A Type-Based Complexity Analysis of Object Oriented Programs

Emmanuel Hainry, Romain Péchoux

▶ **To cite this version:**

Emmanuel Hainry, Romain Péchoux. A Type-Based Complexity Analysis of Object Oriented Programs. Information and Computation, 2018, Information and Computation, 261 (1), pp.78-115. 10.1016/j.ic.2018.05.006 . hal-01712506

HAL Id: hal-01712506
https://inria.hal.science/hal-01712506

Submitted on 19 Feb 2018

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

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

A Type-Based Complexity Analysis of Object Oriented Programs

Emmanuel Hainry, Romain Péchoux

Université de Lorraine, CNRS, Inria, LORIA, Nancy, France

Abstract

A type system is introduced for a generic Object Oriented programming language in order to infer resource upper bounds. A sound and complete characterization of the set of polynomial time computable functions is obtained. As a consequence, the heap-space and the stack-space requirements of typed programs are also bounded polynomially. This type system is inspired by previous works on Implicit Computational Complexity, using tiering and non-interference techniques. The presented methodology has several advantages. First, it provides explicit big O polynomial upper bounds to the programmer, hence its use could allow the programmer to avoid memory errors. Second, type checking is decidable in polynomial time. Last, it has a good expressivity since it analyzes most object oriented features like inheritance, overload, override and recursion. Moreover it can deal with loops guarded by objects and can also be extended to statements that alter the control flow like break or return.

Keywords: Object Oriented Program, Type system, complexity, polynomial time.

1. Introduction

1.1. 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 problematic is highly challenging for popular object oriented programming languages which come equipped with environments for applications running on mobile and other embedded devices (e.g. Dalvik, Java Platform Micro Edition (Java ME), Java Card and Oracle Java ME Embedded).

The current paper tackles this issue by introducing a type system for a compile-time analysis of both heap and stack space requirements of OO programs thus avoiding memory errors. This type system is also sound and complete for the set of polynomial time computable functions on the Object Oriented paradigm.

[1]This work has been partially supported by ANR Project ELICA ANR-14-CE25-0005.

This type system combines ideas coming from tiering discipline, used for complexity analysis of function algebra [1, 2], together with ideas coming from non-interference, used for secure information flow analysis [3]. The current work is an extended version of [4] and is strongly inspired by the seminal paper [5].

1.2. Abstract OO language

The results of this paper will be presented in a formally oriented manner in order to highlight their theoretical soundness. For this, we will consider a generic Abstract Object Oriented language called AOO. It can be seen as a language strictly more expressive than Featherweight Java [6] enriched with features like variable updates and while loops. The language is generic enough. Consequently, the obtained results can be applied both to impure OO languages (*e.g.* Java) and to pure ones (*e.g.* SmallTalk or Ruby). Indeed, in this latter case, it just suffices to forget rules about primitive data types in the type system. Moreover, it does not depend on the implementation of the language being compiled (ObjectiveC, OCaml, Scala, ...) or interpreted (Python standard implementation, OCaml, ...). There are some restrictions: it does not handle exceptions, inner classes, generics, multiple inheritance or pointers. Hence languages such as C++ cannot be handled. However we claim that the analysis can be extended to exceptions, inner classes and generics. This is not done in the paper in order to simplify the technical analysis. The presented work captures Safe Recursion on Notation by Bellantoni and Cook [1] and we conjecture that it could be adapted to programs with higher-order functions. The intuition behind such a conjecture is just that tiers are very closely related to the ! and § modalities of light logics [7].

1.3. Intuition

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 in two universes: tier **0** universe and tier **1** universe. In this setting, the high security level is tier **0** while low security level is tier **1**. 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 the presented 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 storages for the computed data. This is the reason why, in analogy with information-flow analysis, tier **0** is the high security level of the current setting, though this naming is opposed to the ICC standard interpretation where tier **1** is usually seen as "safer" than **0** because its use is controlled and restricted.

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 $O(n^k)$. For this, we put some restrictions on operations that can make the memory grow : constructors for the heap and operators and method calls for the stack.

1.4. Example

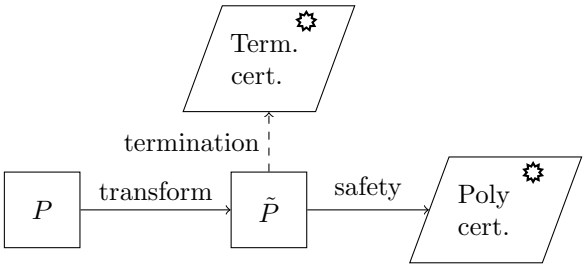
Consider the following Java code duplicating the length of a boolean `BList` as an illustrating example:

```
y := x.clone();
while (x != null){
    y := new BList(true,y);
    x := x.getQueue();
}
```

The tier of variable `x` will be enforced to be **1** since it is used in a while loop guard and in the call of the recursive method `clone`. On the opposite, the tier of variable `y` will be enforced to be **0** since the `y:=new BList(true,y);` instruction enlarges the memory use. For each assignment, we will check that the tier of the variable assigned to is equal to (smaller than for primitive data) the tier of the assigned expression. Consequently, the assignment `y:=x.clone();` is typable in this code (since the call `x.clone();` is of tier **0** as it makes the memory grow) whereas it cannot be typed if the first instruction is to be replaced by either `x:=y.clone();` or `x:=y`.

1.5. Methodology

The OO program complexity analysis presented in this paper can be summed up by the following figure:



In a first step, given a program P of a given OO programming language, we first apply a transformation step in order to obtain the AOO program $\tilde{P}$. This transformation contains the following steps:

- convert each syntactical construct of the source language in P to the corresponding construct in the abstract OO language. In particular, for statements can be replaced by while statements,
- for all public fields of P, write the corresponding getter and setter in $\tilde{P}$,
- α-convert the variables so that there is no name clashes in $\tilde{P}$,
- flatten the program (this will be explained in Subsection 3.5).

All these steps can be performed in polynomial time and the program abstract semantics is preserved. Consequently, P terminates iff $\tilde{P}$ terminates.

In a second step, a termination check and a safety check can be performed in parallel. The termination certificate can be obtained using existing tools (see the related works subsection). As the semantics is preserved, the check can also be performed on the original program P or on the compiled bytecode. In the safety check, a polynomial time type inference (Proposition 5) is performed together with a safety criterion check on recursive methods. This latter check called *safety* can be checked in polynomial time. A more general criterion, called general safety and which is unfortunately undecidable, is also provided. If both checks succeed, Theorem 2 ensures polynomial time termination.

1.6. Outline

In Section 2 and 3, the syntax and semantics of the considered generic language AOO are presented. Note that the semantics is defined on meta-instructions, flattened instructions that make the semantics formal treatment easier. The main contribution: a tier based type system is presented in Section 4 together with illustrating examples. In Section 5, two criteria for recursive methods are provided: the safety that is decidable in polynomial time and the general safety that is strictly more expressive but undecidable. Section 6 establishes the main non-interference properties of the type system. Section 7 and 8 are devoted to prove soundness and, respectively, completeness of the main result: a characterization of the set of polynomial time computable functions. As an aside, explicit space upper bounds on the heap and stack space usage are also obtained. Section 9 is devoted to prove the polynomial time decidability of type inference. Section 10 discusses extensions improving the expressivity, including an extension based on declassification.

2. Syntax of AOO

In this section, the syntax of an Abstract Oriented Object programming language, called AOO, is introduced. This language is general enough so that any well-known OO programming language (Java, OCaml, Scala, ...) can be compiled to it under some slight restrictions (not all the features of these languages – threads, user interface, input/output, ... – are handled and some program transformations/refinements are needed for a practical application).

Grammar. Given four disjoint sets $\mathbb{V}$, $\mathbb{O}$, $\mathbb{M}$ and $\mathbb{C}$ representing the set of variables, the set of operators, the set of method names and the set of class names, respectively, expressions, instructions, constructors, methods and classes are defined by the grammar of Figure 1, where $\mathtt{x} \in \mathbb{V}$, $\mathtt{op} \in \mathbb{O}$, $\mathtt{m} \in \mathbb{M}$ and $\mathtt{C} \in \mathbb{C}$, where $[e]$ denotes some optional syntactic element e and where $\overline{e}$ denotes a sequence of syntactic elements $e_1, \ldots, e_n$. Also assume a fixed set of discrete primitive types $\mathbb{T}$ to be given, *e.g.* $\mathbb{T} = \{\mathtt{void}, \mathtt{boolean}, \mathtt{int}, \mathtt{char}\}$ or $\mathbb{T} = \emptyset$ in the case of a pure OO language. In what follows, $\{\mathtt{void}, \mathtt{boolean}\} \subseteq \mathbb{T}$ will always hold.

$$
\begin{array}{rcl}
\text{Expressions} \ni \mathtt{e} & ::= & \mathtt{x} \mid \mathtt{cst}_\tau \mid \mathtt{null} \mid \mathtt{this} \mid \mathtt{op}(\overline{\mathtt{e}}) \\
& & \mid \mathtt{new\ C}(\overline{\mathtt{e}}) \mid \mathtt{e.m}(\overline{\mathtt{e}}) \\
\text{Instructions} \ni \mathtt{I} & ::= & \mathtt{;} \mid [\tau]\ \mathtt{x{:=}e;} \mid \mathtt{I_1\ I_2} \mid \mathtt{while(e)\{I\}} \\
& & \mid \mathtt{if(e)\{I_1\}else\{I_2\}} \mid \mathtt{e.m}(\overline{\mathtt{e}})\mathtt{;} \\
\text{Methods} \ni \mathtt{m_C} & ::= & \tau\ \mathtt{m}(\overline{\tau\ \mathtt{x}})\{\mathtt{I}[\mathtt{return\ x;}]\} \\
\text{Constructors} \ni \mathtt{k_C} & ::= & \mathtt{C}(\overline{\tau\ \mathtt{y}})\{\mathtt{I}\} \\
\text{Classes} \ni \mathtt{C} & ::= & \mathtt{C}\ [\mathtt{extends\ D}]\ \{\overline{\tau\ \mathtt{x;}}\ \overline{\mathtt{k_C}}\ \overline{\mathtt{m_C}}\}
\end{array}
$$

Figure 1: Syntax of AOO classes

The τs are type variables ranging over $\mathbb{C} \cup \mathbb{T}$. The metavariable $\mathtt{cst}_\tau$ represents a primitive type constant of type $\tau \in \mathbb{T}$. Let $dom(\tau)$ be the domain of values of type τ. For example, $dom(\mathtt{boolean}) = \{\mathtt{true}, \mathtt{false}\}$ or $dom(\mathtt{int}) = \mathbb{N}$. $\mathtt{cst}_\tau \in dom(\tau)$ holds. Finally, define $dom(\mathbb{T}) = \cup_{\tau \in \mathbb{T}} dom(\tau)$. Each primitive operator $\mathtt{op} \in \mathbb{O}$ has a fixed arity n and comes equipped with a signature of the shape $\mathtt{op} :: \tau_1 \times \cdots \times \tau_n \to \tau_{n+1}$ fixed by the language implementation and such that $\tau_1, \ldots, \tau_n \in \mathbb{C} \cup \mathbb{T}$ and $\tau_{n+1} \in \mathbb{T}$. That is, operator outputs are of primitive type. An example of such operator will be `==`. Also note, that primitive operators can be both considered to be applied to finite data-types as for Java integers as well as infinite datatype as for Python integers.

The AOO syntax does not include a `for` instruction based on the premise that, as in Java, a for statement $\mathtt{for}(\tau\ \mathtt{x{:=}e; condition; Increment})\{\mathtt{Ins}\}$ can be simulated by the statement $\tau\ \mathtt{x{:=}e;}\ \mathtt{while(condition)}\ \{\mathtt{Ins\ Increment;}\}$. Given a method $\tau\ \mathtt{m}(\tau_1\ \mathtt{x}_1, \ldots, \tau_n\ \mathtt{x}_n)\{\mathtt{I}\ [\mathtt{return\ x;}]\}$ of $\mathtt{C}$, its signature is $\tau\ \mathtt{m^C}(\tau_1, \ldots, \tau_n)$, the notation $\mathtt{m^C}$ denoting that $\mathtt{m}$ is declared in $\mathtt{C}$. The signature of a constructor $\mathtt{k_C}$ is $\mathtt{C}(\overline{\tau})$. Note that method overload is possible as a method name may appear in several distinct signatures.

Variables toponymy. In a class $\mathcal{C} = \mathtt{C}\{\tau_1\ \mathtt{x}_1; \ldots; \tau_n\ \mathtt{x}_n;\ \ \overline{\mathtt{k_C}}\ \overline{\mathtt{m_C}}\}$, the variables $\mathtt{x}_i$ are called fields. In a method or constructor $\tau\ \mathtt{m}(\overline{\tau\ \mathtt{y}})\{\mathtt{I}[\mathtt{return\ x;}]\}$, the arguments $\mathtt{y}_j$ are called parameters. Each variable $\mathtt{x}$ declared in an assignment of the shape $\tau\ \mathtt{x{:=}e;}$ is called a local variable. Hence, in a given class, a variable is either a field, or a parameter or a local variable. Let $\mathtt{C}.\mathcal{F} = \{\overline{\mathtt{x}}\}$ to be the set of fields in a class $\mathtt{C}\ \{\overline{\tau\ \mathtt{x;}}\ \overline{\mathtt{k_C}}\ \overline{\mathtt{m_C}}\}$ and $\mathcal{F} = \cup_{\mathtt{C} \in \mathbb{C}} \mathtt{C}.\mathcal{F} \subseteq \mathbb{V}$ be the set of all fields of a given program P, when P is clear from the context.

No access level. In an AOO program, the fields of an instance cannot be accessed directly using the "." operator. Getters will be needed. This is based on the implicit assumption that all fields are `private` since there is no field access in the syntax. On the opposite, methods and classes are all `public`. This is not

a huge restriction for an OO programmer since any field can be accessed and updated in an outer class by writing the corresponding getter and setter.

Inheritance. Inheritance is allowed by the syntax of AOO programs through the use of the `extends` construct. Consequently, override is allowed by the syntax. In the case where `C extends D`, the constructors $\texttt{C}(\overline{\tau\ \texttt{y}})\{\texttt{I}\}$ are constructors initializing both the fields of `C` and the fields of `D`. Inheritance defines a partial order on classes denoted by $\texttt{C} \trianglelefteq \texttt{D}$.

AOO programs. A *program* is a collection of classes together with exactly one class $\texttt{Exe}\{\texttt{void main}()\{\texttt{Init Comp}\}\}$ with $\texttt{Init}, \texttt{Comp} \in$ Instructions. The method `main` of class `Exe` is intended to be the entry point of the program. The instruction `Init` is called the *initialization instruction*. Its purpose is to compute the program input, which is strongly needed in order to define the complexity of an AOO program (if there is no input, all terminating programs are constant time programs). The instruction `Comp` is called the *computational instruction*. The type system presented in this paper will analyze the complexity of this latter instruction. See Subsection 3.7 for more explanations about such a choice.

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 $\texttt{C} \in \mathbb{C}$.
- Each local variable `x` is both declared and initialized exactly once by a τ `x` := `e`; instruction for its first use.
- A method output type is `void` iff it has no `return` statement.
- Each method signature is unique with respect to its name, class and input types. This implies that it is forbidden to define two signatures of the shape $\tau\ \texttt{m}^{\texttt{C}}(\overline{\tau})$ and $\tau'\ \texttt{m}^{\texttt{C}}(\overline{\tau})$ with $\tau \neq \tau'$.

Example 1. *Let the class* `BList` *be an encoding of binary integers as lists of booleans (with the least significant bit in head). The complete code will be given in Example 8.*

```
BList {
  boolean value;
  BList queue;

  BList() {
    value := true;
    queue := null;
  }

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

```
  BList getQueue() { return queue; }

  void setQueue(BList q) {
    queue := q;
  }

  boolean getValue() { return value; }

  ...
}
```

3. Semantics of AOO

In this section, a pointer graph semantics of AOO programs is provided. Pointer graphs are reminiscent from Cook and Rackoff's Jumping Automata on Graphs [8]. A pointer graph is basically a multigraph structure representing the memory heap, whose nodes are references. The pointer graph semantics is designed to work on such a structure together with a stack, for method calls. The semantics is a high-level semantics whose purpose is to be independent from the bytecode or low-level semantics and will be defined on meta-instructions, a meta-language of flattened instructions with stack operations.

3.1. Pointer graph

A *pointer graph* $\mathcal{H}$ is a directed multigraph (V, A). The nodes in V are memory references and the arrows in A link one reference to a reference of one of its fields. Nodes are labeled by class names and arrows are labeled by the field name. In what follows, let l_V be the node label mapping from V to $\mathbb{C}$ and l_A be the arrow label mapping from A to $\mathcal{F}$.

The memory heap used by a AOO program will be represented by a pointer graph. This pointer graph explicits the arborescent nature of objects: each constructor call will create a new node (memory reference) of the multigraph and arrows to the nodes (memory references) of its fields. Those arrows will be annotated by the field name. The heap in which the objects are stored corresponds to this multigraph. Consequently, bounding the heap memory use consists in bounding the size of the computed multigraph.

3.2. Pointer mapping

A variable is of primitive (resp. reference) data type if it is declared using a type metavariable in $\mathbb{T}$ (resp. $\mathbb{C}$). A *pointer mapping* with respect to a given pointer graph $\mathcal{H} = (V, A)$ is a partial mapping $p_{\mathcal{H}} : \mathbb{V} \cup \{\texttt{this}\} \mapsto V \cup dom(\mathbb{T})$ associating primitive value in $dom(\mathbb{T})$ to some variable of primitive data type in $\mathbb{V}$ and a memory reference in V to some variable of reference data type or to the current object `this`.

As usual, the domain of a pointer mapping $p_{\mathcal{H}}$ is denoted $dom(p_{\mathcal{H}})$.

By completion, for a given variable $\texttt{x} \notin dom(p_{\mathcal{H}})$, let $p_{\mathcal{H}}(\texttt{x}) = \texttt{null}$, if `x` is of type `C`, $p_{\mathcal{H}}(\texttt{x}) = 0$, if `x` is of type `int`, $p_{\mathcal{H}}(\texttt{x}) = \texttt{false}$, if `x` is of type `boolean`, ...

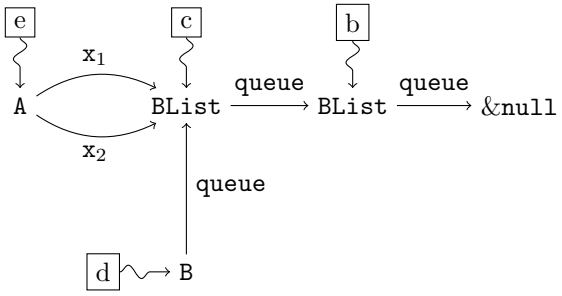


Figure 2: Example of pointer graph and pointer mapping

The use of completion is just here to ensure that the presented semantics will not get stuck.

Example 2. *Consider the class* `BList` *of Example 1. Let* `A` *be a class having two* `BList` *fields* $\mathtt{x}_1$ *and* $\mathtt{x}_2$ *and* `B` *be a class extending* `BList`*. Consider the initialization instruction* `Init` *defined by:*

