Information and Computation 261 (2018) 78–115

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

A type-based complexity analysis of Object Oriented programs ☆

Emmanuel Hainry *, Romain Péchoux

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

ARTICLE INFO

Article history:
Received 2 July 2015

Keywords:
Object Oriented Program
Type system
Complexity
Polynomial time

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.

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.

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].

☆ This work has been partially supported by ANR Project ELICA ANR-14-CE25-0005.
* Corresponding author.
E-mail addresses: hainry@loria.fr (E. Hainry), pechoux@loria.fr (R. Péchoux).

https://doi.org/10.1016/j.ic.2018.05.006
0890-5401/

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: