Objects in Polynomial Time

Emmanuel Hainry, Romain Péchoux

Objects in Polynomial Time*

Emmanuel Hainry and Romain Péchoux

Université de Lorraine, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54506, France
Inria, Villers-lès-Nancy, F-54600, France
CNRS, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54506, France
{hainry,pechoux}@loria.fr

Abstract. A type system based on non-interference and data ramification principles is introduced in order to capture the set of functions computable in polynomial time on OO programs. The studied language is general enough to capture most OO constructs and our characterization is quite expressive as it allows the analysis of a combination of imperative loops and of data ramification scheme based on Bellantoni and Cook's safe recursion using function algebra.

Introduction

Motivations. This paper presents a first characterization of polynomial time computable functions on the Object Oriented paradigm. This characterization is obtained by mixing non-interference techniques à la Volpano et al. [16], for secure flow analysis, with tiering [13] or safe recursion on notation [3] based approaches, in order to capture complexity classes over function algebra.

The main idea is very simple: program data is divided in two classes (called tier **0** and tier **1**); while tier **0** data store computations that may increase, tier **1** data are used for the control flow and never increase. This is ensured by a type system based on non-interference precluding data flows from **0** to **1** and allowing only specific flows from **1** to **0** (when data are cloned). The type system also ensures that data of tier **1** is never modified once initialized and voilà ! As the control flow is only based on tier **1**, then either the program does not terminate or it terminates in polynomial time as tier **1** variables may only point to a number of distinct memory references bounded by the input size.

Though the idea is simple, its implementation is not. Analyzing the complexity of OO programs faces a lot of hardships (complex object data structure, side effects, recursive methods, ...) and one mistake in the application can make the whole system collapse in a dominolike effect. This is the reason why the type system presented in this paper is an original non-straightforward contribution.

There are plenty of works analyzing the complexity of OO programs from a practical perspective in the literature. These works are interesting and sometimes better than the present paper wrt program expressivity. However they suffer

* This work was partially supported by ANR-14-CE25-0005 Elica: Expanding Logical Ideas for Complexity Analysis.

from some of the following gaps that our contribution tries to tackle: first, the theoretical part can be lacking as some works focus on having effective proofs of concept rather than providing characterizations; second, they usually do not provide any completeness result; finally, many works analyze a bytecode rather than the sourcecode itself, thus avoiding the treatment of complex constructs. This makes the analysis more precise but clearly less portable. Though this choice may be better from a practical perspective, it is constrained by one language implementation. In contrast, our work is more conceptual and forges ties between various programming paradigms: the tiering methodology works for functional programming [3], imperative programs [15], fork processes [10], graph-based language [14]. We show in this paper that it also works in the object paradigm.

The analysis presented in this paper is based on an abstract OO language. Consequently, it can be applied both to impure OO languages (*e.g.* Java) and to pure ones (*e.g.* SmallTalk or Ruby). 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, ...). The only restriction is that it does not handle pointer arithmetics. Hence languages such as C++ cannot be handled. However, it merges elegantly with the functional world as it captures Safe Recursion on Notation by Bellantoni and Cook [3]. Consequently, we can expect to handle OO higher-order (as in Scala or Java 1.8). A combination with type discipline for studying complexity for HO functional program such as the ones based Light Linear Logics [2] would then be required.

Related works. This main idea of combining tiering and non-interference does not appear *ex nihilo*: safe recursion was already a non-interference based result (though the connection was not easy to see in the nineties as both domains were emerging research topics). Two decades later Marion has presented a type system [15] using this idea in order to capture polynomial time computable functions on an imperative language. This idea was adapted to a graph based language in [14]. The current paper tries to pursue this objective but on a distinct paradigm: Object. Thus our results strictly extend the ones of [15] while they are applied on a more concrete language than the ones in [14].

The works [1, 12, 5] are based on the analysis of heap-space and time consumption of Java bytecode. The results from [1, 12] 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 [5] and focuses on certifying memory bounds for Java Card. Current analysis can be seen as a complementary approach that is more expressive on the purely OO fragment as it handles while loops guarded by a variable of reference type whereas most of the aforementioned studies are based on invariants generation for primitive types only.

In a similar vein, characterizing complexity classes below polynomial time is studied in [11], which relies on a programming language, PURPLE, combining imperative statements together with pointers on a fixed graph structure. Although not directly related, our type system was strongly inspired by this work.

Outline. In a first section, we present the syntax and semantics of an abstract generic OO language. We also define the notion of input, input size and computation of a program in this language. In Section 2, we introduce a type system based on tiering techniques for controlling program complexity. This is the main contribution of the paper. In Section 3, we define a *Safety* condition based on tiers in order to restrict the allowed forms of recursion in a typed program. In Section 5, we show that safe and terminating programs characterize the class of functions computable in polynomial time. For the soundness, Theorem 1 shows that such a program terminates in time (number of executed base instructions) polynomial in the input size. For the completeness, Theorem 2 shows that any Turing Machine running in polynomial time can be simulated by a safe and terminating program. Section 4 is devoted to examples illustrating the methodology. Finally, we provide extensions and a conclusion in the last section.

1 Object Oriented Programs

In this section we define the syntax and semantics of an abstract OO programming language. We claim that this language is generic enough so that the complexity analysis performed by the type system presented in the next section can be adapted to most of the well-known OO programs. We also provide a notion of input and a notion of size of the heap and the stack in order to be able to discuss the complexity of such programs.

1.1 Abstract syntax

Expressions, instructions, methods and classes are defined by the following grammar:

$$\begin{aligned}
\text{Expressions} \ni \mathtt{e} ::= {} & \mathtt{x} \mid \mathtt{n} \mid \mathtt{null} \mid \mathtt{this} \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{op}(\overline{\mathtt{e}}) \\
& \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{e.clone}() \\
\text{Instructions} \ni \mathtt{I} ::= {} & ; \mid [\tau]\ \mathtt{x{:=}e;} \mid \mathtt{x{+}{+};} \mid \mathtt{x{-}{-};} \mid \mathtt{I_1\ I_2} \mid \mathtt{while(e)\{I\}} \\
& \mid \mathtt{if(e)\{I_1\}else\{I_2\}} \mid \mathtt{e.m}(\overline{\mathtt{e}}); \mid \mathtt{break;} \\
\text{Methods} \ni \mathtt{m_C} ::= {} & \tau\ \mathtt{m}(\overline{\tau\ \mathtt{x}})\{\mathtt{I[return\ x;]}\} \\
\text{Classes} \ni \mathtt{C} ::= {} & \mathtt{C}\ [\mathtt{extends\ D}]\ \{\overline{\tau\ \mathtt{x;}}\ \mathtt{C}(\overline{\tau\ \mathtt{y}})\{\overline{\mathtt{x{:=}y}}\}\ \overline{\mathtt{m_C}}\}
\end{aligned}$$

with $\mathtt{n} \in \mathbb{N}$, the set of integers, $\mathtt{x} \in \mathbb{V}$, the set of variables, $\mathtt{op} \in \mathbb{O}$, the set of operators, $\mathtt{C} \in \mathbb{C}$, the set of class names and $\mathtt{m} \in \mathbb{M}$, the set of method names. $[e]$ denotes some optional syntactic element e and $\overline{e}$ denotes a sequence of syntactic elements $e_1, \ldots, e_n$. The τs are type annotations ranging over $\mathbb{C} \cup \{\mathtt{void}, \mathtt{boolean}, \mathtt{int}\}$. Each operator comes equipped with a signature of the shape $\mathtt{op} :: \tau_1 \times \cdots \times \tau_\mathtt{n} \to \mathtt{boolean}$. For simplicity, the signature is restricted to operators with boolean outputs. It can be extended to general operators but in this case a restricted typing discipline as in [10] is required for operators as their computations might increase the data size. The abstract syntax does not

include a `for` instruction based on the premise that, as in Java, a for statement can be simulated by a while statement. Let $\mathtt{C}.\mathcal{F} = \{\overline{\mathtt{x}}\}$ to be the set of fields in a class $\mathtt{C}\ \{\overline{\tau\ \mathtt{x};}\ \mathtt{C}(\overline{\tau\ \mathtt{y}})\{\overline{\mathtt{x}:=\mathtt{y}}\}\ \overline{\mathtt{m_C}}\}$ and $\mathcal{F} = \cup_{\mathtt{C}\in\mathbb{C}}\mathtt{C}.\mathcal{F} \subseteq \mathbb{V}$ be the set of all fields. In a method or constructor, the arguments are called parameters. Each variable declared in an assignment of the shape $\tau\ \mathtt{x}:=\mathtt{e};$ is called a local variable. Given a method $\tau\ \mathtt{m}(\tau_1\ \mathtt{x}_1, \ldots, \tau_n\ \mathtt{x}_n)\{\mathtt{I}\ [\mathtt{return}\ \mathtt{x};]\}$ of `C`, its signature is $\tau\ \mathtt{m}^{\mathtt{C}}(\tau_1, \ldots, \tau_n)$, the notation $\mathtt{m}^{\mathtt{C}}$ denoting that `m` is declared in `C`. Also notice that there is no field access in our syntax using the "." operator. Consequently, all fields are implicitly `private`. In contrast, methods and classes are all `public`. This is not a huge restriction for an OO programmer since any field can be accessed and updated in an outer class by writing the corresponding getter and setter. In the particular case where `C extends D`, $\mathtt{C}(\overline{\tau\ \mathtt{y}})\{\overline{\mathtt{x}:=\mathtt{y}}\}$ is a constructor initializing both the fields of `C` and the fields of `D`. Inheritance defines a partial order on classes denoted by $\mathtt{C} \trianglelefteq \mathtt{D}$. For readability, classes are assumed to have exactly one constructor initializing all the class fields. The only considered primitive data are boolean values `true` and `false` and integer constants. Other primitive data types such as floats, doubles and characters could be considered and typed as integer values are by the type system provided in Section 2. Notice that overload and override are both allowed by our Syntax.

OO programs. A *program* is a collection of classes together with exactly one executable $\mathtt{Exe}\{\mathtt{void\ main()}\{\mathtt{Init\ Comp}\}\}$ with $\mathtt{Init}, \mathtt{Comp} \in$ Instructions. 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 OO 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.

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. There are two particular cases:

- In the particular case where the initialization instruction is empty, 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.
- In the particular case where the computational instruction is empty (that is ";") then the program will trivially pass the complexity analysis.

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

(i) For each class name `C`, there is exactly one class within the collection. Multiple inheritance and inner classes are prohibited.
(ii) A variable appearing in the collection of classes is either a local variable, a field or a parameter. In order to prevent name clashes, programs are assumed to be statically transformed up-to α-conversion. Each local variable `x` is both

declared and initialized exactly once by a τ `x` := `e`; instruction before its first use.

(iii) Each method signature is unique. A method output type is `void` iff the method has no `return` statement.

1.2 Informal semantics

In this section, we provide an informal semantics of OO programs and introduce data structures for representing the heap and the stack.

The *heap* $\mathcal{H}$ is represented by a a directed multigraph (V, A). The nodes in V are references labeled by class names and the arrows in A are labeled by field names. Given a heap $\mathcal{H} = (V, A)$, a *stack frame* $s_{\mathcal{H}} = \langle s, p\rangle$ is a pair composed by a method signature s and a partial mapping $p : \mathbb{V} \cup \{\texttt{this}\} \mapsto V \cup \mathbb{N} \cup \{\texttt{true}, \texttt{false}\}$ associating, either a reference in V to some variable in $\mathbb{V}$ of reference type in $\mathbb{C}$ or to the current object `this`, or a primitive value in $\mathbb{N}$ (resp. $\{\texttt{true}, \texttt{false}\}$) to some variable of primitive type `int` (resp. `boolean`). Let $dom(p)$ to be domain of p. By abuse of notation, given an expression `e` of reference type, let $p(\texttt{e})$ be the reference of the object corresponding to `e`.

The *stack* $\mathcal{S}_{\mathcal{H}}$ is a LIFO structure of stack frames corresponding to the same heap $\mathcal{H}$.

The mappings of the stack frames map method's parameters, current object and local variables to the references of the arguments on which they are applied.

A *memory configuration* $\mathcal{C}$ is a pair $\langle \mathcal{H}, \mathcal{S}_{\mathcal{H}}\rangle$ consisting in a heap $\mathcal{H}$ and a stack $\mathcal{S}_{\mathcal{H}}$. The *initial configuration* $\mathcal{C}_0$ is a configuration whose heap only consists in the null reference node `null` and whose stack only contains 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 (`true` for `boolean` and 0 for `int`). The evaluation of a `new` operator consists in adding a new node to the heap with arrows pointing to its fields; thus implementing the dynamic binding principle. Calling a method $\texttt{e.m}(\overline{\texttt{e}})$ of the class `C` and shape $\tau\ \texttt{m}(\overline{\tau\ \texttt{x}})\{\ldots\}$ consists in pushing a new stack frame $\langle \tau\ \texttt{m}^{\texttt{C}}(\overline{\tau}), p\rangle$ on the stack with p such that $p(\texttt{this}) = p(\texttt{e})$ and $p(\texttt{x}_i) = p(\texttt{e}_i)$. A call to `e.clone()` consists in duplicating the subgraph of source $p(\texttt{e})$ in $\mathcal{H}$. For simplicity, we assume `clone` to be evaluated in constant time. Though it is a deep-copy, this assumption is reasonable as it makes the analysis easier at a small cost: the polynomial degree of Theorem 1 just differs by one constant as this method is usually evaluated in linear time.

Example 1. Consider the code in Figure 1. At line 1, the program starts on the initial configuration $\mathcal{C}_0$. After executing line 4, it ends in a configuration $\mathcal{C} = \langle \mathcal{H}, \mathcal{S}_{\mathcal{H}}\rangle$ with a heap $\mathcal{H} = (V, A)$ represented by nodes and arrows and with a stack $\mathcal{S}$ consisting in only one stack frame $\langle \texttt{void main}^{\texttt{Exe}}(), p\rangle$; the mapping p being represented by boxed nodes and snake arrows.

```
main(){
1     B b := new B(null, new A());
2     C c := new C(b);
3     D d := new D(c);
4     B e := new B(c, c);
}
```

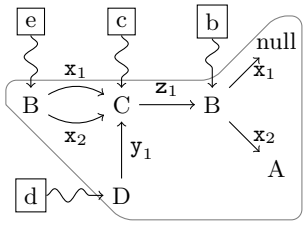


Fig. 1. Example of a pointer graph

1.3 Input and size

Given a program of executable `Exe{void main(){Init Comp}}`, the *input* is the memory configuration $\mathcal{C}$ obtained after executing the initialization instruction `Init` on the initial memory configuration $\mathcal{C}_0$. Consequently, `Init` is assumed to be a terminating instruction.

Definition 1 (Sizes). *The size $|\mathcal{H}|$ of a heap $\mathcal{H} = (V, A)$ is defined to be the number of nodes in V. The size of a mapping p is defined by $|p| = \sum_{\mathtt{x} \in dom(p)} |p(\mathtt{x})|$ where the size of a boolean value is 1, the size of an integer value is the value itself and the size of a memory reference is 1. The size of a stack frame $s_{\mathcal{H}} = \langle s, p \rangle$ is defined by: $|s_{\mathcal{H}}| = 1 + |p|$. The size of a stack $\mathcal{S}_{\mathcal{H}}$ is defined by $|\mathcal{S}_{\mathcal{H}}| = \sum_{s_{\mathcal{H}} \in \mathcal{S}_{\mathcal{H}}} |s_{\mathcal{H}}|$. Finally, the size of a memory configuration $\mathcal{C} = \langle \mathcal{H}, \mathcal{S}_{\mathcal{H}} \rangle$ is defined by $|\mathcal{C}| = |\mathcal{H}| + |\mathcal{S}_{\mathcal{H}}|$.*

The above definition is robust if we consider boolean values to be 8 (bits) values, integer values to be 32 (bits) values, ... Indeed, in such a case integers will be considered as constants so that the upper bounds presented in Theorem 1 remain valid. Notice that the out-degree of a node is bounded by a constant of the program (the maximum number of fields in a class) and, consequently, bounding the number of nodes is sufficient to obtain a big O bound on the heap size. 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).

2 Type system

The main contribution of the paper, a tier based type system for ensuring polynomial time and polynomial space upper bounds on the size of a memory configuration, is introduced in this section. We first define the notion of tiered types inspired by tiering on function algebra and we define the notions of typing environments and judgments. Then we present and explain the type system rules and the notion of well-typedness. Finally, we exhibit the main properties of a well-typed program.

2.1 Tiered types.

A *tiered type* is a pair $\tau(\alpha)$ consisting of a type $\tau \in \{\texttt{void}, \texttt{boolean}, \texttt{int}\} \cup \mathbb{C}$ together with a tier $\alpha \in \{\mathbf{0}, \mathbf{1}\}$. Given a tiered type, the two projections π_1 and π_2 are defined by $\pi_1(\tau(\alpha)) = \tau$ and $\pi_2(\tau(\alpha)) = \alpha$. The order $\preceq$ on tiers is such that $\mathbf{0} \preceq \mathbf{1}$. Let $\wedge$ and $\vee$ be the induced min and max operators on set of tiers and let $\alpha, \beta, \ldots$ denote tier variables.