```
Init ::=    BList b := new BList();
            BList c := new BList(true,b);
            B d := new B(c);
            A e := new A(c,c);
```

Figure 2 illustrates the pointer graph associated to this sequence of object creations. The figure contains both the pointer 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.

As we will see shortly, the semantics of an assignment `x := e;`, for some variable `x` of reference type, consists in updating the pointer mapping in such a way that $p_{\mathcal{H}}(\mathtt{x})$ will be the reference of the object computed by `e`. By abuse of notation, let $p_{\mathcal{H}}(\mathtt{e})$ be a notation for representing this latter reference.

In what follows, let $p_{\mathcal{H}}[\mathtt{x} \mapsto v]$, $v \in V \cup dom(\mathbb{T})$, be a notation for the pointer mapping $p'_{\mathcal{H}}$ that is equal to $p_{\mathcal{H}}$ but on `x` where the value is updated to v.

3.3. Pointer stack

For a given pointer graph $\mathcal{H}$, a *stack frame* $s_{\mathcal{H}}$ is a pair $\langle s, p_{\mathcal{H}} \rangle$ composed by a method signature $s = \tau\ \mathtt{m}^{\mathtt{C}}(\tau_1, \ldots, \tau_n)$ and a pointer mapping $p_{\mathcal{H}}$.

A *pointer stack* $\mathcal{S}_{\mathcal{H}}$ is a LIFO structure of stack frames corresponding to the same pointer graph $\mathcal{H}$. Define $\top\mathcal{S}_{\mathcal{H}}$ to be the top pointer mapping of $\mathcal{S}_{\mathcal{H}}$. Finally, define $\mathtt{pop}(\mathcal{S}_{\mathcal{H}})$ to be the pointer stack obtained from $\mathcal{S}_{\mathcal{H}}$ by removing $\top\mathcal{S}_{\mathcal{H}}$ and $\mathtt{push}(s_{\mathcal{H}}, \mathcal{S}_{\mathcal{H}})$ to be the pointer stack obtained by adding $s_{\mathcal{H}}$ to the top of $\mathcal{S}_{\mathcal{H}}$.

In what follows, we will write just `pop` and $\mathtt{push}(s_{\mathcal{H}})$ instead of $\mathtt{pop}(\mathcal{S}_{\mathcal{H}})$ and $\mathtt{push}(s_{\mathcal{H}}, \mathcal{S}_{\mathcal{H}})$ when $\mathcal{S}_{\mathcal{H}}$ is clear from the context.

As expected, the pointer stack of a program is used when calling a method: references to the parameters are pushed on a new stack frame at the top of the

pointer stack. The pointer mappings of a pointer stack $\mathcal{S}_\mathcal{H}$ map method parameters to the references of the arguments on which they are applied, respecting the dynamic binding principle found in Object Oriented Languages.

Example 3. *For example, considering the method* `setQueue(BList q)` *defined in the class* `BList` *of Example 1, adding a method call* `d.setQueue(b);` *at the end of the initialization instruction of Example 2 will push a new stack frame* $\langle \texttt{void setQueue}^{\texttt{BList}}(\texttt{BList}), p_\mathcal{H}\rangle$ *on the pointer stack.* $p_\mathcal{H}(\texttt{q})$ *will point to* $p_\mathcal{H}(\texttt{b})$*, the node corresponding to the object computed by* `b`*, and* $p_\mathcal{H}(\texttt{this})$ *will point to* $p_\mathcal{H}(\texttt{d})$*, the node corresponding to the object computed by* `d`*.*

At the beginning of an execution, the pointer stack will only contain the stack frame $\langle \texttt{void main}^{\texttt{Exe}}(), p_0\rangle$; p_0 being a mapping associating each local variable in the main method to the `null` reference, whether it is of reference type, and to the basic primitive value otherwise (`false` for `boolean` and 0 for `int`). We will see shortly that a 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.

3.4. Memory configuration

A memory configuration $\mathcal{C} = \langle \mathcal{H}, \mathcal{S}_\mathcal{H}\rangle$ consists in a pointer graph $\mathcal{H}$ together with a pointer stack $\mathcal{S}_\mathcal{H}$. Among memory configurations, we distinguish the *initial configuration* $\mathcal{C}_0$ defined by $\mathcal{C}_0 = \langle(\{\&\texttt{null}\}, \emptyset), [\langle \texttt{void main}^{\texttt{Exe}}(), p_0\rangle]\rangle$ where `&null` is the reference of the `null` object (*i.e.* $l_V(\&\texttt{null}) = \texttt{null}$) and $[s_\mathcal{H}]$ denotes the pointer stack composed of only one stack frame $s_\mathcal{H}$.

In other words, the initial configuration is such that the pointer graph only contains the null reference as node and no arrows and a pointer stack with one frame for the `main` method call.

3.5. Meta-language and flattening

The semantics of AOO 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:

$$\begin{array}{rcl} \texttt{me} & ::= & \texttt{x} \mid \texttt{cst}_\tau \mid \texttt{null} \mid \texttt{this} \mid \texttt{op}(\overline{\texttt{x}}) \\ & & \mid \texttt{new C}(\overline{\texttt{x}}) \mid \texttt{y.m}(\overline{\texttt{x}}) \\ \\ \texttt{MI} & ::= & \texttt{;} \mid [\tau]\ \texttt{x:=me;} \mid \texttt{MI}_1\ \texttt{MI}_2 \mid \texttt{y.m}(\overline{\texttt{x}})\texttt{;} \\ & & \mid \texttt{while(x)\{MI\}} \mid \texttt{if(x)\{MI}_1\texttt{\}else\{MI}_2\texttt{\}} \\ & & \mid \texttt{pop;} \mid \texttt{push}(s_\mathcal{H})\texttt{;} \mid \epsilon \end{array}$$

where ϵ denotes the empty meta-instruction.

Flattening an instruction `I` into a meta-instruction $\underline{\texttt{I}}$ will consist in adding fresh intermediate variables for each complex expression parameter. This procedure is standard and defined in Figure 3. The flattened meta-instruction

$$\underline{[[\tau]\ \mathtt{x}{:=}\mathtt{e}];} = [[\tau]\ \mathtt{x}{:=}\mathtt{e}]; \quad \text{if } \mathtt{e} \in \mathbb{V} \cup dom(\mathbb{T}) \cup \{\mathtt{this}, \mathtt{null}\}$$

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

$$\underline{[\tau]\ \mathtt{x}{:=}\mathtt{new}\ \mathtt{C}(\mathtt{e}_1, \ldots, \mathtt{e}_n);} = \underline{\tau_1\ \mathtt{x}_1{:=}\mathtt{e}_1; \ldots \tau_n\ \mathtt{x}_n{:=}\mathtt{e}_n;}\ [\tau]\ \mathtt{x}{:=}\mathtt{new}\ \mathtt{C}(\mathtt{x}_1, \ldots, \mathtt{x}_n);$$

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

$$\underline{\mathtt{I}_1\ \mathtt{I}_2} = \underline{\mathtt{I}_1}\ \underline{\mathtt{I}_2}$$

$$\underline{\mathtt{while}(\mathtt{e})\{\mathtt{I}\}} = \underline{\mathtt{boolean}\ \mathtt{x}_1 := \mathtt{e};}\ \mathtt{while}(\mathtt{x}_1)\{\underline{\mathtt{I}}\ \underline{\mathtt{x}_1 := \mathtt{e};}\}$$

$$\underline{\mathtt{if}(\mathtt{e})\{\mathtt{I}_1\}\mathtt{else}\{\mathtt{I}_2\}} = \underline{\mathtt{boolean}\ \mathtt{x}_1 := \mathtt{e};}\ \mathtt{if}(\mathtt{x}_1)\{\underline{\mathtt{I}_1}\}\mathtt{else}\{\underline{\mathtt{I}_2}\}$$

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

Figure 3: Instruction flattening

will keep the semantics of the initial instruction unchanged. The main interest in such a program transformation is just that all the variables will be statically defined in a meta-instruction whereas they could be dynamically created by an instruction, hence allowing a cleaner (and easier) semantic treatment of meta-instructions. We extend the flattening to methods (and constructors) by $\tau\ \mathtt{m}(\tau_1\ \mathtt{x}_1, \ldots, \tau_n\ \mathtt{x}_n)\{\underline{\mathtt{I}}\ [\mathtt{return}\ \mathtt{x};]\}$ so that each instruction is flattened. A flattened program $\underline{P}$ is the program obtained by flattening all the instructions in the methods of a program P. Notice that the flattening of an AOO program is also an AOO program, as the flattening is a closed transformation with respect to the AOO syntax, and that the flattening is a polynomially bounded program transformation.

Lemma 1. *Define the size of an instruction $|\mathtt{I}|$ (respectively meta-instruction $|\mathtt{MI}|$) to be the number of symbols in $\mathtt{I}$ (resp. $\mathtt{MI}$). For each instruction $\mathtt{I}$, we have $|\underline{\mathtt{I}}| = O(|\mathtt{I}|^2)$.*

Proof. By induction on the definition of flattening. The flattening of each atomic instruction adds a number of new symbols that is at most linear in the size of the original instruction. The number of instructions is also linear in $|\mathtt{I}|$. □

Corollary 1. *Define the size of an AOO program $|P|$ to be the number of symbols in P. For each program P, we have $|\underline{P}| = O(|P|^2)$.*

An alternative choice would have been to restrict program syntax by requiring expressions to be flattened, thus avoiding the use of the meta-language. However such a choice would impact negatively the expressivity of this study. On another hand, one possibility might have been to generate local variables dynamically in the programming semantics but such a treatment makes the analysis of pointer mapping domain (i.e. the number of living variables) a very hard task to handle.

Example 4. *We add the method* `decrement()` *to the class* `BList` *of Example 1:*

```
void decrement() {
  if (value) {
    value := false;
  }
  else{
    if (queue != null) {
      value := true;
      queue.decrement();
    } else {
      value := false;
    }
  }
}
```

The program flattening will generate the following body for the flattened method:

```
void decrement() {
  if (value) {
    value := false;
  }
  else{
    BList x1 := null;
    BList x2 := queue;
    boolean x3 := x2 != x1
    if (x3) {
      value := true;
      queue.decrement();
    } else {
      value := false;
    }
  }
}
```

where $\mathtt{x}_1$, $\mathtt{x}_2$ *and* $\mathtt{x}_3$ *are fresh variables.*

3.6. Program semantics

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

Now, before defining the formal semantics of AOO programs, we introduce some preliminary notations.

Given a memory configuration $\mathcal{C} = \langle \mathcal{H}, \mathtt{push}(\langle \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau}), p_{\mathcal{H}} \rangle, \mathcal{S}_{\mathcal{H}}) \rangle$:

- $\top\mathcal{C} = p_{\mathcal{H}}$

$$(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{y}; \mathtt{MI}) \to (\mathcal{C}[\mathtt{x} \mapsto \mathcal{C}(\mathtt{y})], \mathtt{MI}) \quad \textit{if } \mathtt{x} \notin \mathbb{C}(\mathcal{C}).\mathcal{F} \tag{1}$$

$$(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{y}; \mathtt{MI}) \to (\mathcal{C}[\mathcal{C}(\mathtt{this}) \overset{\mathtt{x}}{\mapsto} \mathcal{C}(\mathtt{y})], \mathtt{MI}) \quad \textit{if } \begin{cases} \mathtt{x} \in \mathbb{C}(\mathcal{C}).\mathcal{F} \\ \mathcal{C}(\mathtt{this}) \neq \mathtt{null} \end{cases} \tag{2}$$

$$(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{y}; \mathtt{MI}) \to (\mathcal{C}, \mathtt{MI}) \quad \textit{if } \mathtt{x} \in \mathbb{C}(\mathcal{C}).\mathcal{F} \textit{ and } \mathcal{C}(\mathtt{this}) = \mathtt{null} \tag{3}$$

$$(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{cst}_\tau; \mathtt{MI}) \to (\mathcal{C}[\mathtt{x} \mapsto \mathtt{cst}_\tau], \mathtt{MI}) \tag{4}$$

$$(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{null}; \mathtt{MI}) \to (\mathcal{C}[\mathtt{x} \mapsto\ \&\mathtt{null}], \mathtt{MI}) \tag{5}$$

$$(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{this}; \mathtt{MI}) \to (\mathcal{C}[\mathtt{x} \mapsto \mathcal{C}(\mathtt{this})], \mathtt{MI}) \tag{6}$$

$$(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{op}(\overline{\mathtt{y}}); \mathtt{MI}) \to (\mathcal{C}[\mathtt{x} \mapsto [\![\mathtt{op}]\!](\overline{\mathcal{C}(\mathtt{y})})], \mathtt{MI}) \tag{7}$$

$$(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{new}\ \mathtt{C}(\overline{\mathtt{y}}); \mathtt{MI}) \to (\mathcal{C}[v \mapsto \mathtt{C}][v \overset{\mathtt{z}_i}{\mapsto} \mathcal{C}(\mathtt{y}_i)][\mathtt{x} \mapsto v], \mathtt{MI}) \tag{8}$$

where v is a fresh node (memory reference) and $\mathtt{C}.\mathcal{F} = \{\mathtt{z}_1, \ldots, \mathtt{z}_n\}$

$$(\mathcal{C},\ ; \mathtt{MI}) \to (\mathcal{C}, \mathtt{MI}) \tag{9}$$

$$(\mathcal{C}, [\mathtt{u}{:=}]\mathtt{x}.\mathtt{m}(\overline{\mathtt{y}}); \mathtt{MI}) \to (\mathcal{C}, \tag{10}$$

$$\mathtt{push}(\langle \tau\ \mathtt{m}^{\mathtt{C}\star}(\overline{\tau}), \top\mathcal{C}[\mathtt{this} \mapsto \mathcal{C}(\mathtt{x}), \mathtt{z}_i \mapsto \mathcal{C}(\mathtt{x}.\mathtt{z}_i), \mathtt{x}_i \mapsto \mathcal{C}(\mathtt{y}_i)] \rangle);$$

$$\mathtt{MI}'\ [\mathtt{u}{:=}\mathtt{x}';]\ \mathtt{pop};\ \mathtt{MI})$$

if $\mathtt{m}$ is a flattened method $\tau\ \mathtt{m}(\tau_1\ \mathtt{x}_1, \ldots \tau_n\ \mathtt{x}_n)\{\mathtt{MI}'\ [\mathtt{return}\ \mathtt{x}';]\}$ in $\mathtt{C}\star$ and $\mathtt{C}.\mathcal{F} = \{\mathtt{z}_1, \ldots, \mathtt{z}_n\}$

$$(\mathcal{C}, \mathtt{push}(s_\mathcal{H}); \mathtt{MI}) \to (\mathcal{C}[\mathtt{push}(s_\mathcal{H})], \mathtt{MI}) \tag{11}$$

$$(\mathcal{C}, \mathtt{pop}; \mathtt{MI}) \to (\mathcal{C}[\mathtt{pop}], \mathtt{MI}) \tag{12}$$

$$(\mathcal{C}, \mathtt{while}(\mathtt{x})\{\mathtt{MI}'\}\ \mathtt{MI}) \to (\mathcal{C},\ \mathtt{MI}'\ \mathtt{while}(\mathtt{x})\{\mathtt{MI}'\}\ \mathtt{MI}) \quad \text{if } \mathcal{C}(\mathtt{x}) = \mathtt{true} \tag{13}$$

$$(\mathcal{C}, \mathtt{while}(\mathtt{x})\{\mathtt{MI}'\}\ \mathtt{MI}) \to (\mathcal{C},\ \mathtt{MI}) \quad \text{if } \mathcal{C}(\mathtt{x}) = \mathtt{false} \tag{14}$$

$$(\mathcal{C}, \mathtt{if}(\mathtt{x})\{\mathtt{MI}_{\mathtt{true}}\}\mathtt{else}\{\mathtt{MI}_{\mathtt{false}}\}\ \mathtt{MI}) \to (\mathcal{C}, \mathtt{MI}_{\mathcal{C}(\mathtt{x})}\ \mathtt{MI}) \tag{15}$$

Figure 4: Semantics of AOO programs

- $\mathcal{C}(\mathtt{x}) = p_\mathcal{H}(\mathtt{x})$
- $\mathcal{C}(\mathtt{x}.\mathtt{z}) = v$, with $v \in \mathcal{H}$ such that $\mathcal{C}(\mathtt{x}) \overset{\mathtt{z}}{\mapsto} v \in \mathcal{H}$
- $\mathbb{C}(\mathcal{C}) = \mathtt{C}$
- $\mathcal{C}[\mathtt{x} \mapsto v] = \langle \mathcal{H}, \mathtt{push}(\langle \tau \mathtt{m}^{\mathtt{C}}(\overline{\tau}), p_\mathcal{H}[\mathtt{x} \mapsto v] \rangle, \mathcal{S}_\mathcal{H}) \rangle$, $v \in V \cup dom(\mathbb{T})$
- $\mathcal{C}[\mathtt{push}(s_\mathcal{H})] = \langle \mathcal{H}, \mathtt{push}(s_\mathcal{H}, \mathtt{push}(\langle \tau \mathtt{m}^{\mathtt{C}}(\overline{\tau}), p_\mathcal{H} \rangle, \mathcal{S}_\mathcal{H})) \rangle$
- $\mathcal{C}[\mathtt{pop}] = \langle \mathcal{H}, \mathcal{S}_\mathcal{H} \rangle$

In other words, $\top\mathcal{C}$ is a shorthand for the pointer mapping at the top of the stack $\mathtt{push}(\langle \tau \mathtt{m}^{\mathtt{C}}(\overline{\tau}), p_\mathcal{H} \rangle, \mathcal{S}_\mathcal{H})$. $\mathcal{C}(\mathtt{x})$ is a shorthand notation for $\top\mathcal{C}(\mathtt{x})$. $\mathcal{C}(\mathtt{x}.\mathtt{z})$ is the memory reference of the field $\mathtt{z}$ of the object stored in $\mathtt{x}$. $\mathbb{C}(\mathcal{C})$ is the class

of the current object under evaluation in the memory configuration. $\mathcal{C}[\mathtt{x} \mapsto v]$, $v \in V \cup dom(\mathbb{T})$, is a notation for the memory configuration $\mathcal{C}'$ that is equal to $\mathcal{C}$ but on $\top\mathcal{C}(\mathtt{x})$ where the value is updated to v. $\mathcal{C}[\mathtt{push}(s_{\mathcal{H}})]$ and $\mathcal{C}[\mathtt{pop}]$ are notations for the memory configuration where the frame $s_{\mathcal{H}}$ has been pushed to the top of the pointer stack and where the top pointer mapping has been removed from the top of the stack, respectively.

Finally, let $\mathcal{C}[v \mapsto \mathtt{C}]$, $v \in V$ denote a memory configuration $\mathcal{C}'$ whose graph contains the new node v labeled by $\mathtt{C}$ (i.e. $l_V(v) = \mathtt{C}$) and let $\mathcal{C}[v \overset{\mathtt{x}}{\mapsto} w]$, $v, w \in V$, denote a memory configuration $\mathcal{C}'$ whose pointer graph contains the new arrow (v, w) labeled by $\mathtt{x}$ (*i.e.* $l_A((v, w)) = \mathtt{x}$). In the case where there was already some arrow (v, u) labeled by $\mathtt{x}$ (*i.e.* $l_A((v, u)) = \mathtt{x}$) in the graph then it is deleted and replaced by (v, w).

Each operator $\mathtt{op} \in \mathbb{O}$ of signature $\mathtt{op} :: \tau_1 \times \cdots \times \tau_n \to \tau_{n+1}$ is assumed to compute a total function $[\![\mathtt{op}]\!] : \tau_1 \times \cdots \times \tau_n \to \tau_{n+1}$ fixed by the language implementation. Operators will be discussed in more details in Subsection 4.2.

The formal rules of the small step semantics are provided in Figure 4. Let us explain the meaning of these rules.

- Rules (1) to (7) are fairly standard rules only updating the top pointer mapping $\top\mathcal{C}$ of the considered configuration but do not alter the other parts of the pointer stack. Rule (2) is the only rule altering the pointer graph. Rule (1) describes the assignment of a variable to another (that is not a field of the current object). Rule (2) describes a field assignment when the object is not null. Notice that in this case, the pointer graph is updated as the reference node of the current object points to the reference node of the assigned variable. Rule (3) indicates that if the object is null, assigning its fields results in skipping the instruction. Rule (4) is the assignment of a primitive constant to a variable. Rule (5) is the assignment of the null reference &`null` to a variable. Rule (6) consists in the assignment of the self-reference. Notice that such an assignment may only occur in a method body (because of well-formedness assumptions) and consequently the stack is non-empty and must contain a reference to `this`. Rule (7) describes operator evaluation based on the prior knowledge of function $[\![\mathtt{op}]\!]$.

- Rule (8) consists in the creation of a new instance. Consequently, this rule adds a new node v of label $\mathtt{C}$ and the corresponding arrows $(v, \mathcal{C}(\mathtt{y}_i))$ of label $\mathtt{z}_i$ in the pointer graph of $\mathcal{C}$. $\mathcal{C}(\mathtt{y}_i)$ are the nodes of the graph (or the primitive values) corresponding to the parameters of the constructor call and $\mathtt{z}_i$ is the corresponding field name in the class $\mathtt{C}$. Finally, this rule adds a link from the variable $\mathtt{x}$ to the new reference v in the pointer mapping $\top\mathcal{C}$.

- Rule (9) just consists in the evaluation of the empty instruction ';'.

- Rule (10) consists in a call to method $\mathtt{m}$ of class $\mathtt{C}$, provided that $\mathtt{z}$ is of class $\mathtt{C}$, dynamically for overrides. If $\mathtt{m}$ is not defined in $\mathtt{C}$ then it checks

for a method of the same signature in the upper class `D`, and so on. Let `C`$\star$ be a notation for the least superclass of `C` in which `m` is defined. It adds a new instruction for pushing a new stack frame on the stack, containing references of the current object `this` on which `m` is applied, references of each field, 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. The `pop` instruction is crucial to indicate the end of the method body so that the callee knows when to return the control flow to the caller.

- Rules (11) and (12) are standard rules for manipulating the pointer stack through the use of `pop` and `push` instructions.
- Rules (13) to (15) are standard rules for control flow statements.

One important point to stress is that, contrarily to usual OO languages, the above semantics can reduce when a method is called on a variable pointing to the value (see Rule (10)). In this particular case, all assignments to the fields of the current object are ignored (see Rule (3)). This choice has been made to simplify the formal treatment as exceptions are not included in AOO syntax. In general, exception handling generate a new control flow that is tedious to handle with a small step semantics. More, our typing discipline relies on the control flow being smooth.

3.7. Input and Size

In order to analyze the complexity of programs, it is important to relate the execution of a given program to the size of its input. Consequently, we need to define both the notion of input of an AOO program and the notion of size for such an input. This will make clear the choice made in Section 2 to express an executable class by splitting the instruction of the `main` method in an *initialization instruction* `Init` and a *computational instruction* `Comp`.

An AOO program of executable `Exe{void main(){Init Comp}}` terminates if the following conditions hold:

1. $(\mathcal{C}_0, \underline{\texttt{Init}}) \rightarrow^* (\mathcal{I}, \epsilon)$
2. $(\mathcal{I}, \underline{\texttt{Comp}}) \rightarrow^* (\mathcal{C}, \epsilon)$

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

An important point to stress is that, given a program, the choice of initialization and computational instructions is left to the analyzer. This choice is crucial for this analysis to be relevant. Note also that the initialization instruction `Init` can be changed so that the program may be defined on any input.

Another way to handle the notion of input could be to allow i/o (open a stream on a file, deserialization,...) in this instruction but at the price of a more technical and not that interesting treatment. A last possibility is to consider the input to be the main method arguments. But this would mean that as most of

the programs do not use their prompt line argument, they would be considered to be constant time programs.

There are two particular cases:

- If the initialization instruction is empty then there will be no computation on reference type variables apart from constant time or non-terminating ones, as we will see shortly. This behavior is highly expected as it means that the program has no input. As there is no input, it means that either the program does not terminate or it terminates in constant time.
- If the computational instruction is empty (that is ";") then the program will trivially pass the complexity analysis.

Example 5. *Consider the initialization instruction* `Init` *of Example 2.* `Init` *is already flattened so that* $\underline{\texttt{Init}} = \texttt{Init}$ *holds. The input* $\mathcal{I}$ *is a memory configuration* $\langle \mathcal{H}, \mathcal{S}_\mathcal{H} \rangle$ *such that* $\mathcal{H}$ *is the pointer graph described in Figure 2 and* $\mathcal{S}_\mathcal{H} = [\langle \texttt{void main}^{\texttt{Exe}}(), p_\mathcal{H} \rangle]$, $p_\mathcal{H}$ *being the pointer mapping described by the snake arrows of Figure 2.*

Now consider the computational instruction $\texttt{Comp} = \texttt{d.setQueue(b);}$. *It is also flat and so is the method body. Consequently, we have:*

$$\begin{aligned}
(\mathcal{I}, \texttt{d.setQueue(b);}) &\to (\mathcal{I}, \texttt{push}(s_\mathcal{H}); \texttt{queue:=q; pop;}) \\
&\to (\mathcal{I}[\texttt{push}(s_\mathcal{H})], \texttt{queue:=q; pop;}) \\
&\to (\mathcal{I}'[\mathcal{I}'(\texttt{this}) \overset{\texttt{queue}}{\mapsto} \mathcal{I}'(\texttt{q})], \texttt{pop;}) \\
&= (\mathcal{I}'[\mathcal{I}(\texttt{d}) \overset{\texttt{queue}}{\mapsto} \mathcal{I}(\texttt{b})], \texttt{pop;}) \\
&\to (\mathcal{I}'[\mathcal{I}(\texttt{d}) \overset{\texttt{queue}}{\mapsto} \mathcal{I}(\texttt{b})][\texttt{pop}], \epsilon) \\
&= (\mathcal{I}[\mathcal{I}(\texttt{d}) \overset{\texttt{queue}}{\mapsto} \mathcal{I}(\texttt{b})], \epsilon)
\end{aligned}$$

with $s_\mathcal{H} = \langle \texttt{void setQueue}^{\texttt{BList}}(\texttt{BList}), \top\mathcal{I}[\texttt{this} \mapsto \mathcal{I}(\texttt{d}), \texttt{q} \mapsto \mathcal{I}(\texttt{b})] \rangle$ *and* $\mathcal{I}' = \mathcal{I}[\texttt{push}(s_\mathcal{H})]$. *The two first transitions are an application of Rules (8) and (9) of Figure 4. The third one is Rule (14) as* `queue` *is a field of the current object. The first equality holds by definition of* $\mathcal{I}'$ *and its top stack frame* $\top\mathcal{I}' = s_\mathcal{H}$. *The last reduction is Rule (10) and the equality holds as removing the top stack of* $\mathcal{I}'$ *leads to* $\mathcal{I}$*, keeping in mind that the pointer graph has been updated by* $[\mathcal{I}(\texttt{d}) \overset{\texttt{queue}}{\mapsto} \mathcal{I}(\texttt{b})]$. *Indeed now the node referenced by* `d` *points to* `b` *via the arrow labeled by* `queue`.

Definition 1 (Sizes)**.** *The size:*

1. $|\mathcal{H}|$ *of a pointer graph* $\mathcal{H} = (V, A)$ *is defined to be the number of nodes in* V.
2. $|p_\mathcal{H}|$ *of a pointer mapping* $p_\mathcal{H}$ *is defined by* $|p_\mathcal{H}| = \sum_{\texttt{x} \in dom(p_\mathcal{H})} |p_\mathcal{H}(\texttt{x})|$, *where the size of numerical primitive value is the value itself, the size of a boolean is* 1 *and the size of a memory reference is* 1.
3. $|s_\mathcal{H}|$ *of a stack frame* $s_\mathcal{H} = \langle s, p \rangle$ *is defined by:* $|s_\mathcal{H}| = 1 + |p|$.
4. $|\mathcal{S}_\mathcal{H}|$ *of a stack* $\mathcal{S}_\mathcal{H}$ *is defined by* $|\mathcal{S}_\mathcal{H}| = \sum_{s_\mathcal{H} \in \mathcal{S}_\mathcal{H}} |s_\mathcal{H}|$.

5. $|\mathcal{C}|$ *of a memory configuration* $\mathcal{C} = \langle \mathcal{H}, \mathcal{S}_\mathcal{H} \rangle$ *is defined by* $|\mathcal{C}| = |\mathcal{H}| + |\mathcal{S}_\mathcal{H}|$.

In Item 1, it suffices to bound the number of nodes as, for the considered multigraphs, $|A| = \mathcal{O}(|V|)$. Indeed, each node has a number of out arrows bounded by a constant (the maximal number of fields in all classes of a given program). In Item 2, numerical primitive values are not considered to be constant. This definition is robust if we consider their size to be constant: e.g. a signed 32 bit integer could be considered as a constant smaller than $2^{31} - 1$. In such a case, the size of each pointer mapping would be constant as no fresh variable can be generated within a program thanks to flattening. In Item 4, the size of a pointer stack is very close to the size of the usual OO Virtual Machine stack since it counts the number of nested method calls (i.e. the number of stack frames in the stack) and the size of primitive data in each frame (that are duplicated during the pass-by-value evaluation). Finally, Item 5 defines the size of a memory configuration, the program input and bounds both the heap size of the input $\mathcal{I}$ and *a fortiori* the stack size as the stack is empty in $\mathcal{I}$.

Negative integers are not considered for simplicity reasons (see section 4.2 where we need data types to have a lower bound). Floats are not considered as there is an infinite number of floats of the same size. Only finite types and countable types can be treated. To our knowledge this point is not tackled by the related works on ICC.

3.8. Compatible pairs

Given a memory configuration $\mathcal{C}$ and a meta-instruction $\texttt{MI}$, the pair $(\mathcal{C}, \texttt{MI})$ is compatible if there exists an instruction $\texttt{MI}'$ from a well-formed program such that $(\mathcal{C}_0, \texttt{MI}') \to^* (\mathcal{C}, \texttt{MI})$. Throughout the paper, we will only consider compatible pairs $(\mathcal{C}, \texttt{MI})$.

This notion is introduced in order to prevent the consideration of a pair $(\mathcal{C}, \texttt{MI})$ having a variable not defined in the memory configuration $\mathcal{C}$ and called without being declared in the meta-instruction $\texttt{MI}$.

Note that the semantics of Section 3.6 cannot reach uncompatible pairs as each variable is supposed to be declared before its first use by definition of well-formed programs. However, this restriction will be required in order to prevent bad configurations from being considered in Theorem 1.

4. Type system

In this section, a tier based type system for ensuring polynomial time and polynomial space upper bounds on the size of a memory configuration is introduced.

4.1. Tiered types

The set of base types is defined to be the set of all primitive and reference types $\mathbb{C} \cup \mathbb{T}$. *Tiers* are 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 $\preceq$, being such that $\mathbf{0} \preceq \mathbf{1}$. Given a sequence of tiers $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$ define $\vee\overline{\alpha} = \alpha_1 \vee \ldots \vee \alpha_n$. Let $\alpha, \beta, \gamma, \ldots$ denote tiers in $\{\mathbf{0}, \mathbf{1}\}$.

A *tiered type* is a pair $\tau(\alpha)$ consisting of a type $\tau \in \mathbb{C} \cup \mathbb{T}$ together with a tier $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

Notations. Given two sequences of types $\overline{\tau} = \tau_1, \ldots, \tau_n$ and tiers $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$ and a tier α, let $\overline{\tau}(\overline{\alpha})$ denote $\tau_1(\alpha_1), \ldots, \tau_n(\alpha_n)$, $\overline{\tau}(\alpha)$ denote $\tau_1(\alpha), \ldots, \tau_n(\alpha)$ and $\langle\overline{\tau}\rangle$ (resp. $\langle\overline{\tau}(\overline{\alpha})\rangle$) denote the cartesian product of types (resp. tiered types).

For example, given a sequence of types $\overline{\tau} = \mathtt{boolean}, \mathtt{char}, \mathtt{C}$ and a sequence of tiers $\overline{\alpha} = \mathbf{0}, \mathbf{1}, \mathbf{1}$, we have $\overline{\tau}(\mathbf{0}) = \mathtt{boolean}(\mathbf{0}), \mathtt{char}(\mathbf{0}), \mathtt{C}(\mathbf{0})$, $\overline{\tau} = \mathtt{boolean} \times \mathtt{char} \times \mathtt{C}$ and $\langle\overline{\tau}(\overline{\alpha})\rangle = \mathtt{boolean}(\mathbf{0}) \times \mathtt{char}(\mathbf{1}) \times \mathtt{C}(\mathbf{1})$.

What is a tier in essence. Tiers will be used to separate data in two kinds as in Bellantoni and Cook's safe recursion scheme [1] where data are divided into "safe" and "normal" data kinds. According to Danner and Royer [9], "normal data [are the data] that drive recursions and safe data [are the data] over which recursions compute". In this setting, tier **1** will be an equivalent for normal data type, as it consists in data that drive recursion and while loops. Tier **0** will be the equivalent for safe data type, as it consists in computational data storages. Instruction tiers will ensure that expressions of the right tier are used at the right place (e.g. tier **0** data will never drive a loop or a recursion) and that the information flow never go from **1** to **0** so that the first condition is preserved independently of tier **0** during program execution. In particular, a tier **1** instruction will never be controlled by a tier **0** instruction (in a conditional statement or recursive call).

4.2. Operators

In order to control the complexity of programs, we need to constrain the admissible tiered types of operators depending on their computational power. For that purpose and following [10], we define *neutral operators* whose computation does not make the size increase and *positive operators* whose computation may make the size increase by some constant. They are both assumed to be polynomial time computable.

Definition 2. *An operator* $\mathtt{op} :: \tau_1 \times \ldots \times \tau_n \to \tau$ *is:*

1. neutral *if* $[\![\mathtt{op}]\!]$ *is a polynomial time computable function and one of the following conditions hold:*
   (a) *either* $\tau = \mathtt{boolean}$ *or* $\tau = \mathtt{char}$,
   (b) $\mathtt{op} :: \mathtt{int} \times \ldots \times \mathtt{int} \to \mathtt{int}$ *and* $\forall \mathtt{i} \in [1, n]$, $\forall \mathtt{cst}^{\mathtt{i}}_{\mathtt{int}}$:

$$0 \leq [\![\mathtt{op}]\!](\mathtt{cst}^{1}_{\mathtt{int}}, \ldots, \mathtt{cst}^{\mathtt{n}}_{\mathtt{int}}) \leq \max_{\mathtt{j}} \mathtt{cst}^{\mathtt{j}}_{\mathtt{int}}.$$

2. positive *if it is not neutral,* $[\![\mathtt{op}]\!]$ *is a polynomial time computable function,* $\mathtt{op} :: \mathtt{int} \times \ldots \times \mathtt{int} \to \mathtt{int}$*, and:*

$$\forall \mathtt{i} \in [1, \mathtt{n}],\ \forall \mathtt{cst}^{\mathtt{i}}_{\mathtt{int}},\ 0 \leq [\![\mathtt{op}]\!](\mathtt{cst}^{1}_{\mathtt{int}}, \ldots, \mathtt{cst}^{\mathtt{n}}_{\mathtt{int}}) \leq \max_{\mathtt{j}} \mathtt{cst}^{\mathtt{j}}_{\mathtt{int}} + \mathtt{c},$$

for some constant $\mathtt{c} \geq 0$*.*

In the above definition, Item 1a could be extended to any primitive data type inhabited by a finite number of values.

In Items 1b and 2, the comparison is only defined whenever all the τ_i and τ are of type `int`. This could be extended to booleans without any trouble by considering a boolean to be 0 or 1.

Also notice that operator overload can be considered by just treating two such operators as if they were distinct ones.

Example 6. *For simplicity, operators are always used in a prefix notation. The operator* $-$*, whose semantics is such that* $[\![-]\!](x, y) = \max(x - y, 0)$*, is neutral. Indeed for all* $\mathtt{cst}^1_{\mathtt{int}}$*,* $\mathtt{cst}^2_{\mathtt{int}}$*,* $[\![-]\!](\mathtt{cst}^1_{\mathtt{int}}, \mathtt{cst}^2_{\mathtt{int}}) = \max(\mathtt{cst}^1_{\mathtt{int}} - \mathtt{cst}^2_{\mathtt{int}}, 0) \leq \max(\mathtt{cst}^1_{\mathtt{int}}, \mathtt{cst}^2_{\mathtt{int}})$*. The operators* $<$*, testing that its left operand is strictly smaller than its right one, and the operator* `==`*, testing the equality of primitive values or the equality of memory references, and the operator* $!=$ *of Example 4 are neutral operators as their output is* `boolean` *and they are computable in polynomial time.*

The operator $+$ *is neither neutral, nor positive as* $[\![+]\!](\mathtt{cst}^1_{\mathtt{int}}, \mathtt{cst}^2_{\mathtt{int}}) = \mathtt{cst}^1_{\mathtt{int}} + \mathtt{cst}^2_{\mathtt{int}}$ *and there is no constant* $\mathtt{c} \geq 0$ *such that* $\forall \mathtt{cst}^1_{\mathtt{int}}, \mathtt{cst}^2_{\mathtt{int}},\ \mathtt{cst}^1_{\mathtt{int}} + \mathtt{cst}^2_{\mathtt{int}} \leq \mathtt{cst}^i_{\tau_i} + \mathtt{c}$*. However if we consider its partial evaluation* $+n$ *to be an operator, then* $+n$ *is a positive operator as* $\forall \mathtt{cst}^1_{\mathtt{int}},\ [\![+n]\!](\mathtt{cst}^1_{\mathtt{int}}) = \mathtt{cst}^1_{\mathtt{int}} + n \leq \mathtt{cst}^i_{\tau_i} + \mathtt{c}$*. Indeed, just take* $\mathtt{c} \geq n$*. The reason behind such a strange distinction is that a call of the shape* $\mathtt{x} := \mathtt{y} + \mathtt{y}$ *could lead to an exponential computation in a loop while a call of the shape* $\mathtt{x} := \mathtt{y} + n$ *should remain polynomial (under some restrictions on the loop).*

As mentioned above, the notions of neutral and positive operators can be extended to operators with numerical output and boolean input. Hence allowing to consider operators such as `x?y:z` *returning* `y` *or* `z` *depending on whether* `x` *is* `true` *or* `false` *when applied to numerical primitive data. Indeed, we have* $[\![\mathtt{?:}]\!](\mathtt{x}, \mathtt{y}, \mathtt{z}) \leq \max(\mathtt{y}, \mathtt{z})$*. Consequently,* `?:` *is a neutral operator.*

Definition 3. *An* operator typing environments Ω *is a mapping such that each operator* $\mathtt{op} :: \langle \overline{\tau} \rangle \to \tau$ *is assigned the type:*

- $\Omega(\mathtt{op}) = \{\langle \overline{\tau}(\mathbf{0}) \rangle \to \tau(\mathbf{0}), \langle \overline{\tau}(\mathbf{1}) \rangle \to \tau(\mathbf{1})\}$ *if* `op` *is a neutral operator,*
- $\Omega(\mathtt{op}) = \{\langle \overline{\tau}(\mathbf{0}) \rangle \to \tau(\mathbf{0})\}$ *if* `op` *is a positive operator,*
- $\Omega(\mathtt{op}) = \emptyset$ *otherwise.*

4.3. Environments

In this subsection, we define three other kinds of environments:

- *method typing environments* δ associating a tiered type to each program variable $v \in \mathbb{V}$,
- *typing environments* Δ associating a typing environment δ to each method signature $\tau\ \mathtt{m^C}(\overline{\tau})$, i.e. $\Delta(\tau\ \mathtt{m^C}(\overline{\tau})) = \delta$ (by abuse of notation, we will sometime use the notation $\Delta(\mathtt{m^C})$ when the method signature is clear from the context),
- *contextual typing environments* $\Gamma = (s, \Delta)$, a pair consisting of a method signature and a typing environment. The method (or constructor) signature s indicates under which context (typing environment) the fields are typed.

For each $\mathtt{x} \in \mathbb{V}$, define $\Gamma(\mathtt{x}) = \Delta(s)(\mathtt{x})$. Also define $\Gamma\{\mathtt{x} \leftarrow \tau(\alpha)\}$ to be the contextual typing environment Γ' such that $\forall \mathtt{y} \neq \mathtt{x},\ \Gamma'(\mathtt{y}) = \Gamma(\mathtt{y})$ and $\Gamma'(\mathtt{x}) = \tau(\alpha)$. Let $\Gamma\{s'\}$ be a notation for the contextual typing environment that is equal to (s', Δ), *i.e.* the signature s' has been substituted to s in Γ. Method typing environment are defined for method signatures and can be extended without any difficulty to constructor signatures.
What is the purpose of contextual environments. They will allow the type system to type a field with distinct tiered types depending on the considered method. Indeed, a field can be used in the guard of a while loop (and will be of tier **1**) in some method whereas it can store freshly created objects (and will be of tier **0**) in some other method. This is the reason why the presented type system has to keep information on the context.

4.4. Judgments

Expressions and instructions will be typed using tiered types. A method of arity n method will be given a type of the shape $\mathtt{C}(\beta) \times \tau_1(\alpha_1) \times \ldots \times \tau_n(\alpha_n) \rightarrow \tau(\alpha)$. Consequently, given a contextual typing environment Γ, there are four kinds of typing judgments:

- $\Gamma, \Omega \vdash_\beta \mathtt{e} : \tau(\alpha)$ for expressions, meaning that the expression $\mathtt{e}$ is of tiered type $\tau(\alpha)$ under the contextual typing environment Γ and operator typing environment Ω and can only be assigned to in an instruction of tier at least β,
- $\Gamma, \Omega \vdash \mathtt{I} : \mathtt{void}(\alpha)$ for instructions, meaning that the instruction $\mathtt{I}$ is of tiered type $\mathtt{void}(\alpha)$ under the contextual typing environment Γ and operator typing environment Ω,
- $\Gamma, \Omega \vdash_\beta s : \mathtt{C}(\beta) \times \langle \overline{\tau}(\overline{\alpha}) \rangle \rightarrow \tau(\alpha)$ for method signatures, meaning that the method $\mathtt{m}$ of signature s belongs to the class $\mathtt{C}$ ($\mathtt{C}(\beta)$ is the tiered type of the current object `this`), has parameters of type $\langle \overline{\tau}(\overline{\alpha}) \rangle$, has a return variable of type $\tau(\alpha)$, with $\tau = \mathtt{void}$ in the particular case where there is

no return statement, and can only be called in instructions of tier at least β,

- $\Gamma, \Omega \vdash \mathtt{C}(\overline{\tau}) : \langle \overline{\tau}(\mathbf{0}) \rangle \to \mathtt{C}(\mathbf{0})$ for constructor signatures, meaning that the constructor `C` has parameters of type $\langle \overline{\tau}(\mathbf{0}) \rangle$ and a return variable of type $\mathtt{C}(\mathbf{0})$, matching the class type `C`.

Given a sequence $\overline{\mathtt{e}} = \mathtt{e}_1, \ldots, \mathtt{e}_n$ of expressions, a sequence of types $\overline{\tau} = \tau_1, \ldots, \tau_n$ and two sequences of tiers $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$ and $\overline{\beta} = \beta_1, \ldots, \beta_n$, the notation $\Gamma, \Omega \vdash_{\overline{\beta}} \overline{\mathtt{e}} : \overline{\tau}(\overline{\alpha})$ means that $\Gamma, \Omega \vdash_{\beta_i} \mathtt{e}_i : \tau_i(\alpha_i)$ holds, for all $i \in [1, n]$.

4.5. Typing rules

The intuition of the typing discipline is as follows: keeping in mind, that tier **1** corresponds to while loop guards data and that tier **0** corresponds to data storages (thus possibly increasing data), the type system precludes flows from tier **0** data to tier **1** data.

Most of the rules are basic non-interference typing rules following Volpano *et al.* type discipline: tiers in the rule premises (when there is one) are equal to the tier in the rule conclusion so that there can be no information flow (in both directions) using these rules.

4.5.1. Expressions typing rules

The typing rules for expressions are provided in Figure 5. They can be explained briefly:

- Primitive constants and the null reference can be given any tier α as they have no computational power (Rules *(Cst)* and *(Null)*) and can be used in instructions of any tier β. As in Java and for polymorphic reasons, `null` can be considered of any class `C`.

- Rule *(Var)* just consists in checking a variable tiered type with respect to a given contextual typing environment. This is the rule allowing a polymorphic treatment of fields depending on the context. Indeed, remember that $\Gamma(\mathtt{x})$ is a shorthand notation for $\Delta(s)(\mathtt{x})$. Moreover, a variable can be used in instructions of any tier β.

- Rule *(Op)* just consists in checking that the type of an operator with respect to a given operator typing environment matches the input and output tiered types. The operator output has to be used in an instruction of tier being at least the maximal admissible tier of its operand $\vee\overline{\beta} = \beta_1 \vee \ldots \vee \beta_n$. In other words, if one operand can only be used in a tier **1** instruction then the operator output can only be used in a tier **1** instruction.

- The rule *(Self)* makes explicit that the self reference `this` is of type `C` and enforces the tier of the fields to be equal to the tier of the current object, thus preventing "flows by references" in the heap. Moreover it can be used in instructions of any tier β.

- The rule *(Pol)* allows us to type a given expression with the superclass type while keeping the tiers unchanged.
- The rule *(New)* checks that the constructor arguments and output all have tier **0**. The new instance has to be of tier **0** since its creation makes the memory grow (a new reference node is added in the heap). The constructor arguments have also to be of tier **0**. Otherwise a flow from tier **0**, the new instance, to tier **1**, one of its fields, might occur. The explanation for the annotation instruction tier β is the same as the one for operators.
- Rule *(C)* just checks a direct type correspondence between the arguments types and the method type when a method is called. However this rule is very important as it allows a polymorphic type discipline for fields. Indeed the contextual environment is updated so that a field can be typed with respect to the considered method. The explanation for the annotation instruction tier β is the same as the one for operators. We will see shortly how to make restriction on such annotations in order to restrict the complexity of recursive method calls.

$$\frac{}{\Gamma, \Omega \vdash_\beta \mathtt{cst}_\tau : \tau(\alpha)}\ \textit{(Cst)} \qquad \frac{}{\Gamma, \Omega \vdash_\beta \mathtt{null} : \mathtt{C}(\alpha)}\ \textit{(Null)}$$

$$\frac{\Gamma(\mathtt{x}) = \tau(\alpha)}{\Gamma, \Omega \vdash_\beta \mathtt{x} : \tau(\alpha)}\ \textit{(Var)}$$

$$\frac{\Gamma, \Omega \vdash_{\overline{\beta}} \overline{\mathtt{e}} : \overline{\tau}(\alpha) \qquad \langle \overline{\tau}(\alpha) \rangle \to \tau(\alpha) \in \Omega(\mathtt{op})}{\Gamma, \Omega \vdash_{\vee\overline{\beta}} \mathtt{op}(\overline{\mathtt{e}}) : \tau(\alpha)}\ \textit{(Op)}$$

$$\frac{\forall \mathtt{x} \in \mathtt{C}.\mathcal{F}, \exists \tau, \Gamma(\mathtt{x}) = \tau(\alpha)}{\Gamma, \Omega \vdash_\beta \mathtt{this} : \mathtt{C}(\alpha)}\ \textit{(Self)} \qquad \frac{\Gamma, \Omega \vdash_\beta \mathtt{e} : \mathtt{D}(\alpha) \quad \mathtt{D} \trianglelefteq \mathtt{C}}{\Gamma, \Omega \vdash_\beta \mathtt{e} : \mathtt{C}(\alpha)}\ \textit{(Pol)}$$

$$\frac{\Gamma, \Omega \vdash_{\overline{\beta}} \overline{\mathtt{e}} : \overline{\tau}(\mathbf{0}) \quad \Gamma\{\mathtt{C}(\overline{\tau})\}, \Omega \vdash \mathtt{C}(\overline{\tau}) : \langle \overline{\tau}(\mathbf{0}) \rangle \to \mathtt{C}(\mathbf{0})}{\Gamma, \Omega \vdash_{\vee\overline{\beta}} \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}(\mathbf{0})}\ \textit{(New)}$$

$$\frac{\Gamma, \Omega \vdash_\beta \mathtt{e} : \mathtt{C}(\beta)\ \Gamma, \Omega \vdash_{\overline{\beta}} \overline{\mathtt{e}} : \overline{\tau}(\overline{\alpha})\ \Gamma\{\tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau})\}, \Omega \vdash_\beta \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau}) : \mathtt{C}(\beta) \times \langle \overline{\tau}(\overline{\alpha}) \rangle \to \tau(\alpha)}{\Gamma, \Omega \vdash_{\beta \vee (\vee\overline{\beta})} \mathtt{e.m}(\overline{\mathtt{e}}) : \tau(\alpha)}\ \textit{(C)}$$

Figure 5: Typing rules for expressions

4.5.2. Instructions typing rules

The typing rules for instructions are provided in Figure 6. They can be explained briefly:

- Rule *(Skip)* is straightforward.
- Rule *(Seq)* shows that the tier of the sequence $\mathtt{I}_1\ \mathtt{I}_2$ will be the maximum of the tiers of $\mathtt{I}_1$ and $\mathtt{I}_2$. It can be read as "a sequence of instructions

including at least one instruction that cannot be controlled by tier **0** cannot be controlled by tier **0**" and it preserves non-interference as it is a weakly monotonic typing rule with respect to tiers.

- The recovery is performed thanks to the Rule *(ISub)* that makes possible to type an instruction of tier **0** by **1** (as tier **0** instruction use is less constrained than tier **1** instruction use) without breaking the system non-interference properties. Notice also that there is no counterpart for expressions as a subtyping rule from **1** to **0** would allow us to type `x+1;` with `x` of tier **1** while a subtyping rule from **0** to **1** would allow the programmer to type programs with tier **0** variables in the guards of while loops.
- Rule *(Ass)* is an important non-interference rule. It forbids information flows of reference type from a tier to another: it is only possible to assign an expression `e` of tier α to a variable `x` of tier α. In the particular case of primitive data, a flow from **1** to **0** might occur. This is just because primitive data are supposed to be passed-by-value (otherwise we should keep the equality of tier preserved). In the case of a field assignment, all tiers are constrained to be **0** in order to avoid changes inside the tier **1** graph. Finally, if the expression `e` cannot be used in instructions of tier less than β, we constrain the instruction tiered type to be $\texttt{void}(\alpha \vee \beta)$ in order to fulfill this requirement.
- Rule *(If)* constrains the tier of the conditional guard `e` to match the tiers of the branching instructions $\texttt{I}_1$ and $\texttt{I}_2$. Hence, it prevents assignments of tier **1** variables to be controlled by a tier **0** expression.
- Rule *(Wh)* constrains the guard of the loop `e` to be a boolean expression of tier **1**, thus preventing while loops from being controlled by tier **0** expressions.

4.5.3. Methods typing rules

The typing rules for methods are provided in Figure 7. They can be explained briefly:

- Rule *(Body)* shows how to type method definitions. It updates the environment with respect to the parameters, current object and return type in order to allow a polymorphic typing discipline for methods: while typing a program, a method can be given distinct types depending on where and how it is called. The tier γ of the method body is used as annotation so that if $\gamma = \mathbf{1}$, the method cannot be called in a tier **0** instruction. We also check that the current object matches the tier β. As presented, this rule may create an infinite branch in the typing tree of a recursive method. This could be solved by keeping the set of methods previously typed in a third context, we chose not to include this complication. Consequently, a method is implicitly typed only once in a given branch of a typing derivation.

$$\frac{}{\Gamma,\Omega\vdash\ ;\ :\mathtt{void}(\alpha)}\ \textit{(Skip)} \qquad \frac{\forall i,\ \Gamma,\Omega\vdash \mathtt{I}_i:\mathtt{void}(\alpha_i)}{\Gamma,\Omega\vdash \mathtt{I}_1\ \mathtt{I}_2:\mathtt{void}(\alpha_1\vee\alpha_2)}\ \textit{(Seq)}$$

$$\frac{\Gamma,\Omega\vdash \mathtt{I}:\mathtt{void}(\mathbf{0})}{\Gamma,\Omega\vdash \mathtt{I}:\mathtt{void}(\mathbf{1})}\ \textit{(ISub)} \qquad \frac{[\Gamma,\Omega\vdash_{\mathbf{0}}\mathtt{x}:\tau(\alpha')]\quad \Gamma,\Omega\vdash_\beta \mathtt{e}:\tau(\alpha)}{\Gamma,\Omega\vdash [[\tau]\ \mathtt{x}{:=}]\mathtt{e};:\mathtt{void}(\alpha\vee\beta)}\ \textit{(Ass**)}$$

$$\frac{\Gamma,\Omega\vdash_\alpha \mathtt{e}:\mathtt{boolean}(\alpha)\quad \forall i,\ \Gamma,\Omega\vdash \mathtt{I}_i:\mathtt{void}(\alpha)}{\Gamma,\Omega\vdash \mathtt{if(e)\{I_1\}else\{I_2\}}:\mathtt{void}(\alpha)}\ \textit{(If)}$$

$$\frac{\Gamma,\Omega\vdash_{\mathbf{1}}\mathtt{e}:\mathtt{boolean}(\mathbf{1})\quad \Gamma,\Omega\vdash \mathtt{I}:\mathtt{void}(\mathbf{1})}{\Gamma,\Omega\vdash \mathtt{while(e)\{I\}}:\mathtt{void}(\mathbf{1})}\ \textit{(Wh)}$$

(**)
$\mathtt{x}\in\mathcal{F}\implies\alpha=\mathbf{0}$
$\tau\in\mathbb{T}\implies\alpha'\leq\alpha$
$\tau\in\mathbb{C}\implies\alpha'=\alpha$

Figure 6: Typing rules for instructions

- Rule *(Cons)* shows how to type constructors. The constructor body and parameters are enforced to be of tier $\mathbf{0}$ as a constructor invocation makes the memory grow.
- Rule *(OR)* deals with overridden method, keeping tiers preserved, thus allowing standard OO polymorphism.

$$\frac{\Gamma\{\mathtt{this}\leftarrow \mathtt{C}(\beta),\overline{\mathtt{x}}\leftarrow\overline{\tau}(\overline{\alpha}),[\mathtt{x}\leftarrow\tau(\alpha)]\},\Omega\vdash I:\mathtt{void}(\gamma)\quad \Gamma,\Omega\vdash_\gamma \mathtt{this}:\mathtt{C}(\beta)}{\Gamma,\Omega\vdash_\gamma \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau}):\mathtt{C}(\beta)\times\langle\overline{\tau}(\overline{\alpha})\rangle\to\tau(\alpha)}\ \textit{(Body**)}$$

$$\frac{\Gamma\{\overline{\mathtt{x}}\leftarrow\overline{\tau}(\mathbf{0})\},\Omega\vdash I:\mathtt{void}(\mathbf{0})\quad \mathtt{C}(\overline{\tau\ \mathtt{x}})\{\mathtt{I}\}\in\mathtt{C}}{\Gamma,\Omega\vdash \mathtt{C}(\overline{\tau}):\langle\overline{\tau}(\mathbf{0})\rangle\to\mathtt{C}(\mathbf{0})}\ \textit{(Cons)}$$

$$\frac{\mathtt{C}\trianglelefteq\mathtt{D}\qquad \Gamma,\Omega\vdash_\gamma \tau\ \mathtt{m}^{\mathtt{D}}(\overline{\tau}):\mathtt{D}(\beta)\times\langle\overline{\tau}(\overline{\alpha})\rangle\to\tau(\alpha)\quad \tau\ \mathtt{m}(\overline{\tau\ \mathtt{x}})\{\mathtt{I}\ [\mathtt{return\ x};]\}\in\mathtt{D}}{\Gamma,\Omega\vdash_\gamma \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau}):\mathtt{C}(\beta)\times\langle\overline{\tau}(\overline{\alpha})\rangle\to\tau(\alpha)}\ \textit{(OR)}$$

(**) provided that $\tau\ \mathtt{m}(\overline{\tau\ \mathtt{x}})\{\mathtt{I}\ [\mathtt{return\ x};]\}\in\mathtt{C}$

Figure 7: Typing rules for methods

In the above rules, the tier γ is just an annotation on the least tier of the instructions where the method is called. It will be used in the next Section to constrain the tier of recursive methods.

4.6. Well-typedness.

Given a program of executable $\mathtt{Exe\{void\ main()\{Init\ Comp\}\}}$, a typing environment Δ and operator typing environment Ω, the judgment:

$$\Delta,\Omega\vdash \mathtt{Exe\{void\ main()\{Init\ Comp\}\}}:\diamond$$

means that the program is well-typed with respect to Δ and Ω and is defined by:

$$\frac{(\texttt{void main}^{\texttt{Exe}}(), \Delta), \Omega \vDash \texttt{Init} : \texttt{void} \quad (\texttt{void main}^{\texttt{Exe}}(), \Delta), \Omega \vdash \texttt{Comp} : \texttt{void}(\mathbf{1})}{(\texttt{void main}^{\texttt{Exe}}(), \Delta), \Omega \vdash \texttt{Exe}\{\texttt{void main}()\{\texttt{Init Comp}\}\} : \diamond}$$

where $\vDash$ is a judgment derived from the type system by removing all tiers and tier based constraints in the typing rules. For example, Rule *(C)* of Figure 5 becomes:

$$\frac{\Gamma, \Omega \vDash \texttt{e} : \texttt{C} \quad \Gamma, \Omega \vDash \overline{\texttt{e}} : \overline{\tau} \quad \Gamma\{\tau\ \texttt{m}^{\texttt{C}}(\overline{\tau})\}, \Omega \vDash \tau\ \texttt{m}^{\texttt{C}}(\overline{\tau}) : \texttt{C} \times \langle\overline{\tau}\rangle \to \tau}{\Gamma, \Omega \vDash \texttt{e.m}(\overline{\texttt{e}}) : \tau}\ \textit{(C)}$$

or Rule *(Body)* of Figure 7 becomes:

$$\frac{\Gamma\{\texttt{this} \leftarrow \texttt{C}, \overline{\texttt{x}} \leftarrow \overline{\tau}, [\texttt{x} \leftarrow \tau]\}, \Omega \vDash I : \texttt{void} \quad \Gamma, \Omega \vDash \texttt{this} : \texttt{C}}{\Gamma, \Omega \vDash \tau\ \texttt{m}^{\texttt{C}}(\overline{\tau}) : \texttt{C} \times \langle\overline{\tau}\rangle \to \tau}\ \textit{(Body)}$$

It means that all the environments are lifted so that types are substituted to tiered types.

Since no tier constraint is checked in the initialization instruction `Init`, the complexity of this latter instruction is not under control ; as explained previously the main reason for this choice is that this instruction is considered to be building the program input. In contrast, the computational instruction `Comp` is considered to be the computational part of the program and has to respect the tiering discipline.

4.7. Examples

4.7.1. Well-typedness and type derivations

Example 7. *Consider a program having the following computational instruction:*

```
Exe {
  void main(){
    //Init
    int n :=... ;
    BList b := null;
    while(n>0){
      b := new BList(true,b);
      n := n-1;
    }
    //Comp
    int z := 0;
    while(b.getTail() != null){
      b := b.getTail();
      z := z+1;
    }
  }
}
```

It is well-typed as it can be typed by the following derivation:

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathtt{z}) = \mathtt{int}(\mathbf{0})}{\Gamma,\Omega \vdash_{\mathbf{0}} \mathtt{z} : \mathtt{int}(\mathbf{0})}\,(Var) \quad \dfrac{}{\Gamma,\Omega \vdash_{\mathbf{0}} 0 : \mathtt{int}(\mathbf{0})}\,(Cst)}{\Gamma,\Omega \vdash \ \mathtt{z}{:=}0; : \mathtt{void}(\mathbf{0})}\,(Ass) \quad \dfrac{\begin{array}{c}\vdots\\ (\Pi)\end{array}}{\Gamma,\Omega \vdash \mathtt{I} : \mathtt{void}(\mathbf{1})}\,(Wh)}{\Gamma,\Omega \vdash \mathtt{Comp} : \mathtt{void}(\mathbf{1})}\,(Seq)$$

where $\mathtt{I} = \mathtt{while(b.getQueue()! = null)\{b := b.getQueue(); z := z+1;\}}$. *Now we consider the sub-derivation* Π*, where we omit* Γ,Ω *in the context in order to lighten the notation and where* $\mathtt{J} = \mathtt{b := b.getQueue(); z := z+1;}$*:*

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Pi_1\end{array}}{\vdash_{\mathbf{1}} \mathtt{b.getQueue}() : \mathtt{BList}(\mathbf{1})}\,(C) \quad \dfrac{}{\vdash_{\mathbf{1}} \mathtt{null} : \mathtt{BList}(\mathbf{1})}}{\vdash_{\mathbf{1}} \mathtt{b.getQueue}()! = \mathtt{null} : \mathtt{boolean}(\mathbf{1})} \quad \dfrac{\begin{array}{c}\vdots\\ \Pi_2\end{array}}{\vdash \mathtt{J} : \mathtt{void}(\mathbf{1})}\,(Seq)}{\vdash \mathtt{I} : \mathtt{void}(\mathbf{1})}\,(Wh)$$

For Π_1 *we have:*

$$\dfrac{\dfrac{\Gamma(\mathtt{b}) = \mathtt{BList}(\mathbf{1})}{\vdash_{\mathbf{1}} \mathtt{b} : \mathtt{BList}(\mathbf{1})} \quad \dfrac{\dfrac{}{\{\mathtt{getQueue}\}[\mathtt{this}, \mathtt{queue} : \mathtt{BList}(\mathbf{1})], \Omega \vdash ; : \mathtt{void}(\mathbf{1})\ \cdots}}{\{\mathtt{getQueue}\} \vdash_{\mathbf{1}} \mathtt{getQueue} : \mathtt{BList}(\mathbf{1}) \to \mathtt{BList}(\mathbf{1})}}{\vdash_{\mathbf{1}} \mathtt{b.getQueue}() : \mathtt{BList}(\mathbf{1})}\,(C)$$

where $\{\mathtt{getQueue}\}$ *is a shorthand notation for* $\Gamma\{\mathtt{BList\ getQueue}^{\mathtt{BList}}()\}, \Omega$ *and* $[\mathtt{this}, \mathtt{queue} : \mathtt{BList}(\mathbf{1})]$ *is a shorthand notation for* $[\mathtt{this} \leftarrow \mathtt{BList}(\mathbf{1}), \mathtt{queue} \leftarrow \mathtt{BList}(\mathbf{1})]$.

For Π_2*, provided that* $\mathtt{me} = \mathtt{b.getQueue}()$*, we have:*

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathtt{b}) = \mathtt{BList}(\mathbf{1})}{\vdash_{\mathbf{0}} \mathtt{b} : \mathtt{BList}(\mathbf{1})} \quad \dfrac{\Pi_1}{\vdash_{\mathbf{1}} \mathtt{me} : \mathtt{BList}(\mathbf{1})}}{\vdash \mathtt{b} := \mathtt{me}; : \mathtt{void}(\mathbf{1})} \quad \dfrac{\dfrac{\Gamma(\mathtt{z}) = \mathtt{int}(\mathbf{0})}{\vdash_{\mathbf{0}} \mathtt{z} : \mathtt{int}(\mathbf{0})} \quad \dfrac{\vdots}{\vdash_{\mathbf{0}} \mathtt{z}+1 : \mathtt{int}(\mathbf{0})}}{\vdash \mathtt{z} := \mathtt{z}+1; : \mathtt{void}(\mathbf{0})}}{\vdash \mathtt{J} : \mathtt{void}(\mathbf{1})}\,(Seq)$$

Though incomplete, the above derivation can be fully typed as we have already seen in Example 6 that $+1$ *is a positive operator. Consequently, it can be given the type* $\mathtt{int}(\mathbf{0}) \to \mathtt{int}(\mathbf{0})$ *and* $\mathtt{z}+1$ *can be typed, provided that* $\Gamma(\mathtt{z}) = \mathtt{int}(\mathbf{0})$.

4.7.2. Light notation: `BList` *revisited*

As we have seen above, type derivation can be tricky to provide. In order to overcome this problem, we will no longer use typing derivations: tiers will be made explicit in the code. The notation $\mathtt{x}^{\alpha}$ means that $\mathtt{x}$ has tier α under the considered environments Γ,Ω, *i.e.* $\Gamma,\Omega \vdash_{\beta} \mathtt{x} : \tau(\alpha)$, for some τ and β, whereas $\mathtt{I} : \alpha$ means that $\Gamma,\Omega \vdash \mathtt{I} : \mathtt{void}(\alpha)$.

Example 8. *We will consider some of the constructors and methods of the class* `BList` *to illustrate this notation.*

```
BList { /* List of booleans: coding binary integers */
        boolean value;
        BList queue;

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

        BList getQueue() { return queue; }

        void setQueue(BList q) {
                queue := q;
        }

        boolean getValue() { return value; }

        BList clone() {
                BList n;
                if (value != null) {
                        n = new BList(value, queue.clone());
                } else {
                        n = new BList(null, null);
                }
                return n;
        }

        void decrement() {
                if (value == true) {
                        value := false;
                } else {
                        if (queue != null) {
                                value := true;
                                queue.decrement();
                        } else {
                                value := false;
                        }
                }
        }

        int length () {
                int res := 1;
                if ( queue != null ) {
                        res := queue.length();
                        res := res +1;
                }
                else {;}
```

```
                return res ;
        }

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

Let us now consider and type each method and constructor:

```
BList(boolean v^0, BList q^0) {
  value := v; : 0
  queue := q; : 0
}
```

The constructor BList *can be typed by* $\texttt{boolean}(\mathbf{0}) \times \texttt{Blist}(\mathbf{0}) \to \texttt{BList}(\mathbf{0})$, *using Rules (Cons) and (Seq) and twice Rule (Ass). In this latter applications, the tier are enforced to be* $\mathbf{0}$ *because of the side condition* $\texttt{x} \in \mathcal{F} \implies \alpha = \mathbf{0}$. *As a consequence, it is not possible to create objects of tiered type* $\texttt{BList}(\mathbf{1})$ *in a computational instruction.*

```
BList getQueue() {
  return queue;
}
```

We have already seen in Example 7 that the method getQueue *can be typed by* $\texttt{BList}(\mathbf{1}) \to \texttt{BList}(\mathbf{1})$. *It can also be typed by* $\texttt{BList}(\mathbf{0}) \to \texttt{BList}(\mathbf{0})$, *by Rules (Body) and (Self).*

The types $\texttt{BList}(\alpha) \to \texttt{BList}(\beta)$, $\alpha \neq \beta$, *are prohibited because of Rules (Body) and (Self) since the tier of the current object has to match the tier of its fields.*

In the same manner, the method getValue *can be given the types* $\texttt{BList}(\alpha) \to \texttt{boolean}(\alpha)$, $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

The method setQueue*:*

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

can only be given the types $\texttt{BList}(\mathbf{0}) \times \texttt{BList}(\mathbf{0}) \to \texttt{void}(\beta)$. *Indeed, by Rule (Ass),* `queue` *and* `q` *are enforced to be of tier* $\mathbf{0}$. *Consequently,* `this` *is of tier* $\mathbf{0}$ *by Rules (Body) and (Self). The tier of the body is not constrained.*

Consider the method `decrement`*:*

```
void decrement() {
  if (value⁰ == true) {
    value⁰ := false; :0
  } else {
    if (queue⁰ != null) {
      value := true;
      queue⁰.decrement(); :0
    } else {
      value⁰ := false; :0
    }
  }
} :0
```

Because of the Rules (Ass) and (Body), it can be given the type $\texttt{BList}(\mathbf{0}) \to \texttt{void}(\mathbf{0})$. *As we shall see later in Subsection 5.4, this method will be rejected by the safety condition as it might lead to exponential length derivation whenever it is called in a while loop.*

Now consider the following method:

```
int length() {
  int res := 1; : 0
  if (queue¹ != null) {
    res := queue.length(); : 1 //using (Isub)
    res := res+1; : 0
  }
  else {;}
  return res;
}
```

It can be typed by $\Gamma, \Omega \vdash_{\mathbf{1}} \texttt{int length}^{\texttt{BList}}() : \texttt{BList}(\mathbf{1}) \to \texttt{int}(\mathbf{0})$.

