**0**) since it has to match the type of the returned variable. Finally, there is no constraint on the current object admissible types since it is left unchanged by the method.

```
22   void decrement() {
23     if (value¹ == true or value¹ == null) {
24       value¹ = false; :1
25     } else {
26       if (queue¹ != null) {
27         value = true;
28         queue¹.decrement(); :1
29       } else {
30         value¹ = false; :1
31       }
32     }
33   } :1
```

The method `decrement` is recursive and can be given the BList(**1**) → `void`(**1**) (as we shall see later in Subsection 6.2, this type is mandatory by safety condition). This type enforces the tier of the each field to be **1** by rule (Self). Finally, the method body can be typed using a combination of rules (Ass), (If) and (Call).

```
34  void concat(BList other¹) {
35    BList o¹ = this¹;:1
36    while (o¹.getQueue() != null) {
37      o¹ = o¹.getQueue();:1
38    }
39    o¹.setQueue(other¹);:1
40  }
```

In the `concat` method, the presence of the method call `o.getQueue()` in the guard of the while loop enforces its ouput tier to be **1** by rules (Wh), (Null) and (Op). Consequently, the type of `getQueue` has to be BList(**1**) → BList(**1**) under the considered contextual typing environment (see the admissible types of `getQueue`). It also enforces the object `o` to be of tier **1**. Consequently, the current object `this` is also enforced to be of tier **1** by rule (Ass) and the tier of the parameter other is enforced to be **1** (see the `setQueue` admissible types). Consequently, the only admissible type for `concat` is BList(**1**) × BList(**1**) → `void`(**1**), using rule (Seq).

```
41  boolean isEqual(BList other¹) {
42    boolean res⁰ = true; :1 //using (Sub)
43    BList b1¹ = this¹; :1
44    BList b2¹ = other¹; :1
45    while (b1¹ != null && b2¹ != null){
46       if(b1¹.getValue()!=b2¹.getValue()){
47         res⁰ = false;:1 //using (Sub)
48       } else {; }
49       b1¹ = b1¹.getQueue();:1
50       b2¹ = b2¹.getQueue();:1
51    }
52    if (b1¹ != null || b2¹ != null) {
53      res⁰ =false;:1 //using (Sub)
54    } else {; } :1 //using (Sub)
55    return res⁰;
56  }
```

The local variables `b1` and `b2` are enforced to be of tier **1** by rule (Wh). Consequently, `this` and `other` are also of tier **1** using twice rule (Ass). Moreover, the methods `getValue` and `getQueue` will be typed by BList(**1**) → boolean(**1**) and BList(**1**) → BList(**1**), respectively. Finally, the local variable can be given type boolean(**1**) or boolean(**0**) (in the latter case, the subtyping rule (Sub) will be needed) and, consequently, the admissible types for `isEqual` are BList(**1**) × BList(**1**) → boolean(α), $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

```
57  boolean lessOrEqualTo(BList other¹) {
58    BList b1¹ = this¹;
59    BList b2¹ = other¹;
60    boolean res¹ = true;
61    while (b1¹ != null &&
```

```
62              b2[1] != null) {
63        if (!b1.getValue() &&
64              b2.getValue()) {
65          res = true;
66        } else {;}
67        if (b1.getValue() &&
68             !b2.getValue()) {
69          res = false;
70        } else {;}
71        if (b1.getQueue() == null &&
72              b2.getQueue() != null) {
73          res = true;
74        } else {;}
75        if (b2.getQueue() == null &&
76              b1.getQueue() != null) {
77          res = false;
78        } else {;}
79        b1 = b1.getQueue();
80        b2 = b2.getQueue();
81      }
82      return res[1];
83    }
84 }
```

The explanations are the same as the ones for the `isEqual` method. Consequently, the admissible types are $\text{BList}(\mathbf{1}) \times \text{BList}(\mathbf{1}) \rightarrow \text{boolean}(\alpha)$, $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