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).
Intuition: Tiers will be used to separate data in two kinds as in Bellantoni and Cook's safe recursion scheme [3] where data are divided into "safe" and "normal" data kinds. Referring to Danner and Royer [7], "normal data [are the data] that drive recursions and safe data [are the data] over which recursions compute". In our 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 equivalent for safe data type, as it consists in computational data storages.

2.2 Typing environments and judgments.

For a given program, a *method typing environment* δ maps each variable $v \in \mathbb{V}$ to a tiered type. For a given program, a *typing environment* Δ maps each method signature $\tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau})$ to a method typing environment δ, i.e. $\Delta(\tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau})) = \delta$.

A *contextual typing environment* $\Gamma = (s, \Delta)$ is a pair consisting of a method signature and a typing environment. The method signature s in the contextual typing environment (s, Δ) indicates under which context the fields should be typed. $\forall \mathtt{x} \in \mathbb{V}$, 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 $\Gamma'(\mathtt{x}) = \tau(\alpha)$.
Intuition: The main reason for defining typing environments this way is to allow the programmer to type a field with distinct tiers depending on the considered method. This is the reason why the presented type system has to keep information on the context.

Given a contextual typing environment Γ, there are three kinds of typing judgments:

- $\Gamma \vdash \mathtt{e} : \tau(\alpha)$ for expressions, meaning that the expression $\mathtt{e}$ is of tier type $\tau(\alpha)$ under the environment Γ,
- $\Gamma \vdash \mathtt{I} : \mathtt{void}(\alpha)$ for instructions, meaning that the instruction $\mathtt{I}$ is of tier type $\mathtt{void}(\alpha)$ under the environment Γ,
- $\Gamma \vdash s : \mathtt{C}(\beta) \times \langle\overline{\tau}(\overline{\alpha})\rangle \to \tau(\alpha)$ 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 $\mathtt{this}$), has parameters of type $\langle\overline{\tau}(\overline{\alpha})\rangle$ and a return variable of type $\tau(\alpha)$, with $\tau = \mathtt{void}$ in the particular case where there is no return statement.

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 a sequence of tiers $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$, the notation $\Gamma \vdash \overline{\mathtt{e}} : \overline{\tau}(\overline{\alpha})$ means that $\Gamma \vdash \mathtt{e}_i : \tau_i(\alpha_i)$ holds, for all $i \in [1, n]$.

$$\frac{\mathtt{n} \in \mathbb{N}}{\Gamma \vdash \mathtt{n} : \mathtt{int}(\alpha)}\ (Int) \qquad \frac{w \in \{\mathtt{true}, \mathtt{false}\}}{\Gamma \vdash w : \mathtt{boolean}(\alpha)}\ (Bool) \qquad \frac{}{\Gamma \vdash \mathtt{null} : \mathtt{C}(\alpha)}\ (Null)$$

$$\frac{\Gamma(\mathtt{x}) = \tau(\alpha)}{\Gamma \vdash \mathtt{x} : \tau(\alpha)}\ (Var) \qquad \frac{\forall i,\ \Gamma \vdash \overline{\mathtt{e}} : \overline{\tau}(\alpha) \qquad \mathtt{op} :: \langle \overline{\tau} \rangle \to \mathtt{boolean}}{\Gamma \vdash \mathtt{op}(\overline{\mathtt{e}}) : \mathtt{boolean}(\alpha)}\ (Op)$$

$$\frac{\forall \mathtt{x} \in \mathtt{C}.\mathcal{F}, \pi_2(\Gamma(\mathtt{x})) = \alpha}{\Gamma \vdash \mathtt{this} : \mathtt{C}(\alpha)}\ (Self) \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{D}(\alpha) \quad \mathtt{D} \trianglelefteq \mathtt{C}}{\Gamma \vdash \mathtt{e} : \mathtt{C}(\alpha)}\ (Pol)$$

$$\frac{\Gamma \vdash \overline{\mathtt{e}} : \overline{\tau}(\mathbf{0}) \quad \mathtt{C}.\mathcal{F} = \{\overline{\mathtt{x}}\} \quad \Gamma \vdash \overline{\mathtt{x}} : \overline{\tau}(\mathbf{0})}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}(\mathbf{0})}\ (New) \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C}(\mathbf{1})}{\Gamma \vdash \mathtt{e.clone}() : \mathtt{C}(\mathbf{0})}\ (Cln)$$

$$\frac{(s, \Delta) \vdash \mathtt{e} : \mathtt{C}(\beta) \quad (s, \Delta) \vdash \overline{\mathtt{e}} : \overline{\tau}(\overline{\alpha}) \quad (\tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau}), \Delta) \vdash \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau}) : \mathtt{C}(\beta) \times \langle \overline{\tau}(\overline{\alpha}) \rangle \to \tau(\alpha)}{(s, \Delta) \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \tau(\alpha)}\ (Call)$$