Now consider the method testing the equality:

```
boolean isEqual(BList other¹) {
  boolean res⁰ := true; :1
  BList b1¹ := this¹; :1
  BList b2¹ := other¹; :1
  while (b1¹ != null && b2¹ != null){
     if(b1¹.getValue() != b2¹.getValue()){
       res⁰ := false;:1 //using (ISub)
     } else {;}
     b1¹ := b1¹.getQueue();:1
     b2¹ := b2¹.getQueue();:1
  }
  if (b1¹ != null || b2¹ != null) {
    res⁰ := false;:1 //using (ISub)
  } else {;} :1
```

```
    return res^0;
  }
```

The local variables `b1` *and* `b2` *are enforced to be of tier* **1** *by Rules (Wh) and (Op). Consequently,* `this` *and* `other` *are also of tier* **1** *using twice Rule (Ass). This is possible as it does not correspond to field assignments. Moreover, the methods* `getValue` *and* `getQueue` *will be typed by* $\texttt{BList}(\mathbf{1}) \to \texttt{boolean}(\mathbf{1})$ *and* $\texttt{BList}(\mathbf{1}) \to \texttt{BList}(\mathbf{1})$*, respectively. Finally, the local variable can be given the type* $\texttt{boolean}(\mathbf{1})$ *or* $\texttt{boolean}(\mathbf{0})$ *(in this latter case, the subtyping Rule (ISub) will be needed as illustrated in the above instruction) and, consequently, the admissible types for* `isEqual` *are* $\texttt{BList}(\mathbf{1}) \times \texttt{BList}(\mathbf{1}) \to \texttt{boolean}(\alpha)$*,* $\alpha \in \{\mathbf{0}, \mathbf{1}\}$*.*

4.7.3. Examples of overriding

Example 9 (Override1)**.** *Consider the following classes:*

```
A {
  int x;
   ...
  void f(int y){
    x := x+1; : 1 //using (ISub)
  }
}

B extends A{
  void f(int y){
    while(y^1>0){
      x := x+1; : 0
      y := y-1; : 1
    }
  }
}
```

In the class `A`*, the method* `f` *can be given the type* $\texttt{A}(\mathbf{0}) \times \texttt{int}(\mathbf{1}) \to \texttt{void}(\mathbf{1})$ *using Rules (Op), (Ass), (Self) and (ISub). In the class* `B`*, the method* `f` *can be given the type* $\texttt{B}(\mathbf{0}) \times \texttt{int}(\mathbf{1}) \to \texttt{void}(\mathbf{1})$ *using Rules (Op), (Ass), (Seq), (Wh) and (Self). Consequently, the following code can be typed:*

```
A x = ...
if (condition){
  x=new A(...);
}else{
  x=new B(...);
}
x.f(25);
```

provided that `condition` *is of tier* **1** *and using Rules (Pol) and (OR). Indeed, we are not able to predict statically which one of the two method will be called. However we know that the argument is of tier* **0** *as its field will increase polynomially in both cases.*

Example 10 (Override2)**.** *Consider the class* `BList` *and a subclass* `PairOfBList` *:*

```
public class BList{
        ...
        int length () {
                int res := 1;
                if ( queue != null ) {
                        res := queue.length();
                        res := res +1;
                }
                else {;}
                return res ;
        }
}

public class PairOfBList extends BList {
        Blist l^1;

        int length () {
                int res^0 := queue^1.length()^0+2;
                while ( l^1 != null ) {
                        l^1 := l^1.getQueue();
                        res^0 := res^0 +1;
                }
                else {;}
                return res^0 ;
        }
}
```

The override method `length` *of* `PairOfBList` *computes the size of a pair of* `BList` *objects, that is the sum of their respective size. As highlighted by the annotations, it can be given the type* $\texttt{PairOfBList}(\mathbf{1}) \to \texttt{int}(\mathbf{0})$*. Consequently, a method call of the shape:*

$$\texttt{int i}^{\mathbf{0}} := \texttt{p.length}()^{\mathbf{0}}; : \texttt{void}(\mathbf{1})$$

`p` *being of type* `PairOfBList`*, can be typed using the Rule (OR):*

$$\frac{\texttt{PairOfBList} \trianglelefteq \texttt{BList} \qquad \Gamma, \Omega \vdash_{\mathbf{1}} \texttt{int length}^{\texttt{BList}}() : \texttt{BList}(\mathbf{1}) \to \texttt{int}(\mathbf{0})}{\Gamma, \Omega \vdash_{\mathbf{1}} \texttt{int length}^{\texttt{PairOfBList}}() : \texttt{PairOfBList}(\mathbf{1}) \times \to \texttt{int}(\mathbf{0})}$$

as Example 8 has demonstrated that the judgment $\Gamma, \Omega \vdash_{\mathbf{1}} \texttt{int}\ length^{BList}() : BList(\mathbf{1}) \to \texttt{int}(\mathbf{0})$ *can be derived. In order for the program to be safe, the tier annotation is* **1** *and, consequently, the assignment is of tiered type* $\texttt{void}(\mathbf{1})$ *(hence cannot be guarded by* **0** *data).*

4.7.4. Exponential as a counter-example

Example 11. *Now we illustrate the limits of the type system with respect to the following method computing an exponential:*

```
exp(int x, int y){
  while(x^1>0){
       int u^? := y^0;
       while (u^?>0){
          y := y^0+1; : 0
          u := u^?-1; : ?
       }
       x := x^1-1;
  }
  return y^0;
}
```

It is not typable in the presented formalism. Indeed, suppose that it is typable. The expression `y+1;` *enforces* `y` *to be of tier* **0** *by Rule (Op) and by definition of positive operators. Consequently, the instruction* `int u := y;` *enforces* `u` *to be of tier* **0** *because of typing discipline for assignments (Rule (Ass)). However, the innermost while loop enforces* $\mathtt{u} > 0$ *to be of tier* **1** *by Rules (Wh) and (Op), so that* `u` *has to be of tier* **1** *and we obtain a contradiction.*

Now, one might suggest that the exponential could be computed using an intermediate method `add` *for addition:*

```
add(int x, int y){
  while(x^1>0){
    x^1 := x^1-1; :1
    y^0 := y^0+1; :0
  }
  return y^0;
}

expo(int x){
  int res := 1;
  while(x^1>0){
      res := add(res^?, res^?);
      x := x^1-1; : 1
  }
  return res;
}
```

Thankfully, this program is not typable as the first argument of `add` *is enforced to be of tier* **1** *and the second argument is enforced to be of tier* **0** *(see previous section). Hence, the variable* `res` *would have two distinct tiers under the same context. That is clearly not allowed by the type system.*

4.8. 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* $\mathtt{I}$*, a contextual typing environment* Γ *and an operator typing environment* Ω *such that* $\Gamma, \Omega \vdash \mathtt{I} : \mathtt{void}(\alpha)$ *holds, there is a contextual typing environment* Γ' *such that the following holds:*

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

where $\mathtt{x} \in \mathtt{I}$ *means that the variable* $\mathtt{x}$ *appears in* $\mathtt{I}$*.*

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

Proof. By induction on program flattening on instructions. Consider a method call $\mathtt{I} = \tau\ \mathtt{x}{:=}\mathtt{e}.\mathtt{m}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$; such that $\Gamma, \Omega \vdash \mathtt{I} : \mathtt{void}(\alpha)$ and $\Gamma = (s, \Delta)$. This means that $\Gamma, \Omega \vdash_{\gamma_i} \mathtt{e}_i : \tau_i(\alpha_i)$, $\Gamma, \Omega \vdash_{\gamma} \mathtt{x} : \tau(\alpha)$ and $\Gamma, \Omega \vdash_{\gamma'} \mathtt{e} : \mathtt{C}(\beta)$ hold, for some γ_i, γ, γ', τ_i, α_i, τ, α, $\mathtt{C}$ and β (see Rule *(C)* of Figure 5). Applying the rules of Figure 3, the flattening of $\mathtt{I}$ is of the shape $\underline{\mathtt{J}}\ [\tau]\ \mathtt{x} = \mathtt{x}_{n+1}.\mathtt{m}'(\mathtt{x}_1, \ldots, \mathtt{x}_n)$; with $\mathtt{J} = \tau_1\ \mathtt{x}_1{:=}\mathtt{e}_1; \ldots \tau_n\ \mathtt{x}_n{:=}\mathtt{e}_n; \tau_{n+1}\ \mathtt{x}_{n+1}{:=}\mathtt{e};$. Let Γ' for the environment that is equal to Γ but on the method signature s where:

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

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

The converse is straightforward as Γ and Γ' match on the variables that they "share" in common. □

4.9. Subject reduction

We introduce an intermediate lemma stating that an instruction obtained through the evaluation of the computational instruction of a well-typed program is also well-typed. For this, we first need to extend the type system so that it will be defined on any meta-instruction:

$$\frac{}{\Gamma, \Omega \vdash \mathtt{pop};: \mathtt{void}(\alpha)}\ \textit{(Pop)} \qquad \frac{}{\Gamma, \Omega \vdash \mathtt{push}(s_{\mathcal{H}});: \mathtt{void}(\mathbf{0})}\ \textit{(Push)}$$

Figure 8: Extra typing rules for Push and Pop instructions

Notice that this definition is quite natural as `push` makes the memory increase and so is of tier **0** while `pop` makes the memory decrease and so can be of any tier. One possibility would have been to include these typing rules directly in the initial type system. We did the current choice as these meta-instructions

are only obtained during computation whereas the type inference is supposed to be performed statically on the flattened program, that is an AOO program, independently of its execution.

Lemma 2 (Subject reduction)**.** *Let* Γ *be a contextual typing environment,* Ω *be an operator typing environment and* P *be an AOO program of executable* $\texttt{Exe}\{\texttt{void main()}\{\texttt{Init Comp}\}\}$ *such that* P *is well-typed with respect to* Γ *and* Ω*. If* $(\mathcal{I}, \underline{\texttt{Comp}}) \rightarrow^* (\mathcal{C}, \texttt{MI})$ *then* $\Gamma, \Omega \vdash \texttt{MI} : \texttt{void}(\mathbf{1})$*.*

Proof. By Proposition 1, $\underline{\texttt{Comp}}$ can be typed. We proceed by induction on the reduction length n of $\rightarrow$; $\rightarrow^n$ being a notation for a length n reduction:

- If $n = 0$ then $\texttt{MI} = \underline{\texttt{Comp}}$ and since the program is well-typed, we have $\Gamma, \Omega \vdash \underline{\texttt{Comp}} : \texttt{void}(\mathbf{1})$.
- Now consider a reduction of length $n+1$. It can be written as $(\mathcal{I}, \underline{\texttt{Comp}}) \rightarrow^n (\mathcal{C}, \texttt{MI}) \rightarrow (\mathcal{C}', \texttt{MI}')$. By induction hypothesis $\Gamma, \Omega \vdash \texttt{MI} : \texttt{void}(\mathbf{1})$. Moreover, the last rule can be:
  - the evaluation of an assignment (Rules (1-7) of Figure 4) or the evaluation of a push or pop instructions (Rules (9) and (10) of Figure 4). In all these cases, $\texttt{MI} = \texttt{MI}_1\ \texttt{MI}'$ and $(\mathcal{C}, \texttt{MI}) \rightarrow (\mathcal{C}', \texttt{MI}')$. By Rule *(Seq)* of Figure 6, $\Gamma, \Omega \vdash \texttt{MI}' : \texttt{void}(\alpha)$, for some α and, consequently, $\Gamma, \Omega \vdash \texttt{MI}' : \texttt{void}(\mathbf{1})$ using Rule *(ISub)* of Figure 6.
  - the evaluation of a method call (Rule (8) of Figure 4). In this case, $(\mathcal{C}, \texttt{MI}) = (\mathcal{C}, [\texttt{x:=}]\texttt{z.m}(\overline{\texttt{y}}); \texttt{MI}') \rightarrow (\mathcal{C}', \texttt{push}(s_{\mathcal{H}}); \texttt{MI}''\ [\texttt{x:=z}';]\ \texttt{pop};\ \texttt{MI})$, provided that $\texttt{MI}''$ is the flattened body of method $\texttt{m}$. By Rule *(Seq)* of Figure 6, $\Gamma, \Omega \vdash \texttt{MI}' : \texttt{void}(\alpha)$, for some α. By the extra rules of Figure 8, $\Gamma, \Omega \vdash \texttt{push}(s_{\mathcal{H}});: \texttt{void}(\mathbf{0})$ and $\Gamma, \Omega \vdash \texttt{pop};: \texttt{void}(\mathbf{0})$. Moreover by Rules *(Body)* of Figure 6 and *(C)* of Figure 5, the flattened body $\texttt{MI}''$ can be typed by $\Gamma, \Omega \vdash \texttt{MI}'' : \texttt{void}(\beta)$, for some β (the method output tier). We let the reader check that $\texttt{x:=z}'$; can also be typed using the same reasoning in the case where the method returns something. Finally, using several times Rule *(Seq)* and one time Rule *(ISub)* of Figure 6, we obtain that $\Gamma, \Omega \vdash \texttt{push}(s_{\mathcal{H}}); \texttt{MI}''\ [\texttt{x:=z}';]\ \texttt{pop};\ \texttt{MI} : \texttt{void}(\mathbf{1})$.
  - the evaluation of a while loop (Rules (11) and (12) of Figure 4). In this case $\texttt{MI} = \texttt{while(x)}\{\texttt{MI}_1\}\ \texttt{MI}_2$ and either $\texttt{x}$ is evaluated to false, in which case we take $\texttt{MI}' = \texttt{MI}_2$ or $\texttt{x}$ evaluates to true and $\texttt{MI}' = \texttt{MI}_2\ \texttt{while(x)}\{\texttt{MI}_1\}$. In both cases, $\Gamma, \Omega \vdash \texttt{MI}_2 : \texttt{void}(\alpha)$, some α, and $\Gamma, \Omega \vdash \texttt{while(x)}\{\texttt{MI}_1\} : \texttt{void}(\mathbf{1})$ can be derived by Rules *(Seq)* and *(Wh)* of Figure 6. Consequently, we can derive $\Gamma, \Omega \vdash \texttt{MI}' : \texttt{void}(\mathbf{1})$ by either using Rule *(ISub)* (for false), if needed, or just by using Rule *(Seq)* (for true).
  - the evaluation of a `if` ((Rule (13) of Figure 4) can be done in a similar manner.

and so the result. □

Remark 1. *Subject reduction is presented in a weak form as we do not consider the tier* **0** *case. It also holds but is of no interest in this particular case as, by monotonicity of the typing rule, every sub-instruction will be of tier 0 and there will be no loop and no recursive call. This is the reason why the computational instruction is typed using tier* **1** *in the definition of well-typedness.*

5. Safe recursion

5.1. Recursive methods

Given two methods of signatures s and s' and names $\mathtt{m}$ and $\mathtt{m}'$, define the relation $\sqsubset$ on method signatures by $s \sqsubset s'$ if $\mathtt{m}'$ is called in in the body of $\mathtt{m}$. This relation is extended to inheritance by considering that overriding methods are called by the overridden method. 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 class $[s]$ is defined as usual by $[s] = \{s' \mid s' \equiv s\}$. Finally, we write $s \sqsubsetneq^+ s'$ if $s \sqsubset^+ s'$ holds but not $s' \sqsubset^+ s$.

Notice that the extension of $\sqsubset$ to method override is rough as we consider overriding methods to be called by the overridden method though it is clearly not always the case. However it is put in order to make the set $[s]$ computable for any method signature s. Indeed in a method call of the shape $\mathtt{x.m}()$, provided that $\mathtt{x}$ is a declared variable of type $\mathtt{C}$ and that $\mathtt{void\ m}()$ is a method declared in $\mathtt{C}$ and overridden in some subclass $\mathtt{D}$, the evaluation may lead dynamically to either a call to $\mathtt{void\ m}^{\mathtt{C}}()$ or to a call to $\mathtt{void\ m}^{\mathtt{D}}()$ depending on the instance that will be assigned to $\mathtt{x}$ during the program evaluation. This choice is highly undecidable as it depends on program semantics. Here we will consider that $\mathtt{void\ m}^{\mathtt{C}}() \sqsubset \mathtt{void\ m}^{\mathtt{D}}()$. Consequently, if the call $\mathtt{x.m}()$ is performed in the body of a method of signature s then $s \sqsubset \mathtt{void\ m}^{\mathtt{C}}() \sqsubset \mathtt{void\ m}^{\mathtt{D}}()$ and, consequently, $s \sqsubset^+ \mathtt{void\ m}^{\mathtt{D}}()$, *i.e.* a call to the method of signature s may lead to a call to the method of signature $\mathtt{void\ m}^{\mathtt{D}}()$.

Lemma 3. *Given an AOO program, for any method signature s, the set $[s]$ is computable in polynomial time in the size of the program.*

Proof. It just suffices to look at the program syntax to generate linearly the relation $\sqsubset$ and then to compute its transitive closure $\sqsubset^+$. □

5.2. Level

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

Definition 4 (Level)**.** *The level λ of a method signature is defined by:*

- $\lambda(s) = \max\{\lambda(s') \mid s \sqsubset s'\}$ *if $s \notin [s]$ (i.e. the method is not recursive),*
- $\lambda(s) = 1 + \max\{\lambda(s') \mid s \sqsubsetneq^+ s'\}$ *otherwise,*

setting $\max(\emptyset) = 0$.

Let $\lambda(P)$ *be the maximal level of a method in a program* P *(or* $\underline{P}$*). By abuse of notation, we will write* $\lambda(\mathtt{m})$ *and, respectively,* λ *when the signature of the method* `m` *and the program* P*, respectively, are clear from the context.*

Example 12. *Consider the methods of Example 8:*

- $\lambda(\mathtt{getQueue}) = \lambda(\mathtt{getValue}) = \lambda(\mathtt{setQueue}) = \mathtt{0}$ *as these methods do not call any other methods in their body and, consequently, are not recursive,*
- $\lambda(\mathtt{isEqual}) = \mathtt{0}$ *as* `isEqual` *call the methods* `getQueue` *and* `getValue` *but the converse does not hold. Consequently,* $\mathtt{BList\ isEqual}^{\mathtt{BList}}(\mathtt{BList}) \sqsubsetneq^{+} \mathtt{BList\ getQueue}^{\mathtt{BList}}()$ *and* $[\mathtt{BList\ isEqual}^{\mathtt{BList}}(\mathtt{BList})] = \emptyset$*,*
- $\lambda(\mathtt{decrement}) = \mathtt{1} + \max\{\lambda(\mathtt{s}') \mid \mathtt{void\ decrement}^{\mathtt{BList}}() \sqsubsetneq^{+} \mathtt{s}'\} = \mathtt{1} + \max(\emptyset) = \mathtt{1}$,
- $\lambda(\mathtt{length}) = \mathtt{1}$*, for the same reason.*

5.3. Intricacy

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 5 (Intricacy)**.** *Let the intricacy* ν *be partial function from meta-instructions to integers defined as follows:*

- $\nu([[\tau]\mathtt{x}{:=}]\mathtt{me};) = 0$ *if* `me` *is not a method call,*
- $\nu([[\tau]\ \mathtt{x}{:=}]\mathtt{y.m}(\ldots);) = \max_{\mathtt{D} \trianglelefteq \mathtt{C}\star}(\nu(\mathtt{MI_D}))$
  provided that `m` *is of the shape* $\tau\ \mathtt{m}(\ldots)\{\mathtt{MI_D}\ [\mathtt{return\ z};]\}$ *in a class* $\mathtt{D} \trianglelefteq \mathtt{C}\star$*, where* `y` *is of type* `C` *and* $\mathtt{C}\star$ *is the least super-class of* `C` *where* `m` *is defined.*
- $\nu(\mathtt{MI\ MI}') = \max(\nu(\mathtt{MI}), \nu(\mathtt{MI}'))$
- $\nu(\mathtt{if(x)\{MI\}else\{MI}'\}) = \max(\nu(\mathtt{MI}), \nu(\mathtt{MI}'))$
- $\nu(\mathtt{while(x)\{MI\}}) = 1 + \nu(\mathtt{MI})$

Moreover, let $\nu(P)$ *be the maximal intricacy of a meta-instruction within the flattened AOO program* $\underline{P}$*. By abuse of notation, we will write* ν *when the program* P *is clear from the context.*

Observe that for any instruction `I`, the intricacy of its flattening $\nu(\underline{\mathtt{I}})$ is well-defined as there is no `push` or `pop` operations occuring in it. Moreover, in the simple case where there is no inheritance then the intricacy of a method call $\nu([[\tau]\ \mathtt{x}{:=}]\mathtt{y.m}(\ldots);)$, for some method of the shape $\tau\ \mathtt{m}(\ldots)\{\mathtt{MI}\ [\mathtt{return\ z};]\}$ is just equal to $\nu(\mathtt{MI})$ (as $\mathtt{C}\star = \mathtt{C}$ and the only `D` such that $\mathtt{D} \trianglelefteq \mathtt{C}$ is `C` itself).

Example 13. *Consider the following meta-instruction* `MI`*:*

```
while(x){
  while(y){
    b := l.isEqual(o);
  }
}
```

$\nu(\texttt{MI}) = 2 + \nu(\texttt{MI}')$, *if* $\texttt{MI}'$ *is the flattened body of the method* `isEqual`. *We let the reader check that* $\nu(\texttt{MI}') = 1$ *(there is one while inside). Consequently,* $\nu(\texttt{MI}) = 3$.

5.4. Safety

Now we put some aside restrictions on recursive methods to ensure that their computations remain polynomially bounded.