Now we let the reader check the methods of the class `IList` can be typed as follows using the typing environment Δ such that: $\Delta(\mathtt{m}^{\text{IList}})(\text{value})=\mathbf{1}$ and $\Delta(\mathtt{m}^{\text{IList}})(\text{queue})=\mathbf{1}$, for each method `m` of the class `IList`.

```
1 IList {// List of Integers
2   BList value;
3   IList queue;
4
5   IList(BList v, IList q) {
6     value = v;
7     queue = q;
8   }
9
10  BList getValue() {
11   return value[1];
12  }
13
14  IList getQueue() {
15    return queue[1];
16  }
17
18  void setQueue(IList q[1]) {
```

```
19      queue = q; :1
20    }
21
22     BList max() {
23      BList currentMax¹ = null;:1
24      IList o¹ = this¹;:1
25      while (o¹ != null) {
26        BList v¹ = o¹.getValue();:1
27        if (currentMax¹.lessOrEqualTo(v¹)) {
28          currentMax¹ = v¹;:1
29        } else {; }
30        o¹ = o¹.getQueue();:1
31      }
32      return currentMax¹;
33    }
34
35    void remove(BList element¹) {
36      IList o¹ = this¹;
37      IList p¹ = null;
38      while(!o¹.getValue().isEqual(element¹)){
39        p¹ = o¹;
40        o¹ = o.getQueue();
41      }
42      if (p¹ != null) {
43        p¹.setQueue(o¹.getQueue()¹);:1
44      } else {
45        this¹ = o¹.getQueue()¹;:1
46      }
47    }
48
49    IList sort() {
50      IList o¹ = this;
51      IList s⁰ = null;
52      while (o¹ != null) {
53        m¹ = o¹.max(); :1
54        s⁰ = new IList(m¹, s⁰); :0
55        o¹.remove(m¹); :1
56      } //by rules (Seq) and (Wh)
57      return s⁰;
58    }
59 }
```

Now we show that the executable Exe using instances of both `BList` and `IList` is well-typed. First notice that the type system only checks constraints on base type in the initialization instruction. Second, constraints on tiered types are checked in the computational instruction: since the method sort can be given IList(**1**) → IList(**0**), the variable `s` is enforced to be of the tiered type IList(**0**) (as the corresponding data might increase though it is not the case in this

particular example) whereas variable `l` is enforced to be of tier **1** (as the method sort iterates on its data). Finally, `i3` has to be of tier **1** because of the type given to the method decrement (for more details, see also the definition of safety in Subsection 6.2).

```
1 Exe {
2   main() {
3     // Initialization instruction
4     BList i1 = new BList(true,
5                          new BList(true, null));
6     BList i2 = new BList(true, null);
7     BList i3 = new BList(false,
8                          new BList(true, null));
9     IList l = new IList(i1,
10                        new IList(i2,
11                        new IList(i3, null)));
12
13    // Computational instruction
14    IList s^0 = l^1.sort(); :0
15    i3^1.decrement(); :1
16  }
17 }:◇
```

Note that this implementation of `sort` creates a new structure (of tier **0**), hence is not composable with itself. In the case of sort, it is possible to do it in place, meaning that it would be of type IList(**1**) → IList(**1**) which in turn would be composable. This illustrates the fact that if one wants to compose methods in a typable way, there needs to be at most one method from tier **1** to tier **0**.

5.2 Cyclic structure

As we have previously defined a list of booleans (for simulating integers), we can design classes for cyclic data. This is illustrated by the following example.

```
Ring { //class implementing a ring as a cyclic list

    boolean data;
    Ring next;
    Ring prev;

    Ring(boolean data, Ring next, Ring prev) {
        this.data = data;
        this.next = next;
        this.prev = prev;
    }

    void init() {//to be called after the constructor.
        if (next == null) {
            next = this;
            prev = this;
```

```
        } else {
            prev = next.prev;
            next.setPrev(this);
            prev.setNext(this);
        }
    }

    boolean getData() { return this.data; }
    // getData() : Ring(1) → boolean(1)
    Clist getNext() { return this.next; }
    // getNext() : Ring(1) → Ring(1)
    Ring getPrev() { return this.prev; }
    void setPrev(Ring nprev) { this.prev = nprev; }
}

Exe {
    main() {
        // Create a Ring
        Ring a = new Ring(true, null);
        a.init();
        Ring input = new Ring(true, a);
        input.init();
        //end of initialization
        //Search for a false in the input.
        copy¹ = input¹;
        while (copy¹.getData() != false) {
            copy¹ = copy¹.getNext();
        }
}
```

The main program can be typed by the above annotations. It is obvious that if the main program halts, it will do so in time linear in the size of the input. But it can loop infinitely if the ring does not contain any `false`.

6 Upper bound on the stack size and the heap size

In this section, we state our main result showing that well-typed programs have both pointer stack size and pointer graph size bounded polynomially by the input size under termination and safety assumptions. Moreover, precise upper bounds can be extracted.

6.1 Size, level and intricacy

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 $|\mathcal{G}_\mathcal{P}|+|dom(\mathcal{P})|+|\mathcal{S}_\mathcal{G}|+|dom(\sigma)|$.*

Since a pointer graph contains both references to the objects (the nodes) and references to the field 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 out-degree of a node is bounded by a constant of the program (the maximum number of fields 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_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 $\mathtt{m}'$ is called in M_C, i.e. in the body of M_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_C of name $\mathtt{m}$ is clear from the context, we will write $[\mathtt{m}]$ as an abuse of notation for $[s]$ and say that M_C is a recursive method. Finally, we write $s \sqsubset^+_{\not\equiv} s'$ 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 λ of a method signature be defined as follows:*

- $\lambda(s) = 0$ *if* $s \notin [s]$
- $\lambda(s) = 1 + \max\{\lambda(s') \mid s \sqsubset^+_{\not\equiv} s'\}$ *otherwise.*[4]

By abuse of notation, we will write $\lambda(\mathtt{m})$ when the signature of $\mathtt{m}$ is clear from the context. For a given program, we denote the maximal level of a method by λ.

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 ν of a meta-instruction be defined as follows:*

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

Moreover, let ν be the maximal intricacy of a meta-instruction within a given program.

Notice that both intricacy ν and level λ are bounded by the size of their corresponding program.

[4] As usual, we assume that $\max \emptyset = 0$

6.2 Safety restriction on recursive methods

Now we put some side 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 **1** input (as the guard of a `while`) and output (to prevent a recursive dependence on a tier **0** variable).

Definition 10 (Safety)**.** *A well-typed program wrt a typing environment* Δ *is* safe *if for each recursive method* $M_C = \tau\ \mathtt{m}(\tau_1\ \mathtt{y}_1, \ldots, \tau_n\ \mathtt{y}_n)\{MI\ [\mathtt{return\ x;}]\}$*:*

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

$$(\epsilon, \Delta) \vdash M_C : C(\mathbf{1}) \times \tau_1(\mathbf{1}) \times \cdots \times \tau_n(\mathbf{1}) \to \tau(\mathbf{1}).$$

Remark 1. *A program is safe with respect to a typing environment* Δ *iff its flattened version is safe with respect to the contextual typing environment* (ϵ, Δ') *that is obtained using Proposition 1.*

Example 1. *The program of Section 5 is safe. Indeed the recursive method decrement can be typed by BList(***1***)* $\to \mathtt{void}(\mathbf{1})$*, does not contain any while loop and has only one recursive call in its body.*

6.3 Intermediate lemmata

In this section, we introduce intermediate lemmata that allow us to prove the main result. In order to present a simple proof, we will make the assumption that all the fields of each class C have the same tier under all the distinct field typing environment (i.e. $\forall x,\ \forall \mathtt{m}, \mathtt{m}',\ \Delta(\mathtt{m}^{\mathrm{C}})(\mathtt{x}) = \Delta(\mathtt{m}'^{\mathrm{C}})(\mathtt{x})$). This assumption is not strong in the sense that a class C with a field `x` having distinct tiers depending on the considered method can be easily transformed into a typable class satisfying this condition. For this purpose, it suffices to replace `x` by two new fields $\mathtt{x_0}$ and $\mathtt{x_1}$ of the same base type τ such that $\mathtt{x}_\alpha$ will only be used in methods $\mathtt{m}^C$ where $\Delta(\mathtt{m}^{\mathrm{C}})(\mathtt{x}) = \tau(\alpha)$. Given a memory configuration $\mathcal{C}$ and a typing environment Δ, define the *tier* **1** *memory configuration* $\mathcal{C}_{\Delta_1}$ by:

$$\mathcal{C}_{\Delta_1}(\mathtt{x}) = \begin{cases} \mathcal{C}(\mathtt{x}) & \text{if } \Delta(\mathtt{m}^{\mathrm{C}})(\mathtt{x}) = \tau(\mathbf{1}),\ \tau \in \mathbb{C} \cup \{\mathtt{boolean}\} \\ \bot & \text{otherwise} \end{cases}$$

where the symbol $\bot$ means that $\mathcal{C}_{\Delta_1}$ is undefined on the given input. Given a configuration $\mathcal{C}$ and a meta-instruction MI, the *distinct tier* **1** *configuration sequence* $\xi_{\Delta_1}(\mathcal{C}, MI)$ wrt contextual typing environment Δ, is defined by:

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

$$\xi_{\Delta_1}(\mathcal{C}, MI) = \begin{cases} \mathcal{C}_{\Delta_1}.\xi_{\Delta_1}(\mathcal{C}', MI') & \text{if } \mathcal{C}_{\Delta_1} \neq \mathcal{C}'_{\Delta_1} \\ \xi_{\Delta_1}(\mathcal{C}', MI') & \text{otherwise} \end{cases}$$

- If $MI = \epsilon$ then $\xi_{\Delta_1}(\mathcal{C}, MI) = \mathcal{C}_{\Delta_1}$.

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, $\xi_{\Delta_1}(\mathcal{C}, MI)$ is a record of the distinct tier **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 **0** variables to **1** variables.

Lemma 2 (Non-interference)**.** *Given a meta-instruction MI of a safe program with respect to typing environment Δ, let $\mathcal{C}$ and $\mathcal{C}'$ be two memory configurations, if $\mathcal{C}_{\Delta_1} = \mathcal{C}'_{\Delta_1}$ then $\xi_{\Delta_1}(\mathcal{C}, MI) = \xi_{\Delta_1}(\mathcal{C}', MI)$. In other words, tier* **1** *variables do not depend on tier* **0** *variables.*

Proof. First, note that Rule *(Wh)* of Figure 5b 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 **1**. Applying Proposition 1, tier **1** meta-instructions do not depend on loops controlled by tier **0** expressions. Second, in a `if` meta-instruction of tier **0** guard, all the commands are of tier **0** by Rule *(If)*. Consequently, no tier **1** variable is updated in these commands. Indeed a tier **1** variable assignment enforces the containing command to be of tier **1** using Rule *(Ass)* and Rule *(Seq)*. Finally, the rule (Ass) in Figure 5b enforces that tier **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 **1** (except the current object or the parameters in the special case of non recursive methods) by the type system. Consequently, if $\mathcal{C}_{\Delta_1} = \mathcal{C}'_{\Delta_1}$ and $(\mathcal{C}, \mathtt{x}{:=}E;\ MI') \rightarrow (\mathcal{D}, MI')$ then $(\mathcal{C}', \mathtt{x}{:=}E;\ MI') \rightarrow (\mathcal{D}', MI')$ and $\mathcal{D}_{\Delta_1} = \mathcal{D}'_{\Delta_1}$. Now the case of a non recursive method is trivial since its code can be inlined while still being typed under Δ. And so is the case where the current object variable is of tier **0** since the method return variable tier is enforced to be **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. □

Using Lemma 2, if a safe program evaluation encounters twice the same meta-instruction under two configurations equal on tier **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 environment Δ, if $(\mathcal{C}, MI) \rightarrow^+ (\mathcal{C}', MI)$ and $\mathcal{C}_{\Delta_1} = \mathcal{C}'_{\Delta_1}$, then the meta-instruction MI does not terminate on memory configuration $\mathcal{C}$.*

Proof. Assume that during the transition $(\mathcal{C}, MI) \rightarrow^+ (\mathcal{C}', MI)$ there is a $\mathcal{C}''$ such that $\mathcal{C}''_{\Delta_1} \neq \mathcal{C}_{\Delta_1}$, then the distinct tier **1** configuration sequence $\xi_{\Delta_1}(\mathcal{C}, MI)$ contains this $\mathcal{C}''_{\Delta_1}$ before $\mathcal{C}'_{\Delta_1}$. From the construction of the sequence, we deduce that $\xi_{\Delta_1}(\mathcal{C}, MI)$ is of the shape $\ldots \mathcal{C}''_{\Delta_1} \ldots \xi_{\Delta_1}(\mathcal{C}', MI)$. However from Lemma 2, $\xi_{\Delta_1}(\mathcal{C}, MI) = \xi_{\Delta_1}(\mathcal{C}', MI)$, hence it is infinite and the meta-instruction MI does not terminate on memory configuration $\mathcal{C}$.