Fig. 2. Type system for expressions

$$\frac{}{\Gamma \vdash ; : \mathtt{void}(\alpha)}\ (Skip) \qquad \frac{\Gamma \vdash \mathtt{I} : \mathtt{void}(\mathbf{0})}{\Gamma \vdash \mathtt{I} : \mathtt{void}(\mathbf{1})}\ (ISub) \qquad \frac{\mathtt{x} \notin \mathcal{F} \quad \Gamma \vdash \mathtt{x} : \tau(\alpha) \quad \Gamma \vdash \mathtt{e} : \tau(\alpha)}{\Gamma \vdash [\tau]\ \mathtt{x}{:=}\mathtt{e}; : \mathtt{void}(\alpha)}\ (VA)$$

$$\frac{\mathtt{x} \in \mathcal{F} \quad \Gamma \vdash \mathtt{x} : \tau(\mathbf{0}) \quad \Gamma \vdash \mathtt{e} : \tau(\mathbf{0})}{\Gamma \vdash \mathtt{x}{:=}\mathtt{e}; : \mathtt{void}(\mathbf{0})}\ (FA) \qquad \frac{\Gamma \vdash \mathtt{x} : \mathtt{int}(\mathbf{0})}{\Gamma \vdash \mathtt{x{+}{+};} : \mathtt{void}(\mathbf{0})}\ (Inc) \qquad \frac{\Gamma \vdash \mathtt{x} : \mathtt{int}(\alpha)}{\Gamma \vdash \mathtt{x{-}{-};} : \mathtt{void}(\alpha)}\ (Dec)$$

$$\frac{}{\Gamma \vdash \mathtt{break}; : \mathtt{void}(\mathbf{1})}\ (Brk) \qquad \frac{\forall i,\ \Gamma \vdash \mathtt{I}_i : \mathtt{void}(\alpha_i)}{\Gamma \vdash \mathtt{I}_1\ \mathtt{I}_2 : \mathtt{void}(\alpha_1 \vee \alpha_2)}\ (Seq)$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{boolean}(\alpha) \quad \forall i,\ \Gamma \vdash \mathtt{I}_i : \mathtt{void}(\alpha)}{\Gamma \vdash \mathtt{if(e)\{I_1\}else\{I_2\}} : \mathtt{void}(\alpha)}\ (If) \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{boolean}(\mathbf{1}) \quad \Gamma \vdash \mathtt{I} : \mathtt{void}(\mathbf{1})}{\Gamma \vdash \mathtt{while(e)\{I\}} : \mathtt{void}(\mathbf{1})}\ (Wh)$$

Fig. 3. Type system for instructions

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

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

Fig. 4. Type system for methods

2.3 Typing rules

The typing rules for expressions, instructions and methods are provided in Figures 2, 3 and 4, respectively.

The intuition 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. These rules can be divided into two categories:

- The unconstrained rules *(Int)*, *(Bool)*, *(Null)*, *(Var)*, *(Op)*, *(Self)*, *(Pol)*, *(VA)*, *(Skip)*, *(Dec)*, *(If)*, for which the tier is not constrained by the rule. They are fairly standard and only a few of them need some deeper explanations:
  - Primitive constants and the null reference can be given any tier as they have no computational power (Rules *(Int)*, *(Bool)* and *(Null)*). As in Java and for polymorphic reasons, `null` can be considered of any class `C`.
  - 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 field of the current object, thus preventing "flows by references" in the heap.
  - The rule *(VA)* is the main non-interference rule. It forbids information flows from a tier to another: it is only possible to assign an expression `e` of tier α to a variable `x` of tier α.
  - The rule *(If)* constrains the tier of the conditional guard `e` to match the tiers of the branching instructions $\mathtt{I}_1$ and $\mathtt{I}_2$. Hence, it prevents assignments of tier **1** variables to be controlled by a tier **0** expression.
- The constrained rules *(New)*, *(FA)*, *(Inc)*, *(Brk)* and *(Wh)* for which the tier is fixed by the rule, for some precise "complexity" purpose:
  - 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.
  - In the case of a field assignment (Rule *(FA)*), all tiers are constrained to be **0** in order to avoid changes inside the tier **1** graph.
  - In rule *(Inc)*, the tier is constrained to be **0** as the integer value increases.
  - Rule *(Brk)* enforces the tier of a break instruction to be **1**. This prevents the programmer from writing conditionals of the shape:

$$\mathtt{while}(\underbrace{\mathtt{x}}_{\mathbf{1}})\{\mathtt{I}_1\ \mathtt{if}(\underbrace{\mathtt{y}}_{\mathbf{0}})\{\underbrace{\mathtt{break;}}_{\mathbf{0}}\}\ \mathtt{else}\{\mathtt{I}_2\}\ \mathtt{I}_3\}$$

    that would break the non-interference property of tiers (see Rule *(If)* above). Indeed, in the above example the number of iterations in the while loop might depend on the value of the tier **0** variable `y`.

- 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.

Now there only remain some particular rules to discuss:

- The 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 wrt 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++;` 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.
- Consequently, only a restricted form of subtyping is allowed for expressions. This is the purpose of Rule *(Cln)* allowing the programmer to declassify information from a tier **1** expression to a **0** expression through the use of the `clone` method. Consequently, the tier **0** modifications on the copy will not affect the original tier **1** object. Notice that the choice to include the `clone` method as a primitive construct of the language has been made to make this subtyping explicit. An alternative would have been to program this method as usual in any class `C` and to check in a straightforward manner that the following judgment can be derived $\Gamma \vdash \mathtt{C\ clone}() : \mathtt{C}(\mathbf{1}) \to \mathtt{C}(\mathbf{0})$.
- Methods typing and polymorphism is handled by rules *(Call)*, *(Body)* and *(OverR)*. Rule *(Call)* 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 wrt to the considered method. Rule *(Body)* shows how to type method definitions. It updates the environment wrt 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. Rule *(OverR)* deals with overridden method, keeping tiers preserved, thus allowing standard OO polymorphism.

2.4 Well-typedness.

Given a program of executable $\mathtt{Exe}\{\mathtt{main}()\{\mathtt{Init\ Comp}\}\}$ and a typing environment Δ, the judgment $\Delta \vdash \mathtt{Exe} : \diamond$ means that the program is well-typed wrt Δ and is defined by:

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

where $\vDash$ is a judgment derived from the type system by removing all tiers and tier based constraints in the typing rules. 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.

2.5 Type system non-interference properties

Now we sum up some of the most crucial properties of the presented type system:

Property 1. There is no information flows from tier **0** data to tier **1**. The only flows from tier **1** data to tier **0** is through cloning.