Definition 6 (Safety)**.** *A well-typed program with respect to a typing environment* Δ *and operator typing environment* Ω *is* safe *if for each recursive method* $\tau\ \texttt{m}(\overline{\tau\ \texttt{x}})\{\texttt{I}\ [\texttt{return x;}]\}$*:*

1. *there is exactly one call to some* $\texttt{m}' \in [\texttt{m}]$ *in the instruction* `I`,
2. *there is no while loop inside* `I`, *i.e.* $\nu(\texttt{I}) = 0$,
3. *and only judgments of the shape* $(s, \Delta), \Omega \vdash_{\mathbf{1}} \tau\ \texttt{m}^{\texttt{C}}(\overline{\tau\ \texttt{x}}) : \texttt{C}(\mathbf{1}) \times \langle\overline{\tau}(\mathbf{1})\rangle \to \tau(\alpha)$ *can be derived using Rules (Body) and (OR).*

Item 1 ensures that recursive methods will be restricted to have only one recursive call performed during the evaluation of their body. It will prevent exponential accumulation through mutual recursion. A counter-example is a program computing the Fibonacci sequence with a mutually recursive call of the shape $\texttt{m}(n-1) + \texttt{m}(n-2)$, for $n > 2$. Item 2 is here to simplify the complexity analysis. It is not that restrictive in the sense that it enforces the programmer to make a choice between a functional programming style using pure recursive calls or an imperative one using while loops. Item 3 is very important. It enforces recursive methods to have tier **1** inputs in order to prevent a control flow depending on tier **0** variables during the recursive calls evaluation and to have an output whose use is restricted to tier **1** instructions (this is the purpose of the $\vdash_{\mathbf{1}}$ use) in order to prevent the recursive method call to be controlled by tier **0** instructions. The method output tier α is not restricted.

Example 14. *A well-typed program whose computational instruction uses the methods* `getQueue`, `getValue`, `setQueue`, `length` *and* `isEqual` *of Example 8 will be safe. Indeed the only recursive method is length. As illustrated in Example 8, it can be typed by* $\texttt{BList}(\mathbf{1}) \to \texttt{int}(\mathbf{0})$*, it does not contain any while loop and has only one recursive call in its body.*

A program whose computational instruction uses the method `decrement` *will not be safe as this method can only be given the type* $BList(\mathbf{0}) \to \texttt{void}(\mathbf{0})$*. Though it entails a lack of expressivity, there is no straightforward reason to reject this code. However we will see clearly in the next subsection that we do not want tier* **0** *data to control a while or a recursion while we do not want tier* **1** *data to be altered. Changing the type system by allowing* `decrement` *to apply to tier* **1** *objects would allow codes like:*

```
while(!o.isEqual(l)){
  o.decrement();
}
```

where `l` *is a* `BList` *of* n *booleans equal to* `false`*. Clearly, such a loop can be executed exponentially in the input size (the size of the lists). This is highly undesirable.*

An important point to mention is that safety allows an easy way to perform expression subtyping for objects through the use of cloning. In other words, it is possible to bring a reference type expression from tier **1** to tier **0** based on the premise that it has been cloned in memory (this was already true for primitive data by Rule *(Ass)*). Thus a form of subtyping that does not break the non-interference properties is admissible as illustrated by the following example:

Example 15. *Let us add a clone method to the to the class* `BList`*. This clone method will output a copy of the current object that, as it is constructed, will be of tier* **0***.*

```
BList clone(){
  BList res^0 := null;
  int v^0 := value^1;
  if(queue^1 == null){
    res^0 := new BList(v^0,null)^0;
  }
  else{
   res^0 := new BList(v^0,queue.clone()^0);
  }
  return res;
}
```

This is typable by $\texttt{BList}(\mathbf{1}) \to \texttt{BList}(\mathbf{0})$*. Moreover, the method is safe, as there is only one recursive call.*

Lemma 4. *A program* P *is safe iff* $\underline{P}$ *is safe.*

Proof. By Proposition 1, P is well-typed iff $\underline{P}$ is well-typed. The flattening transformation has no effect on method calls, while loops and method signatures. Consequently, P satisfies Items 1, 2 and 3 iff so does $\underline{P}$. □

Lemma 5. *Given a well-typed program* P *with respect to some typing derivation* Π*, the safety of* P *can be checked in polynomial time in* $|P|$*.*

Proof. By Lemma 3, computing the sets of "equivalent" recursive methods can be done in polynomial time in $|P|$. Once, this is done, checking that there is no while loops and only one recursive call inside can be done quadratically in the program size. Finally, checking for Item 3 can be done linearly in the size of the typing derivation Π of P. However such a derivation is quadratic in the size of the program as all typing rules apart from *(ISub)* and *(Pol)* correspond to some program construct. Just notice that Rule *(ISub)* can be applied only once per instruction while Rule *(Pol)* can be applied at most k times per expression,

provided that k is an upper bound on the number of nested inheritance (and k is bounded by $|P|$). □

Corollary 2. *Given a well-typed program P with respect to some typing derivation Π, the safety of $\underline{P}$ can be checked in polynomial time in $|P|$.*

Proof. By Lemma 5, the safety of P can be checked in polynomial time in $|P|$ and so the result, by Lemma 4. □

5.5. General safety

The safety criterion is sometimes very restrictive from an expressivity point of view because of Item 1. This Item is restrictive but simple: the interest is to have a decidable criterion for safety while keeping a completeness result for polynomial time as we shall see shortly. However it can be generalized to a semantics (thus undecidable criterion) that we call general safety in order to increase program expressivity.

Definition 7 (General safety)**.** *A well-typed program with respect to a typing environment Δ and operator typing environment Ω is* generally safe *if for each recursive method $\tau\ \mathtt{m}(\overline{\tau\ \mathtt{x}})\{\mathtt{I}\ [\mathtt{return\ x;}]\}$:*

1. *the following condition is satisfied:*
   (a) *either there is at most one call to some $\mathtt{m}' \in [\mathtt{m}]$ in the evaluation of $\mathtt{I}$,*
   (b) *or all recursive calls are performed on a distinct field of the current object. Such fields being used at most once.*
2. *there is no while loop inside $\mathtt{I}$, i.e. $\nu(\mathtt{I}) = 0$,*
3. *and only judgments of the shape $(s, \Delta), \Omega \vdash_{\mathbf{1}} \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau\ \mathtt{x}}) : \mathtt{C}(\mathbf{1}) \times \langle\overline{\tau}(\mathbf{1})\rangle \to \tau(\alpha)$ can be derived using Rules (Body) and (OR).*

Example 16. *General safety improves the expressivity of captured programs as illustrated by the following example:*

```
class Tree {
  int node;
  Tree left;
  Tree right;

  int value(BList b¹) {
    int res¹ := node¹;
    if(b¹ != null && right¹ != null && left¹ != null){
      if(b.getValue()){
        res := right.value(b.getQueue()¹); : 1
      }else{
        res := left.value(b.getQueue()¹); : 1
      }
   }else{;}
   return res;
  }
}
```

The method `value` *returns the value of the node whose path is encoded by a boolean list in the method parameter* `b` *(the boolean constants* `true` *and* `false` *encoding the right and left sons, respectively). It can be typed by* $\texttt{Tree}(\mathbf{1}) \times \texttt{BList}(\mathbf{1}) \to \texttt{int}(\mathbf{1})$*. Consequently, it is generally safe (but not safe) as exactly one branch of the conditional (thus one recursive call) can be reached, thus satisfying Item 1a of Definition 7.*

In the same manner, Item 1b allows us to implement the clone method on recursive structures:

```
Tree clone(){
  Tree res^0 := null;
  int n^0 := node^1;
  if(left^1 == null && right^1 == null){
  //The case of non well-balanced trees is not treated.
    res^0 := new Tree(n^0,null,null)^0;
  }
  else{
   res^0 := new Tree(n^0,left.clone()^0, right.clone()^0);
  }
  return res;
}
```

This is typable by $\texttt{Tree}(\mathbf{1}) \to \texttt{Tree}(\mathbf{0})$ *thanks to Item 1b of Definition 7 as the two recursive calls are applied exactly once on distinct fields (*`left` *and* `right`*).*

We can now compare safety and general safety:

Proposition 2. *If a program is safe then it is generally safe. The converse is not true.*

Proof. If there is only one recursive call in the body of any recursive method then, a fortiori, there is only one recursive call during the body evaluation. On the opposite in an instruction of the shape $\texttt{I} = \texttt{if(e)\{m();\}else\{m();\}}$ there is only one call of `m()` during the evaluation of `I` whereas `m` is called twice in `I`. □

Proposition 3. *General safety is undecidable.*

Proof. Since Item 1a can be reduced to either a dead-code problem or a reachability problem that are known to be undecidable problems. Items 1b, 2 and 3 are still decidable in polynomial time. □

Note that there are decidable classes of programs between safe programs and generally safe programs. For example, one may ask a program to have only one recursive call per reachable branch of a conditional in a method body. This requirement is clearly decidable in polynomial time.

Hence general safety can be seen as a generalization of the safe recursion on notation (SRN) scheme by Bellantoni and Cook [1]. Indeed a SRN function can be defined (and typed) by a method `f` in a class `C` by:

```
int f(int x, int y){
    int res = 0;
    if (x==0) {
        res = g(y);
    } else {
        if (x%2 == 0) {
            res = h0(f(x/2,y));
        } else {
            res = h1(f(x/2,y));
        }
    }
    return res;
}
```

This can be typed provided that $\mathtt{f} : \mathtt{C}(\mathbf{1}) \times \mathtt{int}(\mathbf{1}) \times \mathtt{int}(\mathbf{1}) \to \mathtt{int}(\mathbf{0})$, $\mathtt{h}_i : \mathtt{C}(\mathbf{1}) \times \mathtt{int}(\mathbf{0}) \to \mathtt{int}(\mathbf{0})$, $i \in \{0, 1\}$ and $\mathtt{g} : \mathtt{C}(\mathbf{1}) \times \mathtt{int}(\mathbf{1}) \to \mathtt{int}(\mathbf{0})$. As $\mathtt{f}$ is recursive, its parameters have to be of tier $\mathbf{1}$. However its output can be of tier $\mathbf{0}$ provided that we can use only derivations of the shape $\Gamma, \Omega \vdash_{\mathbf{1}} \mathtt{int\ f}() : \mathtt{C}(\mathbf{1}) \times \mathtt{int}(\mathbf{1}) \times \mathtt{int}(\mathbf{1}) \to \mathtt{int}(\mathbf{0})$ and $\Gamma, \Omega \vdash_{\mathbf{1}} \mathtt{h}_i : \mathtt{C}(\mathbf{1}) \times \mathtt{int}(\mathbf{0}) \to \mathtt{int}(\mathbf{0})$. If $\mathtt{f}$ output is of tier $\mathbf{0}$ (i.e. computes something) then $\mathtt{h}_i$ will not be able to recurse on it (as in SRN). Clearly, the above program fullfills the general safety criterion for some typing context Γ such that $\Gamma(\mathtt{x}) = \Gamma(\mathtt{y}) = \mathtt{int}(\mathbf{1})$ and $\Gamma(\mathtt{res}) = \mathtt{int}(\mathbf{0})$ as the recursive calls that are performed in the $\mathtt{res} = \mathtt{h}_0(\mathtt{f}(\mathtt{x}/2, \mathtt{y}))$; instruction can be given the type $\mathtt{void}(\mathbf{1})$ using Rules *(Ass)* and *(ISub)*. The operators %2 and == 0 can be given the type $\Omega(\%2) \ni \mathtt{int}(\mathbf{1}) \to \mathtt{int}(\mathbf{1})$ and $\Omega(== 0) \ni \mathtt{int}(\mathbf{1}) \to \mathtt{boolean}(\mathbf{1})$ as they are neutral. Finally, the method body is typable using twice the Rule *(If)* on tier $\mathbf{1}$ guard and instructions.

6. Type system non-interference properties

In this section, we demonstrate that classical non-interference results are obtained through the use of the considered type system. For that purpose, we introduce some intermediate lemmata. Notice that all the results presented in this section hold for compatible pairs only (see Section 3.8).

The confinement Lemma expresses the fact that no tier $\mathbf{1}$ variables are modified by a command of tier $\mathbf{0}$.

Lemma 6 (Confinement)**.** *Let P be an AOO program of computational instruction* $\mathtt{Comp}$*, Γ be a contextual typing environment and Ω be an operator typing environment such that P is (generally) safe with respect to Γ and Ω. If $\Gamma, \Omega \vdash \mathtt{Comp} : \mathtt{void}(\mathbf{0})$, then every variable assigned to during the execution of $(\mathcal{I}, \underline{\mathtt{Comp}})$ is of tier $\mathbf{0}$.*

Proof. By Proposition 1 and Lemma 4, we know that $\underline{\mathtt{Comp}}$ is safe with respect to environments Γ' and Ω such that $\Gamma', \Omega \vdash \underline{\mathtt{Comp}} : \mathtt{void}(\mathbf{0})$. Now we prove by induction that for any $\mathtt{MI}$ such that $(\mathcal{I}, \underline{\mathtt{Comp}}) \to^* (\mathcal{C}, \mathtt{MI})$, every variable assigned to in the evaluation of $\mathtt{MI}$ is of tier $\mathbf{0}$:

- if $\mathtt{MI} = \epsilon$ or ; then the result is straightforward.

- if $\texttt{MI} = [\tau]\ \texttt{x:=me;}$. Assume $\texttt{x}$ is of tier $\mathbf{1}$. From rule *(Ass)*, it means that $\alpha' = \mathbf{1}$, implying that $\alpha = \mathbf{1}$, then that $\texttt{MI} : \texttt{void}(\mathbf{1})$.
- $\texttt{MI} = \texttt{I}_1\ \texttt{I}_2$ (or $\texttt{if(x)\{I}_1\texttt{\}else\{I}_2\texttt{\}}$) then both $\texttt{I}_i$ are of tiered type $\texttt{void}(\mathbf{0})$ by Rule *(Seq)* (respectively *(If)*) and so the result by induction.
- $\texttt{MI} = \texttt{y.m}(\overline{\texttt{x}})$; then $\texttt{y.m}(\overline{\texttt{x}})$; is of tiered type $\texttt{void}(\mathbf{0})$ by Rule *(Ass)*. Hence $\texttt{m}$ cannot be a recursive method because of safety. More, $\texttt{m}$ should be typed as $\Gamma', \Omega \vdash_\gamma \texttt{void}\ \texttt{m}^{\texttt{C}}(\overline{\tau}) : \texttt{C}(\beta) \times \langle \overline{\tau}(\overline{\alpha}) \rangle \to \texttt{void}(\mathbf{0})$. Finally, by Rule *(Body)* the flattened body of $\texttt{m}$ is also of tier $\texttt{void}(\mathbf{0})$. Consequently, by induction hypothesis, it does not contain assignments of tier $\mathbf{1}$ variables.

and so the result. □

Lemma 7. *Let P be an AOO program of computational instruction* $\texttt{Comp}$*, Γ be a contextual typing environment and Ω be an operator typing environment such that P is (generally) safe with respect to Γ and Ω. For all* $\texttt{MI}$ *such that $(\mathcal{I}, \texttt{Comp}) \to^* (\mathcal{C}, \texttt{MI})$ and $\Gamma, \Omega \vdash \texttt{MI} : \texttt{void}(\mathbf{0})$, there is no while loops and recursive calls evaluated during this execution.*

Proof. From Lemma 2, we know that all instructions reached from a tier $\mathbf{0}$ instruction are of tier $\mathbf{0}$.

- By rule *(Wh)*, if a while loop occurs, it is of tier $\mathbf{1}$.
- By safety assumption, calls to recursive functions are of tier $\mathbf{1}$. □

We now establish a non-interference Theorem which states that if $\texttt{x}$ is variable of tier $\mathbf{1}$ then the value stored in $\texttt{x}$ is independent from variables of tier $\mathbf{0}$. For this, we first need to define the suitable relation on memory configurations and meta-instructions:

Definition 8. *Let Γ be a contextual typing environment and Ω be an operator typing environment.*

- *The equivalence relation $\approx_{\Gamma,\Omega}$ on memory configurations is defined as follows:*
  $\mathcal{C} \approx_{\Gamma,\Omega} \mathcal{D}$ iff the transitive closure of the pointer graph of $\mathcal{C}$ corresponding to $\mathbf{1}$ variables with respect to Γ is equal to the one of $\mathcal{D}$ and both stacks match everywhere except on tier $\mathbf{0}$ variables.
- *The relation $\approx_{\Gamma,\Omega}$ is extended to commands as follows:*
  1. *If $\texttt{I} = \texttt{J}$ then $\texttt{I} \approx_{\Gamma,\Omega} \texttt{J}$*
  2. *If $\Gamma, \Omega \vdash \texttt{I} : \texttt{void}(\mathbf{0})$ and $\Gamma, \Omega \vdash \texttt{J} : \texttt{void}(\mathbf{0})$ then $\texttt{I} \approx_{\Gamma,\Omega} \texttt{J}$*
  3. *If $\texttt{I} \approx_{\Gamma,\Omega} \texttt{J}$ and $\texttt{K} \approx_{\Gamma,\Omega} \texttt{L}$ then $\texttt{I K} \approx_{\Gamma,\Omega} \texttt{J L}$*
- *Finally, it is extended to configurations as follows:*
  If $\texttt{I} \approx_{\Gamma,\Omega} \texttt{J}$ and $\mathcal{C} \approx_{\Gamma,\Omega} \mathcal{D}$ then $(\mathcal{C}, \texttt{I}) \approx_{\Gamma,\Omega} (\mathcal{D}, \texttt{J})$

In other words, two commands that configurations are equivalent for $\approx_{\Gamma,\Omega}$ if their memory configurations restricted to tier **1** are the same and their commands of tier **1** are the same.

Theorem 1 (Non-interference)**.** *Assume that* Γ *is a contextual typing environment and* Ω *is an operator typing environment such that* `MI` *and* `MJ` *are the computational instructions of two safe programs with respect to* Γ *and* Ω *having exactly the same classes. Assume also that* $(\mathcal{C}, \texttt{MI}) \approx_{\Gamma,\Omega} (\mathcal{D}, \texttt{MJ})$*. If* $(\mathcal{C}, \texttt{MI}) \to (\mathcal{C}', \texttt{MI}')$ *then there exist* $\mathcal{D}'$ *and* $\texttt{MJ}'$ *such that:*

- $(\mathcal{D}, \texttt{MJ}) \to^* (\mathcal{D}', \texttt{MJ}')$
- *and* $(\mathcal{C}', \texttt{MI}') \approx_{\Gamma,\Omega} (\mathcal{D}', \texttt{MJ}')$

Proof. We proceed by induction on `MI`.

- If $\texttt{MI} = \texttt{x:=me;}$. There are two cases to consider.
  - If $\Gamma, \Omega \vdash \texttt{MI} : \texttt{void}(\mathbf{0})$ then, by definition of $\approx_{\Gamma,\Omega}$, $\Gamma, \Omega \vdash \texttt{MJ} : \texttt{void}(\mathbf{0})$. Indeed, either $\texttt{MJ} = \texttt{MI}$, hence can be typed by tier **0**, or it is of tier **0** (`MJ` cannot be a sequence of tier **0** instructions). Hence Lemma 6 tells us that every variable assigned to in `MJ` is of tier **0** and there exists $\mathcal{D}'$ such that $(\mathcal{D}, \texttt{MJ}) \to^* (\mathcal{D}', \epsilon)$ and $(\mathcal{C}', \epsilon) \approx_{\Gamma,\Omega} (\mathcal{D}', \epsilon)$. This holds as $\mathcal{C}'$ and $\mathcal{D}'$ match on tier **1** values.
  - If $\Gamma, \Omega \vdash \texttt{MI} : \texttt{void}(\mathbf{1})$ and cannot be given the type $\texttt{void}(\mathbf{0})$ (i.e. both `x` and `e` are of tier **1**) then, by definition of $\approx_{\Gamma,\Omega}$, $\texttt{MI} = \texttt{MJ}$. We let the reader check that the evaluation of `me` leads to the same tier **1** value as it only depends on tier **1** variables (the only difficulty is for non recursive methods that might have tier **0** arguments but this is straightforward as it just consists in inlining the method body that is also safe with respect to Γ and Ω). Consequently, $\mathcal{C}' \approx_{\Gamma,\Omega} \mathcal{D}'$ as the change on tier **1**
- If $\texttt{MI} = \texttt{MI}_1\ \texttt{MI}_2$. There are still two cases to consider:
  - Either `MI` is of tiered type $\texttt{void}(\mathbf{0})$ then so is `MJ` and the result is straightforward.
  - Or `MI` is of tiered type $\texttt{void}(\mathbf{1})$ and $\texttt{MJ} = \texttt{MJ}_1\ \texttt{MJ}_2$ with $\texttt{MI}_i \approx_{\Gamma,\Omega} \texttt{MJ}_i$. By induction hypothesis if $(\mathcal{C}, \texttt{MI}_1) \to (\mathcal{C}', \texttt{MI}_1')$ there exists $(\mathcal{D}', \texttt{MJ}_1')$ such that $(\mathcal{D}, \texttt{MJ}_1) \to^* (\mathcal{D}', \texttt{MJ}_1')$ and $(\mathcal{C}', \texttt{MI}_1') \approx_{\Gamma,\Omega} (\mathcal{D}', \texttt{MJ}_1')$. Consequently, $(\mathcal{C}, \texttt{MI}) \to (\mathcal{C}', \texttt{MI}_1'\ \texttt{MI}_2)$, $(\mathcal{D}, \texttt{MJ}_1\ \texttt{MJ}_2) \to^* (\mathcal{D}', \texttt{MJ}_1'\ \texttt{MJ}_2)$ and $(\mathcal{C}', \texttt{MI}_1'\ \texttt{MI}_2) \approx_{\Gamma,\Omega} (\mathcal{D}', \texttt{MJ}_1'\ \texttt{MJ}_2)$, by definition of $\approx_{\Gamma,\Omega}$.