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

Lemma 3 permits to demonstrate that the number of distinct tier **1** memory configurations encountered during the evaluation of a terminating and safe program is polynomially bounded

in the input size. Indeed, as tier **1** variables cannot grow, they can only refer to some node of the input memory configuration. As the number of such variables is bounded by the program size.

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

$$\#\left\{\mathcal{C}'_{\Delta_1} \mid (\mathcal{C}, MI) \rightarrow^* (\mathcal{C}', MI')\right\} \leq |\mathcal{C}|^{n_{\mathbf{1}}}.$$

*where $n_{\mathbf{1}}$ is the number of tier **1** variables in the whole program.*

Proof. By Lemma 2, there is no information flow from tier **0** to tier **1**. Moreover, Rule *(New)* of Figure 5a enforces that tier **1** expressions cannot correspond to the creation of a new instance. Indeed in an assignment of the shape $\mathtt{x}$:=`new` $\mathrm{C}(\mathtt{y}_1, \ldots, \mathtt{y}_n)$, the judgment $\Gamma \vdash$ `new` $\mathrm{C}(\mathtt{y}_1, \ldots, \mathtt{y}_n) : \mathrm{C}(\mathbf{0})$ holds and, consequently, the judgment $\Gamma \vdash \mathtt{x} : \mathrm{C}(\mathbf{1})$ cannot hold because of Rule *(Ass)*. Consequently, variables of tiered type $\mathrm{C}(\mathbf{1})$ 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 `x` of tier **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 `x` is in the domain of the primitive store of $\mathcal{C}$. The number of tier **1** variables being at most $n_{\mathbf{1}} = \sum_{\delta^{\mathbf{1}} \in \Delta_{\mathbf{1}}} \#dom(\delta^{\mathbf{1}})$, by definition of $\Delta_{\mathbf{1}}$, the number of distinct configurations is bounded by $|\mathcal{C}|^{n_{\mathbf{1}}}$. □

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 with respect to typing environment Δ such that MI terminates on input $\mathcal{C}$. Each* `while` *loop in MI can be executed at most $|\mathcal{C}|^{n_{\mathbf{1}} \times \nu(MI)}$ 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 we have:

$$MI = MI_1 \ \mathtt{while(x)\{}MI'\mathtt{\}}\ MI_2,$$

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 bounded by $|\mathcal{C}|^{n_{\mathbf{1}}}$, 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 $MI = MI_1 \ \mathtt{while(x)\{}MI'\mathtt{\}}\ MI_2$ 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 $k \leq |\mathcal{C}|^{n_{\mathbf{1}}}$. By induction hypothesis, any while loop within MI' can be executed at most $|\mathcal{C}|^{n_{\mathbf{1}} \times \nu(MI')}$ and, consequently, it can be executed at most $k \times |\mathcal{C}|^{n_{\mathbf{1}} \times \nu(MI')} \leq |\mathcal{C}|^{n_{\mathbf{1}} \times (\nu(MI')+1)} \leq |\mathcal{C}|^{n_{\mathbf{1}} \times \nu(MI)}$ 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* $MI = [[\tau]\mathtt{x}{:=}]\mathtt{y.m}(\mathtt{y}_1, \ldots, \mathtt{y}_n)$; *of a safe program with respect to typing environment* Δ. *If* $(\mathcal{C}, MI) \to^k (\mathcal{C}', \epsilon)$ *(i.e. MI terminates on input* $\mathcal{C}$ *in k steps) then* $k = O(|\mathcal{C}|^{n_1 \times (\nu+1) \times \lambda(\mathtt{m})})$.