This is due to the non-interference nature of the type system (see Volpano et al. [16] for more details). The only change imposed by the OO paradigm being that the current object has the same tier as its fields (rule *(Self)* of Figure 2). By looking carefully at the rules of Figure 2, we can check that Rule *(Cln)* is the only rule allowing flows from **1** to **0**. It does not break the property as the data flow is on a freshly cloned part of the heap.

Property 2. Tier **1** data cannot be altered.

Object creation is restricted to tier **0** by rule *(New)* of Figure 2. Moreover, tier **1** references cannot change as field assignments are restricted to tier **0** by rule *(FA)* of Figure 2.

Property 3. Given a program with no recursive method, execution time does not depend on tier **0** data.

This is straightforward if we do not consider recursive methods as while loop guards are restricted to be of tier **1** by rule *(Wh)* of Figure 3. The next section will be devoted to putting a restriction on recursive methods in order to extend this property. Carefully notice that it does not prevent a tier **0** variable to appear in the guard of a while loop but still the control flow will not depend on it.

3 Safe recursion

In this section, a safety criterion is provided in order to allow the programmer to use an admissible but restricted form of recursion. Indeed, Property 3 is valid under the hypothesis that there is no recursive call. At the present time, a recursive method might make a recursive call to be controlled by tier **0** data. This is a highly unwanted behavior. Moreover, even assuming such a Property to be satisfied, one would still be able to program a multiply recursive method with an exponential number of recursive calls. The safety criterion eliminates these two issues.

3.1 Level and intricacy

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 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 $s' \sqsubset^+ s$.

We introduce the level and intricacy of instructions and extend them to programs. The level bounds the number of recursive calls while the intricacy corresponds to the number of nested while loops.

Definition 2 (Level). *The level λ of a method signature is defined by:*

- $\lambda(s) = 0$ *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 λ be the maximal level of a method in a given program.

Definition 3 (Intricacy). *The intricacy ν of an instruction is defined by:*

- $\nu([[\tau]\ \mathtt{x{:=}e}];) = 0$
- $\nu(\mathtt{x{+}{+}};) = 0$
- $\nu(\mathtt{x{-}{-}};) = 0$
- $\nu(\mathtt{break};) = 0$
- $\nu(\mathtt{I_1\ I_2}) = \max(\nu(\mathtt{I_1}), \nu(\mathtt{I_2}))$
- $\nu(\mathtt{if(x)\{I_1\}else\{I_2\}}) = \max(\nu(\mathtt{I_1}), \nu(\mathtt{I_2}))$
- $\nu(\mathtt{while(x)\{I\}}) = 1 + \nu(\mathtt{I})$

Let ν be the maximal intricacy of an instruction within a given program.

Both intricacy and level are bounded by the size of their program.

3.2 Safety restriction.

Now some side restrictions on recursive methods are provided to ensure that their flow is only controlled by tier **1** variables and to prevent exponential growth rate.

Definition 4 (Safety). *A well-typed program wrt a typing environment Δ is* safe *if for each recursive method* $\tau\ \mathtt{m}(\overline{\tau\ \mathtt{x}})\{\mathtt{I}\ [\mathtt{return\ x};]\}$*:*

1. *there is at most one call to some* $\mathtt{m}' \in [\mathtt{m}]$ *in the evaluation of* $\mathtt{I}$*,*
2. *there is no while loop inside* $\mathtt{I}$*, i.e.* $\nu(\mathtt{I}) = 0$*,*
3. *and* $(s, \Delta) \vdash \tau\ \mathtt{m}^{\mathtt{C}}(\overline{\tau\ \mathtt{x}}) : \mathtt{C}(\mathbf{1}) \times \langle\overline{\tau}(\mathbf{1})\rangle \to \tau(\alpha)$ *can be derived and the call to* $\mathtt{m}$ *is a* **1** *instruction.*

Remark 1. Notice that safety is a generalization of the safe recursion on notation (SRN) scheme by Bellantoni and Cook [3]. Indeed a function SRN function can be defined (and typed) in our setting:

```
f(int x, τ y){
        int res := 0 ;
        if(x == 0){res := g(y;)}
        else{if(x%2 == i){res := h_i(f(x/2,y)); }}
        return res;
}
```

If `f` output is of tier $\mathbf{0}$ (i.e. computes something) then $\mathtt{h}_i$ will not be able to recurse on it. Clearly, the above program fullfills the above Definition for some typing context Γ such that $\Gamma(\mathtt{x}) = \Gamma(\mathtt{y}) = \mathtt{int}(\mathbf{1})$.

4 Boolean lists as an illustrating example

Consider the class `BList` encoding integers as linked lists of bits and including a field `value` of type `boolean` and a field `tail` of type `BList`. Suppose that this class comes with a getter and a setter:

```
BList getTail(){return tail;}
void setTail(BList q){tail := q;}
```

They can be typed by:

- $\Gamma \vdash \mathtt{BList\ getTail()} : \mathtt{BList}(\alpha) \to \mathtt{Blist}(\alpha)$, $\alpha \in \{\mathbf{0}, \mathbf{1}\}$, by Rules *(Body)* and *(Self)*. The tier of the current object has to match the tier of the tail because of Rule *(Self)*. However it still can be $\mathbf{0}$ or $\mathbf{1}$ depending on the object on which the method is called. Recall that methods are polymorphic in our typing discipline and two distinct calls can be sometimes given distinct tiers.
- $\Gamma \vdash \mathtt{void\ setTail(BList\ q)} : \mathtt{BList}(\mathbf{0}) \times \mathtt{Blist}(\mathbf{0}) \to \mathtt{void}(\mathbf{0})$ by Rules *(Body)*, *(Self)* and *(FA)*. Here the tiers of the field and the parameter are forced to be $\mathbf{0}$ by Rule *(FA)* and, consequently, the tier of the current object is enforced to be the same by Rule *(Body)*.