And so on, for all the other remaining cases. □

Given a configuration $\mathcal{C}$ and a meta-instruction `MI` of a safe program with respect to contextual typing environment Γ and operator typing environment Ω, the *distinct tier* **1** *configuration sequence* $\xi_{\Gamma,\Omega}(\mathcal{C}, \texttt{MI})$ is defined by:

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

$$\xi_{\Gamma,\Omega}(\mathcal{C}, \texttt{MI}) = \begin{cases} \xi_{\Gamma,\Omega}(\mathcal{C}', \texttt{MI}') & \text{if } \mathcal{C} \approx_{\Gamma,\Omega} \mathcal{C}' \\ \mathcal{C}.\xi_{\Gamma,\Omega}(\mathcal{C}', \texttt{MI}') & \text{otherwise} \end{cases}$$

- If $(\mathcal{C}, \texttt{MI}) = (\mathcal{C}, \epsilon)$ then $\xi_{\Gamma,\Omega}(\mathcal{C}, \texttt{MI}) = \mathcal{C}$.

Informally, $\xi_{\Gamma,\Omega}(\mathcal{C}, \texttt{MI})$ is a record of the distinct tier **1** memory configurations encountered during the evaluation of $(\mathcal{C}, \texttt{MI})$. Notice that this sequence may be infinite in the case of a non-terminating program. The relation $\approx_{\Gamma,\Omega}$ is extended to sequences by $\epsilon \approx_{\Gamma,\Omega} \epsilon$ (ϵ being the empty sequence) and $\mathcal{C}.\xi \approx_{\Gamma,\Omega} \mathcal{D}.\xi'$ iff both $\mathcal{C} \approx_{\Gamma,\Omega} \mathcal{D}$ and $\xi' \approx_{\Gamma,\Omega} \xi'$ hold.

Now we can show another non-interference property à la Volpano et al. [3] stating that given a safe program, traces (distinct tier **1** configuration sequence) do not depend on tier **0** variables.

Lemma 8 (Trace non-interference)**.** *Given a meta-instruction* $\texttt{MI}$ *of a safe program with respect to environments* Γ *and* Ω*, let* $\mathcal{C}$ *and* $\mathcal{D}$ *be two memory configurations, if* $\mathcal{C} \approx_{\Gamma,\Omega} \mathcal{D}$ *then* $\xi_{\Gamma,\Omega}(\mathcal{C}, \texttt{MI}) \approx_{\Gamma,\Omega} \xi_{\Gamma,\Omega}(\mathcal{D}, \texttt{MI})$*.*

Proof. By induction on the reduction $\to$ and a case analysis on meta-instructions $\texttt{MI}$:

- If $\texttt{MI} = \texttt{x:=me}$ then there are two cases to consider:
  - either $\Gamma, \Omega \vdash \texttt{MI} : \texttt{void}(\mathbf{0})$. In this case, by Confinement Lemma 6, $\texttt{x}$ is of tier **0** and thus if $(\mathcal{C}, \texttt{MI}) \to (\mathcal{C}', \epsilon)$ and $(\mathcal{D}, \texttt{MI}) \to (\mathcal{D}', \epsilon)$, we have $\mathcal{C}' \approx_{\Gamma,\Omega} \mathcal{C} \approx_{\Gamma,\Omega} \mathcal{D} \approx_{\Gamma,\Omega} \mathcal{D}'$.
  - or $\Gamma, \Omega \vdash \texttt{MI} : \texttt{void}(\mathbf{1})$ and not $\Gamma, \Omega \vdash \texttt{MI} : \texttt{void}(\mathbf{0})$. In this case, $\texttt{x}$ is a tier **1** variable. However as $\mathcal{C} \approx_{\Gamma,\Omega} \mathcal{D}$, $\texttt{me}$ evaluates to the same value or reference under both memory configuration. For example, in the case of a variable assignment (Rule (1) of Figure 4), we have $(\mathcal{C}, [\tau]\ \texttt{x:=y;}) \to (\mathcal{C}[\texttt{x} \mapsto \mathcal{C}(\texttt{y})], \epsilon)$ and $(\mathcal{D}, [\tau]\ \texttt{x:=y;}) \to (\mathcal{D}[\texttt{x} \mapsto \mathcal{D}(\texttt{y})], \epsilon)$. However, as $\mathcal{C} \approx_{\Gamma,\Omega} \mathcal{D}$ and $\texttt{y}$ is of tier **1** by Rule *(Ass)* of Figure 6, $\mathcal{C}(\texttt{y}) = \mathcal{D}(\texttt{y})$. Consequently $\mathcal{C}[\texttt{x} \mapsto \mathcal{C}(\texttt{y})] \approx_{\Gamma,\Omega} \mathcal{D}[\texttt{x} \mapsto \mathcal{D}(\texttt{y})]$. All the other cases of assignments (operator, methods) can be treated in the same manner, the constructor case being excluded as it enforces the output to be of tier **0**.
- If $\texttt{MI} = \texttt{while(x)\{MI'\}}$ then, by Rule *(Wh)* of Figure 6, $\texttt{x}$ is enforced to be of tier **1**. Consequently, $\mathcal{C}(\texttt{x}) = \mathcal{D}(\texttt{x})$ and, consequently, $(\mathcal{C}, \texttt{MI}) \to (\mathcal{C}, \texttt{MI}')$ and $(\mathcal{D}, \texttt{MI}) \to (\mathcal{D}, \texttt{MI}')$, for some $\texttt{MI}'$, independently of whether the guard evaluates to $\texttt{true}$ or $\texttt{false}$. As a consequence, $\xi_{\Gamma,\Omega}(\mathcal{C}, \texttt{MI}') = \xi_{\Gamma,\Omega}(\mathcal{C}, \texttt{MI}) \approx_{\Gamma,\Omega} \xi_{\Gamma,\Omega}(\mathcal{D}, \texttt{MI}) = \xi_{\Gamma,\Omega}(\mathcal{D}, \texttt{MI}')$.

All the other cases for meta-instructions can be treated in the same manner and so the result. □

This Lemma implies that tier **1** variables do not depend on tier **0** variables.

Using Lemma 8, 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.

Corollary 3. *Given a memory configuration $\mathcal{C}$ and a meta-instruction* `MI` *of a safe program with respect to environments Γ and Ω, if $(\mathcal{C}, \texttt{MI}) \rightarrow^+ (\mathcal{C}', \texttt{MI})$ and $\mathcal{C} \approx_{\Gamma,\Omega} \mathcal{C}'$, then the meta-instruction* `MI` *does not terminate on memory configuration $\mathcal{C}$.*

Proof. Assume that during the transition $(\mathcal{C}, \texttt{MI}) \rightarrow^+ (\mathcal{C}', \texttt{MI})$ there is a $\mathcal{C}''$ such that $\mathcal{C}'' \approx_{\Gamma,\Omega} \mathcal{C}$ does not hold, then the distinct tier **1** configuration sequence $\xi_{\Gamma,\Omega}(\mathcal{C}, \texttt{MI})$ contains this $\mathcal{C}''$. From the construction of the sequence, we deduce that $\xi_{\Gamma,\Omega}(\mathcal{C}, \texttt{MI})$ is of the shape $\ldots \mathcal{C}'' \ldots \xi_{\Gamma,\Omega}(\mathcal{C}', \texttt{MI})$. However, by Lemma 8, $\xi_{\Gamma,\Omega}(\mathcal{C}, \texttt{MI}) \approx_{\Gamma,\Omega} \xi_{\Delta_\mathbf{1}}(\mathcal{C}', \texttt{MI})$, hence it is infinite and the meta-instruction `MI` does not terminate on memory configuration $\mathcal{C}$.

Otherwise, we are in a state $(\mathcal{C}, \texttt{MI})$ from which the set of variables of tier **1** will never change and containing either a while loop or a recursive call. Consequently, there is some $\mathcal{C}''$ such that $(\mathcal{C}', \texttt{MI}) \rightarrow^+ (\mathcal{C}'', \texttt{MI})$ and so on. This means that the meta-instruction `MI` does not terminate on $\mathcal{C}$. □

7. Polynomial time soundness

It is possible to bound the number of distinct configurations of tier **1** variables that can be met during the execution of a program, that is the number of different equivalence classes for the $\approx$ relation on configurations.

Lemma 9. *Given a safe program with respect to environments Γ and Ω of computational instruction* `Comp` *on input $\mathcal{I}$, the number of equivalence classes for $\approx_{\Gamma,\Omega}$ on configurations is at most $|\mathcal{I}|^{n_\mathbf{1}}$ where $n_\mathbf{1}$ is the number of tier* **1** *variables in the computational instruction.*

Proof. First, let us note that the nodes and internal edges of tier **1** of the pointer graph do not change: new nodes created by constructors can only be of tier **0** from Rule *(Cons)*. Field assignments can only be of tier **0** according to Rule (*Ass*). Moreover, Rule *(Self)* enforces all the field of an object to have tiers matching that of the current object. Consequently, reference type variables of tier **1** may only point to nodes of the initial pointer graph corresponding to input $\mathcal{I}$. The number of such nodes is bounded by $|\mathcal{I}|$ as $|\mathcal{I}|$ bounds the number of nodes in the initial pointer graph.

Second, we look at primitive type variables. By Definition 3 of operator typing environments, only the output of neutral operators (or of methods of return type of tier **1**) can be applied. Indeed, they are the only operators to have a tier **1** output and in an assignment of the shape $\texttt{x}:=\texttt{op}(\overline{\texttt{y}})$, if `op` is of type $\langle \overline{\tau}(\mathbf{0}) \rangle \rightarrow \tau(\mathbf{0})$ then `x` is enforced to be of tier **0** by Rule *(Ass)* of Figure 6. Neutral operators are operators whose output is either a value of a constant domain (`boolean`, `char`, ...) and, hence, has a constant number of distinct

values, or whose output is a positive integer value smaller than one of its input (also tier **1** values by Definition 3). Consequently, they can have a number of distinct values in $O(|\mathcal{I}|)$, as, by definition, $|\mathcal{I}|$ bounds the initial primitive values stored in the initial pointer mapping.

To conclude, let the number of tier **1** variables be $n_{\mathbf{1}}$, the number of distinct possible configurations is $|\mathcal{I}|^{n_{\mathbf{1}}}$. □

Now we can show the soundness: a safe and terminating program terminates in polynomial time.

Theorem 2 (Polynomial time soundness)**.** *If an AOO program of computational instruction* `Comp` *is safe with respect to environments* Γ *and* Ω *and terminates on input* $\mathcal{I}$*:*

$$(\mathcal{I}, \underline{\texttt{Comp}}) \to^k (\mathcal{C}', \epsilon)$$

then:

$$k = O(|\mathcal{I}|^{n_{\mathbf{1}}(\nu+\lambda)}).$$

Proof. For simplicity, each recursive call can be simulated through derecursivation by a while loop instruction of intricacy λ (keeping the number of reduction steps preserved relatively to some multiplicative and additive constants). Indeed recursive calls do not contain while loops and have only one method call in their body. Consequently, the maximum intricacy of an equivalent program with no recursive call is $\lambda + \nu$. Now we prove the result by induction on the intricacy ν of the transformed program:

- if $\nu = 0$ then the program has no while loops. Consequently, $k = O(1) = O(|\mathcal{I}|^{n_{\mathbf{1}} \times 0})$
- if $\nu = k + 1$ then, by definition of intricacy, the program contains at least one while loop. Let `while(x){MI}` be the first outermost while loop of the program. We have that $\nu(\texttt{MI}) = k$. By Induction hypothesis, `MI` can be evaluated in time $O(|\mathcal{I}|^{n_{\mathbf{1}}k}$, that is $O(|\mathcal{I}|^{n_{\mathbf{1}}(\nu-1)})$. By Lemma 9, the tier **1** variable `x` can take at most $O(|\mathcal{I}|^{n_{\mathbf{1}}})$ distinct values. Consequently, by termination assumption, the evaluation of `while(x){MI}` will take at most $O(|\mathcal{I}|^{n_{\mathbf{1}}} \times |\mathcal{I}|^{n_{\mathbf{1}}(\nu-1)})$ steps, that is $O(|\mathcal{I}|^{n_{\mathbf{1}}\nu})$ steps. Indeed, by Corollary 3, we know that a terminating program cannot reach twice the same configuration on tier **1** variables for a fixed meta-instruction. Now it remains to see that the instruction in the context have only while loops of intricacy smaller than ν. So, by induction again, they can be evaluated in time $O(|\mathcal{I}|^{n_{\mathbf{1}}\nu})$. Finally the total time is the sum so in time $O(|\mathcal{I}|^{n_{\mathbf{1}}\nu})$. □

The same kind of results can be obtained for generally safe programs:

Proposition 4. *If an AOO program of computational instruction* `Comp` *is generally safe with respect to environments* Γ *and* Ω *and terminates on input* $\mathcal{I}$*:*

$$(\mathcal{I}, \underline{\texttt{Comp}}) \to^k (\mathcal{C}', \epsilon)$$

then:

$$k = O(|\mathcal{I}|^{n_{\mathbf{1}}(\nu+\lambda)}).$$

As a side effect, we obtain polynomial upper bounds on both the stack size and the heap size of safe terminating programs:

Theorem 3 (Heap and stack size upper bounds)**.** *If an AOO program of computational instruction* `Comp` *is safe with respect to environments* Γ *and* Ω *and terminates on input* $\mathcal{I}$ *then for each memory configuration* $\mathcal{C} = \langle \mathcal{H}, \mathcal{S}_\mathcal{H} \rangle$ *and meta-instruction* `MI` *such that* $(\mathcal{I}, \underline{\texttt{Comp}}) \to^* (\mathcal{C}, \texttt{MI})$ *we have:*

1. $|\mathcal{H}| = O(\max(|\mathcal{I}|, |\mathcal{I}|^{n_1(\nu+\lambda)}))$
2. $|\mathcal{S}_\mathcal{H}| = O(|\mathcal{I}|^{n_1(\nu+2\lambda)})$

Proof. (1) By Theorem 2, the number of reductions is in $O(|\mathcal{I}|^{n_1(\nu+\lambda)})$. The only instructions making the heap (pointer graph) increase are constructor calls. The number of such calls is thus bounded by $O(|\mathcal{I}|^{n_1(\nu+\lambda)})$ and consequently the heap size is bounded by the size of the original heap (that is in $O(|\mathcal{I}|)$) plus the size of the added nodes. Consequently, $|\mathcal{H}| = O(\max(|\mathcal{I}|, |\mathcal{I}|^{n_1(\nu+\lambda)}))$ as if $f = O(f')$ and $g = O(g')$ then $f + g = O(\max(f', g'))$.

(2) The number of stack frames added is bounded by $O(|\mathcal{I}|^{n_1\lambda})$ as, for some recursive method call of signature s, each level of recursion may add at most $O(|\mathcal{I}|^{n_1})$ stack frames corresponding to method signatures in the set $[s]$. Remember that a stack frame is just a signature of constant size 1 and a pointer mapping. The size of a pointer mapping is constant on boolean, character and reference type variables (it is equal to 1) and corresponds to the discrete value stored for numerical primitive type variables. As such values may only augment by a constant for each call to a positive operator and, as such calls may happen $O(|\mathcal{I}|^{n_1(\nu+\lambda)})$ times, we obtain that each stack frame has size bounded by $O(|\mathcal{I}|^{n_1(\nu+\lambda)})$. This is also due to the fact that no fresh variable is generated: consequently, the domain of pointer mapping is constantly bounded by the flattened program size, that is linear in the original program size by Corollary 1. Here we clearly understand the advantage to deal with flattened programs! All together, we obtain that $|\mathcal{S}_\mathcal{H}| = O(|\mathcal{I}|^{n_1\lambda} \times |\mathcal{I}|^{n_1(\nu+\lambda)})$, that is $O(|\mathcal{I}|^{n_1(\nu+2\lambda)})$. □

Again the same results hold for generally safe programs:

Corollary 4. *If an AOO program of computational instruction* `Comp` *is generally safe with respect to environments* Γ *and* Ω *and terminates on input* $\mathcal{I}$ *then for each memory configuration* $\mathcal{C} = \langle \mathcal{H}, \mathcal{S}_\mathcal{H} \rangle$ *and meta-instruction* `MI` *such that* $(\mathcal{I}, \underline{\texttt{Comp}}) \to^* (\mathcal{C}, \texttt{MI})$ *we have:*

1. $|\mathcal{H}| = O(\max(|\mathcal{I}|, |\mathcal{I}|^{n_1(\nu+\lambda)}))$
2. $|\mathcal{S}_\mathcal{H}| = O(|\mathcal{I}|^{n_1(\nu+2\lambda)})$

Example 17. *The AOO program of Example 7. This program is clearly terminating and safe (there is no recursive method call) with respect to the provided environments. Moreover intricacy* ν *is equal to* 1 *since there is no nested while loops in the method* `getTail`*. Moreover, its level is equal to* 0 *as there is no recursive call. Moreover there is one tier* **1** *variable* $n_\mathbf{1} = 1$*. Consequently, applying*

Theorem 2, we obtain that it terminates in $O(n^1)$, on some input of size n. Moreover, by Corollary 4, the heap size and stack size are in $O(\max(n, n^1)) = O(n)$ and $O(n^{1(1+2\times 0)}) = O(n)$.

Example 18. *Consider the below example representing cyclic data:*

```
Ring {
  boolean data;
  Ring next;
  Ring prev;

  Ring(boolean d, Ring old) {
    data = d;
    if (old == null) {
        next = this;
        prev = this;
    } else {
        next = old.next;
        next.setPrev(this);
        prev = old.prev;
        prev.setNext(this);
    }
  }

  boolean getData() {return data;}
  Ring getNext() {return next;}
  Ring getPrev() {return prev;}
  void setPrev(Ring p) {prev = p;}
  void setNext(Ring n) {next = n;}
}

Exe {
  void main() {
    // Init
    Ring a = new Ring(true, null);
    Ring input = new Ring(true, a);
    //Comp: Search for a false in the input.
    copy¹ = input¹;
    while (copy¹.getData() != false) {
        copy¹ = copy¹.getNext();
    }
  }
}
```

The program is safe and can be typed with respect to the following judgments:

- $\texttt{getData()} : \texttt{Ring}(\mathbf{1}) \to \texttt{boolean}(\mathbf{1})$
- $\texttt{getNext()} : \texttt{Ring}(\mathbf{1}) \to \texttt{Ring}(\mathbf{1})$

with respect to environments Γ and Ω such that $\Gamma(\texttt{copy}) = \Gamma(\texttt{input}) = \texttt{Ring}(\mathbf{1})$. Notice that methods `setNext(Ring n)` *and* `setPrev(Ring p)` *are not required to*

be typed with respect to tiers as they only appear in the initialization instruction and, consequently, their complexity is not under control (they are just supposed to build the input). 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`*. We obtain a bound through the use of Theorem 2, that is* $O(n^{n_1 \times 1}) = O(n^2)$*. Notice that this bound can be ameliorated to* $O(n)$ *at the price of a non-uniform formula by noticing that only* 1 *tier* $\mathbf{1}$ *variable occurs in the while loop (see the remarks about declassification). Notice that this program could be adapted and typed in while loops of the shape:*

```
while(copy.getData() != false && n>0){
  ...
  n--;
}
```

and this would be still typable.

Consequently, we can see that the presented methodology does not only apply to trivial data structures but can take benefit of any complex object structure.

8. Completeness

Another direct result is that this characterization is complete with respect to the class of functions computable in polynomial time as a direct consequence of Marion's result [5] since both language and type system can be viewed as an extension of the considered imperative language. This means that the type system has a good expressivity. We start to show that any polynomial can be computed by a safe and terminating program. Consider the following method of some class `C` computing addition:

```
add(int x, int y){
  while(x¹>0){
    x¹ := x¹-1; :1
    y⁰ := y⁰+1; :0
  }
  return y⁰;
}
```

It can be typed by $\mathtt{C}(\beta) \times \mathtt{int}(\mathbf{1}) \times \mathtt{int}(\mathbf{0}) \to \mathtt{int}(\mathbf{0})$, for any tier β, under the typing environment Δ such that $\Delta(\mathtt{add}^{\mathtt{C}})(\mathtt{x}) = \mathtt{int}(\mathbf{1})$ and $\Delta(\mathtt{add}^{\mathtt{C}})(\mathtt{y}) = \mathtt{int}(\mathbf{0})$. Notice that `x` is enforced to be of tier $\mathbf{1}$, by Rules *(Wh)* and *(Op)* as it appears in the guard of a while loop. The operators > 0 and -1 are neutral. Consequently, they can be given the tiered types $\mathtt{int}(\mathbf{1}) \to \mathtt{boolean}(\mathbf{1})$ and, $\mathtt{int}(\mathbf{0}) \to \mathtt{int}(\mathbf{0})$, respectively, in Rule *(Op)*. The operator $+1$ is positive and its tiered type is enforced to be $\mathtt{int}(\mathbf{0}) \to \mathtt{int}(\mathbf{0})$ by Rule *(Op)*.

Consider the below method encoding multiplication:

```
mult(int x, int y){
  int z⁰ := 0;
  while(x¹>0){
```

```
    x¹ := x¹-1;
    int u¹ := y¹;
    while(u¹>0){
      u¹ := u¹-1;
      z⁰ := z⁰+1;
    }
  }
  return z⁰;
}
```

It can be typed by $\mathtt{C}(\beta)\times\mathtt{int}(\mathbf{1})\times\mathtt{int}(\mathbf{1})\to\mathtt{int}(\mathbf{0})$, for any tier β, under the typing environment Δ such that $\Delta(\mathtt{mult}^{\mathtt{C}})(\mathtt{x})=\Delta(\mathtt{mult}^{\mathtt{C}})(\mathtt{y})=\Delta(\mathtt{mult}^{\mathtt{C}})(\mathtt{u})=\mathtt{int}(\mathbf{1})$ and $\Delta(\mathtt{mult}^{\mathtt{C}})(\mathtt{z})=\mathtt{int}(\mathbf{0})$. Notice that `x` and `u` are enforced to be of tier **1**, by Rules *(Wh)* and *(Op)*. Moreover `y` is enforced to be of tier **1**, by Rule *(Ass)* applied to instruction `int u=y;`, `u` being of tier **1**. Finally, `z` is enforced to be of tier **0**, by Rule *(Op)*, as its stored value increases in the expression `z+1;`.