Proof. By induction on the level. Consider a method of the shape:

$$\tau\ \mathtt{m}(\ldots)\{MI'\ [\mathtt{return\ z};]\}.$$

If $\lambda(\mathtt{m}) = 1$. By definition of the level, this means $\mathtt{m} \notin [\mathtt{m}]$, i.e. the method m is not recursive. Hence, by Lemma 5, each meta-instruction can be executed at most $|\mathcal{C}|^{n_1 \times (\nu(MI')+1)}$ (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 $|MI'| \times |\mathcal{C}|^{n_1 \times (\nu(MI')+1)}$ executed meta-instruction before the program terminates. i.e. $k = O(|\mathcal{C}|^{n_1 \times ((\nu+1) \times 1)})$.

Assume $\lambda(\mathtt{m}) = i + 1$. Either m is recursive, this means $\mathtt{m} \in [\mathtt{m}]$. Hence, by safety assumption and by Lemma 4, we know that there are at most $|\mathcal{C}|^{n_1}$ 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 $|MI'| \times |\mathcal{C}|^{n_1}$. Consequently, the number of meta-instructions unfolded by method calls of [m] is at most $O(|\mathcal{C}|^{n_1})$ (indeed just take the finite sum of $|MI'| \times |\mathcal{C}|^{n_1}$, 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 $O(|\mathcal{C}|^{n_1 \times ((\nu+1) \times i)})$. Putting all together, it generates at most $O(|\mathcal{C}|^{n_1} \times |\mathcal{C}|^{n_1 \times ((\nu+1) \times i)}) = O(|\mathcal{C}|^{n_1 \times ((\nu+1) \times \lambda(\mathtt{m}))})$ 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 ν times. This means that the $|MI'|$ meta-instructions of level i can be unfolded $|\mathcal{C}|^{n_1 \nu}$ times from Lemma 4. Since by induction hypothesis, each can yield $O(|\mathcal{C}|^{n_1 \times ((\nu+1) \times i)})$ meta-instructions, it finally gives $O\left(|\mathcal{C}|^{n_1 \times ((\nu+1) \times (i+1))}\right)$ meta-instructions. □

6.4 Main results

First, the presented type system allows us to infer polynomial upper bounds on the stack and the heap of safe programs.

Theorem 1. *If a core Java program of computational instruction I is safe wrt to typing environment* Δ *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:*

$$|\mathcal{C}'| = O(|\mathcal{C}|^{n_1((\nu+1)\lambda)}).$$

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* $O(|\mathcal{C}|^{n_1((\nu+1)\lambda)})$.

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 obtained by 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). □

Example 2. *Consider the program presented in Section 5. This program is clearly terminating and safe wrt to the provided typing environment. Moreover its maximal intricacy ν is equal to* 1 *since there is no nested while loops in its methods. Moreover, its maximal level λ is equal to* 2 *since there is one level of recursion in the method decrement. Consequently, applying Theorem 1, we obtain a $O(|\mathcal{C}|^{4\times n_{\mathbf{1}}})$ upper bound on the memory use. Notice that this global upper bound can be improved to obtain a tighter upper bound (at the price of a non-uniform formula) by only considering particular instructions. For example, the instruction IList $s^{\mathbf{0}} = l^{\mathbf{1}}.sort(); : \mathbf{0}$ will have a $O(|\mathcal{C}|^2)$ complexity since there is only one tier* **1** *variable l, $\nu(sort) = 1$ and $\lambda(sort) = 1$. The instruction $i3^{\mathbf{1}}.decrement(); : \mathbf{1}$ will have a linear complexity since there is only one tier* **1** *variable $i3$, one recursive call and $\nu(decrement) = 0$.*

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

Corollary 1. *If a core Java program of computational instruction I is safe wrt to typing environment Δ and terminates on input $\mathcal{C}$ then it does terminate in time $O(|\mathcal{C}|^{n_{\mathbf{1}}((\nu+1)\lambda)})$.*

Another corollary of interest is that tier **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.