We can then type the methods `concat` and `length` below. We write $\mathtt{e}^{\alpha}$ (resp. $\mathtt{I} : \alpha$) to denote that `e` (resp. `I`) is of tier α w.r.t. the environment Γ.

```
void concat(BList other) {
    BList o^1 := this^1;
    BList t^0 := this.clone()^0;
    while (o.getTail()^1 != null) {
       o^1 := o.getTail()^1;
       t^0 := t.getTail()^0;
    }
    t.setTail(other^0);
}
```

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

$\Gamma \vdash$ `void concat(BList other)` : $\texttt{BList}(\mathbf{1}) \times \texttt{BList}(\mathbf{0}) \to \texttt{void}(\mathbf{1})$.
$\Gamma \vdash$ `int length()` : $\texttt{BList}(\mathbf{1}) \to \texttt{int}(\mathbf{0})$.

The recursive method `length` satisfies conditions 1 and 2 of Definition 4. It can be typed by $\texttt{BList}(\mathbf{1}) \to \texttt{int}(\mathbf{0})$ and it is called in a tier **1** instruction. Consequently, the program is safe. Moreover, the program obtained is clearly terminating. Its intricacy ν is equal to 0, since there is no nested while loops in its methods, and its level λ is equal to 1, since there is one level of recursion in the method `length`. Consequently, it terminates in time $O(|\mathcal{C}|)$ on input $\mathcal{C}$ by Theorem 1 (see the next section).

5 Characterization of polynomial time

In this section, we show the main result of our paper: a characterization of the class of functions computable in polynomial time by a Turing Machine, known as *FPtime*, with respect to safe and terminating OO programs. We first show the soundness by providing a polynomial upper bound on the time of such kind of programs. Then we prove the completeness by simulating polynomial time Turing Machines by safe and terminating programs. We conclude by showing that type inference can performed in polynomial time.

5.1 Polynomial Time Soundness

Let $n_{\mathbf{1}}$ be the number of tier **1** variables in the whole program wrt the typing environment under consideration.

Theorem 1 (Soundness). *If a program is safe and terminates on input $\mathcal{C}$ then it does terminate in time $O(|\mathcal{C}|^{n_{\mathbf{1}}(\nu+\lambda)})$.*

Proof. By Property 1 there is no information flow from tier **0** to tier **1**. Control flow in while loops and recursive calls only depends on tier **1** variables by Property 3 and by definition of safety. Moreover tier **1** variables cannot point out of the initial pointer subgraph by Property 2. Consequently, if the program terminates such a variable has at most $|\mathcal{C}|$ possible distinct values during the program execution. Otherwise it contradicts the termination assumption. Indeed if the same program pointer instruction is encountered during the execution of one program with the same tier **1** values in the heap then an infinite loop is reached as programs are deterministic. Consequently, there can be only $|\mathcal{C}|^{n_{\mathbf{1}}}$

distinct configurations restricted to tier **1** variable (by restricted we mean that two configurations only differing on tier **0** variables are supposed to be equal). Finally, the level λ, intricacy ν and constant $n_{\mathbf{1}}$ are used to compute the global upper bound. Indeed, the unfolding of while and recursive calls can generate a complexity in $O((|\mathcal{C}|^{n_1})^{(\nu+\lambda)})$.

5.2 Polynomial time completeness.

We start to show that any polynomial can be computed by a safe and terminating program. By abuse of notation, we will use the notation $\Delta(\mathtt{m}^{\mathtt{C}})$ in all the examples when the method signature is clear from the context. Consider the following method of some class `C` computing addition:

```
add(int x, int y){
  while(x¹>0){
    x¹--; :1
    y⁰++; :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 **1**, by typing rules *(Wh)* and *(Op)* as it appears in the guard of a while loop (the operator $>$ keeping the tier unchanged in rule *(Op)*). Moreover `y` is enforced to be of tier **0**, by typing rule *(Inc)*.

Consider the below method encoding multiplication:

```
mult(int x, int y){
  int z⁰ := 0;
  while(x¹>0){
    x¹--;
    int u¹ := y¹;
    while(u¹>0){
      u¹--;
      z⁰++;
    }
  }
  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 typing rules *(Wh)* and *(Op)*. Moreover `y` is enforced to be of tier **1**, by typing rule *(VA)* applied to instruction `int u=y;`, `u` being of tier **1**. Finally, `z` is enforced to be of tier **0**, by typing rule *(Inc)*, as its stored value increases in `z++;`.

Consequently, any polynomial can be computed by using a composition of the two above methods.