Theorem 4 (Completeness)**.** *Each function computable in polynomial time by a Turing Machine can be computed by a safe and terminating program.*

Proof. We show that every polynomial time function over binary words, encoded using the class `BList`, can be computed by a safe and terminating program. Consider a Turing Machine TM, with one tape and one head, which computes within n^k steps for some constant k and where n is the input size. The tape of TM is represented by two variables `x` and `y` which contain respectively the reversed left side of the tape and the right side of the tape. States are encoded by integer constants and the current state is stored in the variable `state`. We assign to each of these three variables that hold a configuration of TM the tier **0**. A one step transition is simulated by a finite cascade of if-commands of the form:

```
if(y.getHead()⁰){
        if(state⁰ == 8⁰){
                state⁰ = 3⁰;:0
                x⁰ = new BList(false,x⁰);:0
                y⁰ = y.getTail()⁰);:0
        }else{...:0}
}
```

The above command expresses that if the current read symbol is `true` and the state is 8, then the next state is 3, the head moves to the right and the read symbol is replaced by `false`. The methods `getTail()` and `getHead()` can be given the types $\mathtt{BList}(\mathbf{0})\to\mathtt{BList}(\mathbf{0})$ and $\mathtt{BList}(\mathbf{0})\to\mathtt{boolean}(\mathbf{0})$, respectively (see previous Example). Since each variable inside the above command is of tier **0**, the tier of the if-command is also **0**. As shown above, any polynomial can be computed by a safe and terminating program: we have already provided the programs for addition and multiplication and we let the reader check that it can be generalized to any polynomial. Thus the cascade of one step transitions can be included in an intrication of while loops computing the requested polynomial.

Note that this is possible as the cascade will be of tiered type $\texttt{void}(\mathbf{0})$ and it can be typed by $\texttt{void}(\mathbf{1})$ through the use of Rule *(ISub)*. □

Corollary 5. *Each function computable in polynomial time by a Turing Machine can be computed by a generally safe and terminating program.*

Proof. By Proposition 2. □

9. Type inference

Now we show a decidability result for type inference in the case of safe programs.

Proposition 5 (Type inference)**.** *Given an AOO program P, deciding if there exist a typing environment Δ and an operator typing environment Ω such that P is well-typed can be done in time polynomial in the size of the program.*

Proof. We work on the flattened program. Type inference can be reduced to a 2-SAT problem. We encode the tier of each field $\texttt{x}$ (respectively instruction $\texttt{MI}$) within the method $\texttt{m}$ of class $\texttt{C}$ by a boolean variable $x^{\texttt{m}^{\texttt{C}}}$ (respectively $I^{\texttt{m}^{\texttt{C}}}$) that will be true if the variable (instruction) is of tier $\mathbf{1}$, false if it is of tier $\mathbf{0}$ in in the context of $\texttt{m}^{\texttt{C}}$. Then we generate boolean clauses with respect to the program code:

- An assignment $\texttt{x} := \texttt{y}$; corresponds to $(y^{\texttt{m}^{\texttt{C}}} \wedge x^{\texttt{m}^{\texttt{C}}})$
- A sequence $\texttt{MI}_1\ \texttt{MI}_2$ corresponds to $(I_1^{\texttt{m}^{\texttt{C}}} \vee I_2^{\texttt{m}^{\texttt{C}}})$
- An $\texttt{if(x)\{MI}_1\texttt{\}else\{MI}_2\texttt{\}}$ corresponds to $(\neg I_1^{\texttt{m}^{\texttt{C}}} \vee x^{\texttt{m}^{\texttt{C}}}) \wedge (\neg I_2^{\texttt{m}^{\texttt{C}}} \vee x^{\texttt{m}^{\texttt{C}}})$
- A expression involving a neutral operator $\texttt{x}_0{:=}\texttt{op}(\texttt{x}_1, \ldots, \texttt{x}_\texttt{n})$ corresponds to $\wedge_{i,j}(x_i^{\texttt{m}^{\texttt{C}}} \vee \neg x_j^{\texttt{m}^{\texttt{C}}})$
- and so on...

The number of generated clauses is polynomial in the size of the flattened program and also of the initial program by Corollary 1. The polynomiality comes from the fact that a method may be given distinct types on several calls (thus its body might by typed several times statically). Consequently, tiered types are inferred in polynomial time as 2-SAT problems can be solved in linear time. □

Corollary 6. *Given an AOO program P, deciding if there exist a typing environment Δ and an operator typing environment Ω such that P is well-typed and safe can be done in time polynomial in the size of the program.*

Proof. Using Lemma 5. □

Just remark that the above corollary becomes false for generally safe programs as a consequence of Proposition 3.

10. Extensions

In this section, we discuss several possible improvements of the presented methodology.

10.1. Control flow alteration

Constructs altering the control flow like break, return and continue can also be considered in this fragment. For example, a break statement has to be constrained to be of tiered type $\texttt{void}(\mathbf{1})$ 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:

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

This prevents the programmer from writing conditionals of the shape:

$$\texttt{while}(\texttt{x}^{\mathbf{1}})\{\ldots\texttt{if}(\texttt{y}^{\mathbf{0}})\{\texttt{break;}\}\texttt{else}\{\ldots\}\ldots\}$$

that would break the non-interference result of Lemma 8. Such a conditional cannot be typed in such a way since `y` has to be of tier **1** by Rules *(If)* and *(Break)*. Notice that Theorem 1 remains valid since in a *terminating program*, an instruction containing break statements will have an execution time smaller than the same instruction where the break statements have been deleted. 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. In previous Section, we have made the choice not to include break statement in the code as this make the definition of a formal semantics much more difficult: at any time, we need to keep in mind the innermost loop executed. The same remarks hold for the continue construct.

Another possible extension is to consider methods with no restriction on the return statement. For the simplicity of the type system, we did not present a flexible treatment of the return statements by allowing the execution to leave the current subroutine anywhere in the method body. But this can be achieved in the same manner. The difficulty here is that all the return constructs have to be of the same tiered type (this is a global check on the method body). Moreover, as usual, we need to enforce that in a conditional all reachable flows lead to some return construct.

10.2. Static methods, static variables and access modifiers

The exposed methodology can be extended without any problem to static methods since they can be considered as a particular case of methods. In a similar manner, static variables can be captured since they are global (and can be considered as variables of the executable class) and add no complexity to the program. We claim that the current analysis can handle all usual access modifiers. Indeed the presented work is based on the implicit assumption that all fields are `private` since there is no field access in the syntax. On the opposite, methods and classes are all `public`. Consequently, method access restriction only consists in restricting the class of analyzed programs.

10.3. Abstract classes and interfaces

Both abstract classes and interfaces can be analyzed by the presented framework. This is straightforward for interfaces since they do not add any complexity to the program. We claim that abstract classes can be analyzed since they are just a particular and simple case of inheritance.

10.4. Garbage Collecting

In the pointer graph semantics, dereferenced objects stay in memory forever. It does not entail the bound on the heap size, however, it is far from optimal from a memory usage point of view. However, it is easy to add naive garbage collecting to the system. Indeed, finding which objects in the graph are dereferenced is simply recursively deleting nodes whose indegree is zero (counting pointers in the indegree).

Two different strategies can be thought of to implement dynamically this idea: either use an algorithm similar to the mark-and-don't-sweep algorithm that will color the part of the graph that is referenced then delete the uncolored part; or noting that as assignments and `pop`; are the only instructions that may dereference a pointer, maintain for each node of the graph a counter for its indegree, update it at each instruction and delete the dereferenced nodes (and its children if they have no other parents) whenever it happens.

The first strategy has the main drawback that exploring the whole graph will block the execution for some time, especially as starting from each node in each pointer mapping of the pointer stack is needed.

The second strategy will be far more flexible and the real wiping of memory may be deferred if necessary (as the deleting of files in a Unix system for example). An assignment increments the indegree of the node assigned and decrements the indegree of the node previously assigned to the variable. An instantiation (`new`) adds arrows in the graph, hence increments the indegree of all the nodes associated to its parameter. `push` and `pop` respectively increment and decrement the indegree of the parameters and the current object of the method. Whenever a node's indegree reaches zero, it can be deleted. Finally, deleting a node of the graph decrements the indegree of the nodes it pointed to.

10.5. Alleviating the safety condition

Another way to alleviate the safety condition is to reuse the distinct recursion schemata provided in the ICC works of the function algebra. For example, over `Tree` structures, a recursive definition -written in a functional manner - of the shape:

```
f(t) = new Tree(f(t.getLeft()),f(t.getRight()));
```

should be accepted by the analysis as recursive calls work on partitioned data and, consequently, the number of recursive calls remains linear in the input size.

One sufficient condition to ensure that property is that:

- recursive calls of the method body are performed on distinct fields

- and these fields are never assigned to in the method body

The following depth first tree traversal algorithm satisfies this condition:

```
class Tree {
  int node;
  Tree left;
  Tree right;
  ...
  void visit() {
    println(node);
    if(left != null){
      left.visit();
    }else{;}
    if(right != null){
      right.visit();
    }else{;}

  }
}
```

on the assumption that the method `println(n)` behaves as expected. We let the reader check that the method `visit()` can be typed by $\texttt{Tree}(\mathbf{1}) \to \texttt{void}(\mathbf{1})$.

10.6. Declassification

In non-interference settings, declassification consists in lowering the confidentiality level of part of the data. For example, when verifying a password, the password database itself is at the highest level, knowing whether an input password matches should have the same level of confidentiality. As this is highly impractical, declassifying this partial information makes sense.

In this context, declassifying would mean retyping some tier **0** variables into **1**. Such a flow is strictly forbidden by the type system, but it would make sense, for example, to compose treatments that are separately well-typed. As long as those treatments are in finite number, we keep the polynomial bound as bounded composition of polynomials remains polynomial.

Formally, we will say that programs of the form `Exe{void main(){Init` $\texttt{I}_1$ $\texttt{I}_2$ ... $\texttt{I}_n$`}}` are well-typed iff for each $i \leq n$, `Exe{void main(){Init` $\texttt{I}_1$ ... $\texttt{I}_i$`}}` is well-typed when we consider `Init` $\texttt{I}_1$ ... $\texttt{I}_{i-1}$ to be the initialization instruction.

Example 19. *The following program:*

```
Exe {
  void main() {
    //Init
    int n := ...;
    BList b := null;
    while (n>0) {
      b := new BList(true, b);
      n := n-1;
```

```
    }
    //Comp1
    z := 0;
    while (y.getQueue()) {
      z := z+1
    }
    //Comp2
    x := z;
    BList c := null;
    while (x>0) {
      c := new BList(false, c);
    }
    //Comp3
    x := z;
    while (x>0) {
      c := new BList(true, c);
    }
  }
}
```

cannot be typed without declassification as in `Comp1`*,* `z` *needs to be of tier* **0***, while in* `Comp2` *and* `Comp3`*, it needs to be of tier* **1**.

Remark 2. *Note that the proof of Theorem 4 could be improved by using declassification. Indeed, we could write a first computation instruction that creates a polynomial bound on the number of steps of the Turing Machine, and a second computation instruction that executes the simulation of each step while a counter is lower than the bound. In the first part, the bound needs to be of tier* **0***, in the second of tier* **1***, hence the use of declassification.*

11. Related works

11.1. Related works on tier-based complexity analysis

The current work is inspired by three previous works:

- the seminal paper [5], initiating imperative programs type-based complexity analysis using secure information flow and providing a characterization of polynomial time computable functions,
- the paper [10], extending previous analysis to C processes with a fork/wait mechanism, which provides a characterization of polynomial space computable functions,
- and the paper [11], extending this methodology to a graph based language.

The current paper tries to pursue this objective but on a distinct paradigm: Object. It differs from the aforementioned works on the following points:

- first, it is an extension to the object-oriented paradigm (although imperative features can be dealt with). In particular, it characterizes the complexity of recursive and non-recursive method calls whereas previous works [5, 10, 11] where restricted to while loops and to non-object data type (words in [5, 10] and records in [11]),
- second, it studies program intensional properties (like heap and stack) whereas previous papers were focusing on the extensional part (characterizing function spaces). Consequently, it is closer to a programmer's expectations,
- 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.
- last, from an expressivity point of view, the presented results strictly extend the ones of [5], as the restriction of this paper to primitive data types is mainly the result of [5], while they are applied on a more concrete language than the ones in [11].

The current work is an extended version of [4]. The main distinctions are the following:

- In [4], only general safety is studied. Here we have presented a decidable (and thus restricted) safety condition.
- Contrarily to [4], the paper presents a formal semantics. The pros are a cleaner theoretical treatment. The cons are that we do not have constructs changing the flow like "break". Such constructs can be handled by the tier-based type system. However, their use would increase drastically the complexity of the semantics as a program counter would be required.
- The requirement that a recursive method can only be called in a tier **1** instruction is formalized through the use of the $\vdash_{\mathbf{1}}$ notation while it was only informally stated in [4].

11.2. Other related works on complexity

There are several related works on the complexity of imperative and object oriented languages. On imperative languages, the papers [12, 13, 14] study theoretically the heap-space complexity of core-languages using type systems based on a matrices calculus.

On OO programming languages, the papers [15, 16] control the heap-space consumption using type systems based on amortized complexity introduced in previous works on functional languages [17, 18, 19]. Though similar, the presented result differs on several points with this line of work. First, this analysis is not restricted to linear heap-space upper bounds. Second, it also applies to stack-space upper bounds. Last but not least, this language is not restricted to

the expressive power of method calls and includes a `while` statement, controlling the interlacing of such a purely imperative feature with functional features like recurrence being a very hard task from a complexity perspective.

Another interesting line of research is based on the analysis of heap-space and time consumption of Java bytecode [20, 21, 22, 23]. The results from [20, 21, 22] 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 [23] and focuses on certifying memory bounds for Java Card. The 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 the programmer to deal with this typing discipline on an abstract OO code very close to the original Java code without considering the corresponding Java bytecode. Moreover, this 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.

The concerns of this study are also related to the ones of [24, 25], that try to predicts the minimum amount of heap space needed to run the program without exhausting the memory, but the methodology, the code analyzed (source vs compiled) and the goals differ.

A complex type-system that allows the programmer to verify linear properties on heap-space is presented in [26]. The presented 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 [27, 28]. These works rely on a programming language called PURPLE combining imperative statements together with pointers on a fixed graph structure. Although not directly related, the presented type system was inspired by this work.

11.3. Related works on termination

The presented work is independent from termination analysis but the 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 3 only hold for a terminating computation. Consequently, this 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* [29, 30], tools like Terminator [31] based on *Transition predicate abstraction* [32] or symbolic complexity bound generation based on abstract interpretations, see [33, 34] for example.

12. Conclusion

This work presents a simple but highly expressive type-system that is sound and complete with respect to the class of polynomial time computable functions, that can be checked in polynomial time and that provides explicit polynomial upper bounds on the heap size 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 18). 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 this analysis.

This analysis has several advantages:

- It provides a high-level alternative to usual complexity studies mainly based on the compiled bytecode. The analysis is high-level but the obtained bound are quite tight.
- It is decidable in polynomial time on the safety criterion.
- It merges both theoretical and applied results as we both obtain bounds on real programs and a sound and complete characterization of polynomial time computable functions.
- It is able to deal with the complexity of programs whose loops are guarded by object. This is not a feature of bytecode-based approaches that are restricted to primitive data and are in need of costly program transformation techniques and tools to comply with such kind of programs.
- It uses previous theoretical techniques (tiering, safe recursion on notation) for functional programs and function algebra and shows that they can be adapted elegantly (though technically) to the OO paradigm.
- It uses previous security techniques (non-interference, declassification) for imperative programs. The use is slightly different (even orthogonal) but the methodology is surprisingly very close.

We expect this paper to be a first step towards the use of tiers and non-interference for controlling the complexity of OO programs. The next steps are the design of a practical application and extensions to (linear or polynomial) space using threads.

Acknowledgments. The authors gratefully acknowledge the advises and comments from anonymous referees that contributed to improving this article.

[1] S. Bellantoni, S. Cook, A new recursion-theoretic characterization of the poly-time functions, Comput. Complex. 2 (1992) 97–110.

[2] D. Leivant, J.-Y. Marion, Lambda calculus characterizations of poly-time, Fundam. Inform. 19 (1/2) (1993) 167–184.

[3] D. Volpano, C. Irvine, G. Smith, A sound type system for secure flow analysis, J. Computer Security 4 (2/3) (1996) 167–188.

[4] E. Hainry, R. Péchoux, Objects in Polynomial Time, in: Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Vol. 9458 of Lecture Notes in Computer Science, 2015, pp. 387–404.

[5] J.-Y. Marion, A type system for complexity flow analysis, in: 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, 2011, pp. 123–132.

[6] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight java: a minimal core calculus for java and GJ, ACM Trans. Program. Lang. Syst. 23 (3) (2001) 396–450.

[7] J.-Y. Girard, Light linear logic, Inf. Comput. 143 (2) (1998) 175–204.

[8] S. A. Cook, C. Rackoff, Space lower bounds for maze threadability on restricted machines, SIAM J. Comput. 9 (3) (1980) 636–652.

[9] N. Danner, J. S. Royer, Ramified structural recursion and corecursion, CoRR abs/1201.4567.
URL `http://arxiv.org/abs/1201.4567`

[10] E. Hainry, J.-Y. Marion, R. Péchoux, Type-based complexity analysis for fork processes, in: FOSSACS, Vol. 7794 of Lecture Notes in Computer Science, 2013, pp. 305–320.

[11] D. Leivant, J.-Y. Marion, Evolving graph-structures and their implicit computational complexity, in: Automata, Languages, and Programming - ICALP 2013, Vol. 7966 of Lecture Notes in Computer Science, 2013, pp. 349–360.

[12] K.-H. Niggl, H. Wunderlich, Certifying polynomial time and linear/polynomial space for imperative programs, SIAM J. Comput. 35 (5) (2006) 1122–1147.

[13] J.-Y. Moyen, Resource control graphs, ACM Trans. Comput. Logic 10 (4) (2009) 29:1–29:44.

[14] N. D. Jones, L. Kristiansen, A flow calculus of *wp*-bounds for complexity analysis, ACM Trans. Comput. Log. 10 (4).

[15] M. Hofmann, S. Jost, Type-based amortised heap-space analysis, in: ESOP, Vol. 3924 of Lecture Notes in Computer Science, 2006, pp. 22–37.

[16] M. Hofmann, D. Rodriguez, Efficient type-checking for amortised heap-space analysis, in: CSL, Vol. 5771 of Lecture Notes in Computer Science, 2009, pp. 317–331.

[17] M. Hofmann, S. Jost, Static prediction of heap space usage for first-order functional programs, in: Symposium on Principles of Programming Languages, POPL 2003, ACM, 2003, pp. 185–197.

[18] S. Jost, K. Hammond, H.-W. Loidl, M. Hofmann, Static determination of quantitative resource usage for higher-order programs, in: Symposium on Principles of Programming Languages, POPL 2010, 2010, pp. 223–236.

[19] L. Beringer, M. Hofmann, A. Momigliano, O. Shkaravska, Automatic certification of heap consumption, in: LPAR, Vol. 3452 of Lecture Notes in Computer Science, 2004, pp. 347–362.

[20] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Costa: Design and implementation of a cost and termination analyzer for java bytecode, in: FMCO, Vol. 5382 of Lecture Notes in Computer Science, 2008, pp. 113–132.

[21] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of object-oriented bytecode programs, Theor. Comput. Sci. 413 (1) (2012) 142–159.

[22] R. Kersten, O. Shkaravska, B. van Gastel, M. Montenegro, M. C. J. D. van Eekelen, Making resource analysis practical for real-time java, in: Java Technologies for Real-time and Embedded Systems, JTRES '12, 2012, pp. 135–144.

[23] D. Cachera, T. Jensen, D. Pichardie, G. Schneider, Certified memory usage analysis, in: FM 2005: Formal Methods, Vol. 3582 of Lecture Notes in Computer Science, 2005, pp. 91–106.

[24] E. Albert, S. Genaim, M. Gómez-Zamalloa Gil, Live heap space analysis for languages with garbage collection, in: Proceedings of the 2009 international symposium on Memory management, ACM, 2009, pp. 129–138.

[25] E. Albert, S. Genaim, M. Gómez-Zamalloa, Parametric inference of memory requirements for garbage collected languages, in: ACM Sigplan Notices, Vol. 45, ACM, 2010, pp. 121–130.

[26] W. Chin, H. Nguyen, S. Qin, M. Rinard, Memory usage verification for OO programs, in: Static Analysis, SAS 2005, 2005, pp. 70–86.

[27] M. Hofmann, U. Schöpp, Pointer programs and undirected reachability, in: 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, IEEE Computer Society, 2009, pp. 133–142.

[28] M. Hofmann, U. Schöpp, Pure pointer programs with iteration, ACM Trans. Comput. Log. 11 (4).

[29] A. M. Ben-Amram, Size-change termination, monotonicity constraints and ranking functions, Log. Meth. Comput. Sci. 6 (3).

[30] A. M. Ben-Amram, S. Genaim, A. N. Masud, On the termination of integer loops, in: Verification, Model Checking, and Abstract Interpretation, VMCAI 2012, Vol. 7148 of Lecture Notes in Computer Science, 2012, pp. 72–87.

[31] B. Cook, A. Podelski, A. Rybalchenko, Terminator: Beyond safety, in: Computer Aided Verification, CAV 2006, Vol. 4144 of Lecture Notes in Computer Science, 2006, pp. 415–426.

[32] A. Podelski, A. Rybalchenko, Transition predicate abstraction and fair termination, in: Symposium on Principles of Programming Languages, POPL 2005, ACM, 2005, pp. 132–144.

[33] S. Gulwani, Speed: Symbolic complexity bound analysis, in: Computer Aided Verification, CAV 2009, Vol. 5643 of Lecture Notes in Computer Science, 2009, pp. 51–62.

[34] S. Gulwani, K. K. Mehra, T. M. Chilimbi, Speed: precise and efficient static estimation of program computational complexity, in: Symposium on Principles of Programming Languages, POPL 2009, ACM, 2009, pp. 127–139.