Corollary 2. *If a core Java program of computational instruction I is safe wrt to typing environment Δ then, on input $\mathcal{C}$, for each memory configuration $\mathcal{C}'$, meta-instruction MI s.t. $(\mathcal{C}, MI) \to^* (\mathcal{C}', MI')$ and a local variable* `x` *of tier* **1** *we have:*

$$|\mathcal{C}'(\mathtt{x})| = O(|\mathcal{C}|^{n_{\mathbf{1}}((\nu+1)\lambda)}).$$

where $|\mathcal{C}'(\mathtt{x})|$ denotes the size of the subgraph of nodes reachable from node $\mathcal{C}'(\mathtt{x})$ whenever `x` *is of reference type.*

Another 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 [23] 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. Finally, we show the decidability of type inference:

Proposition 2 (Type inference)**.** *Deciding if there exists a typing environment Δ 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 fields declarations. We encode the tier of each field `x` within the method `m` of class C by a boolean variable $x^{\mathtt{m}^{\mathrm{C}}}$ that will be true if the variable is of tier **1**, false if it is of tier **0** in the context of $\mathtt{m}^{\mathrm{C}}$. All local variables and parameters can be encoded by a single variable as their tier is independent from the context. Each instruction generates some constraints. For example, in the case of an

assignment $\mathtt{x} := \mathtt{y}$; in the context $\mathtt{m}^{\mathrm{C}}$, we have to check $\pi_2(\Delta(\mathtt{m}^{\mathrm{C}})(\mathtt{x})) \preceq \pi_2(\Delta(\mathtt{m}^{\mathrm{C}})(\mathtt{y})))$, which can be represented as $(y^{\mathtt{m}^{\mathrm{C}}} \vee \neg x^{\mathtt{m}^{\mathrm{C}}})$. All these constraints generate a conjunction 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. □

7 Extensions

7.1 Control flow alteration

Constructs altering the control flow like break, return and continue can also be considered in our fragment. For example, a break statement has to be constrained to be of tier **1** so that if such an instruction is to be executed, then we know that it does not depend on tier **0** expressions. More precisely, it can be typed by the rule:

$$\frac{}{\Gamma \vdash \mathtt{break;} : \mathtt{void}(\mathbf{1})}\ (Break)$$

This prevent the programmer from writing conditionals of the shape: Using the same kind of typing rule, return statements can also be used in a more flexible manner by allowing the execution to leave the current subroutine anywhere in the method body.

7.2 Inheritance

Inheritance is a major trait of OOP that has to be treated by any reasonable static analysis tool on Java like programs. In this subsection, 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.

Syntax. We extend the class grammar by class declarations of the shape:

$$\mathrm{D}\ \mathtt{extends}\ \mathrm{C}\{\tau_1\ \mathtt{x}_1; \ldots; \tau_n\ \mathtt{x}_n;\ K_{\mathrm{D}}\ M_{\mathrm{D}}^1 \ldots M_{\mathrm{D}}^k\}$$

with $\mathrm{D} \in \mathbb{C}$ and K_{D} being a constructor initializing both the fields of C and the fields of D. 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.

Semantics. The semantics can be extended by creating a new node of label D in the graph each time a `new` $\mathrm{D}(\ldots)$ 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 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 searches for the method signature in the corresponding class and evaluates its body once found. In the particular case where this signature does not exist, the search is extended to the super class C and, so on.

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

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

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:

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

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

Safety. Now a method call can be evaluated dynamically depending on the current object type. It 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:

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

i.e. $\mathtt{m}^{\mathrm{C}}$ is considered to call its override $\mathtt{m}^{\mathrm{D}}$ by dynamic binding.

8 Conclusion

This work presents a simple but highly expressive type-system that can be checked in linear time and that provides explicit polynomial upper bounds on the heap and stack size of an object oriented program allowing (recursive) method calls. 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. See Example 2). OO features, such as abstract classes, interfaces and static fields and methods, were not considered here, but we claim that they can be treated by our analysis. Finally, notice that the safety condition can be alleviated by ensuring that only one recursive call is reachable in the execution of the method body thus improving expressivity.

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, vol. 7794, 2013, pp. 305–320.

[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] R. Kersten, O. Shkaravska, B. van Gastel, M. Montenegro, and M. C. J. D. van Eekelen, "Making resource analysis practical for real-time java," in *JTRES*, 2012, pp. 135–144.

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

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

[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.