Theorem 2 (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 $\mathbf{0}$. A one step transition is simulated by a finite cascade of if-commands of the form:

```
if(y.getHead()^0){
        if(state^0 == 8^0){
                state^0 = 3^0;:0
                x^0 =new BList(false,x^0);:0
                y^0 = 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 (see the Example of Section 4 for more details) $\texttt{BList}(\mathbf{0}) \to \texttt{BList}(\mathbf{0})$ and $\texttt{BList}(\mathbf{0}) \to \texttt{boolean}(\mathbf{0})$, respectively. Since each variable inside the above command is of tier $\mathbf{0}$, the tier of the if-command is also $\mathbf{0}$. 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.

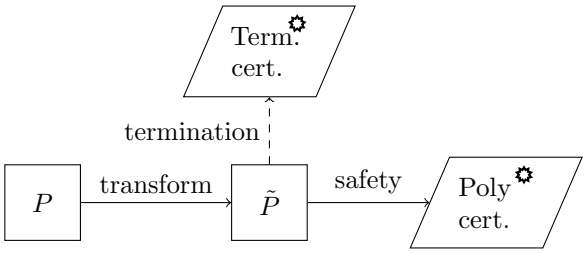
5.3 Decidability of type inference.

Theorem 3 (Decidability of type inference). *Deciding if there exists a typing environment s.t. a program is well-typed can be done in polynomial time in the size of the program.*

Proof. The type inference problem can, be reduced to 2-SAT. Notice that the number of distinct tiered types that can be given to a method is bounded by the number of calls in the program and thus bounded by the program size. Indeed the type checking is a static analysis performed on the code. Consequently, this problem can be solved in linear time. Types can be checked in linear time in the size of the program as typing mainly consists in checking type annotations with respect to method signatures, operator signatures and attributes declarations. We encode the tier of each attribute `x` within the method `m` of class `C` by a boolean variable $x^{\mathtt{m}^{\mathtt{c}}}$ that will be true if the variable is of tier $\mathbf{1}$, false if it is of tier $\mathbf{0}$ in the context of $\mathtt{m}^{\mathtt{C}}$. All local variables and parameters can be encoded by a single variable as their tier is independent from the context. Each instruction generates

some constraints. For example, in the case of an assignment $\mathtt{x} := \mathtt{y}$; in the context $\mathtt{m}^{\mathtt{C}}$, we have to check $\pi_2(\Delta(\mathtt{m}^{\mathtt{C}})(\mathtt{x})) \preceq \pi_2(\Delta(\mathtt{m}^{\mathtt{C}})(\mathtt{y})))$, which can be represented as $(y^{\mathtt{m}^{\mathtt{C}}} \vee \neg x^{\mathtt{m}^{\mathtt{C}}})$. All these constraints generate a conjunction of such clauses which are in number linear in the size of the program. As a result, the type inference problem is reduced to 2-SAT and can be solved in polynomial time.

6 Methodology of the presented analysis



The OO program complexity analysis presented in this paper can be summed up by the above 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 program $\tilde{P}$ of our abstract language. This transformation contains the following steps:

- convert syntactical constructs of the source language in P to constructs in the abstract OO language. In particular, transform `for` into `while`.
- for all public fields of P, write the corresponding getter and setter in $\tilde{P}$
- for each constructor in P, write a corresponding factory in $\tilde{P}$ (and just keep the basic constructor for object instantiation)
- α-rename the variables so that there is no name clashes in $\tilde{P}$.

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, we can perform in parallel a termination check and a safety check. The termination certificate can be obtained using existing tools such as [6, 8, 9, 4] (as the semantics is preserved, the check can also be performed on the original program P or on the compiled bytecode). If both succeed, Theorem 1 ensures polynomial time termination. In the safety check, a polynomial time type inference (Theorem 3) is performed together with a criterion check on recursive methods. This latter check is generally undecidable because of condition 1 of Definition 4. However it is very easy to restrict syntactically this condition so that the check becomes decidable (*e.g.* restrict a recursive call to appear at most once in a conditional branching).

7 Expressivity and open issues

The expressivity of the presented analysis is good. Most polynomial programs over inductive data types such as linked lists and trees are captured (in particular

BC's Safe recursion on notation). It also handles while loops guarded by objects and circular data structure. This is new.

Let us highlight that the expressivity of the method can be improved by a compositional type check: even if an instruction $\mathtt{I} = \mathtt{I}_1 \ \ldots \ \mathtt{I}_n$ fails the safety check, if each of the $\mathtt{I}_i$ passes the safety check then we can consider that $\mathtt{I}$ succeeds. Indeed, a bounded composition of polynomials remains polynomial.

We could also add operators with outputs distinct from booleans. In such a case, a restriction on the size of their computations is required (see [10] for more details).

A characterization of polynomial space based on the same methodology and OO threads is expected to be highly plausible.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theor. Comput. Sci. 413(1), 142–159 (2012)
2. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda-calculus. In: Logic in Computer Science, LICS 2004. pp. 266–275 (2004)
3. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the poly-time functions. Comput. Complex. 2, 97–110 (1992)
4. Ben-Amram, A.M., Genaim, S., Masud, A.N.: On the termination of integer loops. In: VMCAI. LNCS, vol. 7148, pp. 72–87 (2012)
5. Cachera, D., Jensen, T., Pichardie, D., Schneider, G.: Certified memory usage analysis. In: FM 2005: Formal Methods. LNCS, vol. 3582, pp. 91–106 (2005)
6. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Computer Aided Verification, CAV 2006. LNCS, vol. 4144, pp. 415–426 (2006)
7. Danner, N., Royer, J.S.: Ramified structural recursion and corecursion. CoRR abs/1201.4567 (2012), `http://arxiv.org/abs/1201.4567`
8. Gulwani, S.: Speed: Symbolic complexity bound analysis. In: Computer Aided Verification, CAV 2009. LNCS, vol. 5643, pp. 51–62 (2009)
9. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: precise and efficient static estimation of program computational complexity. In: POPL 2009. pp. 127–139. ACM (2009)
10. Hainry, E., Marion, J.Y., Péchoux, R.: Type-based complexity analysis for fork processes. In: FoSSaCS 2013. LNCS, vol. 7794, pp. 305–320 (2013)
11. Hofmann, M., Schöpp, U.: Pure pointer programs with iteration. ACM Trans. Comput. Log. 11(4) (2010)
12. Kersten, R., Shkaravska, O., van Gastel, B., Montenegro, M., van Eekelen, M.C.J.D.: Making resource analysis practical for real-time Java. In: JTRES. pp. 135–144 (2012)
13. Leivant, D., Marion, J.Y.: Lambda calculus characterizations of poly-time. Fundam. Inform. 19(1/2), 167–184 (1993)
14. Leivant, D., Marion, J.Y.: Evolving graph-structures and their implicit computational complexity. In: ICALP 2013. LNCS, vol. 7966, pp. 349–360 (2013)
15. Marion, J.Y.: A type system for complexity flow analysis. In: Logic in Computer Science, LICS 2011. pp. 123–132 (2011)
16. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Computer Security 4(2/3), 167–188 (1996)