Science of Computer Programming 95 (2014) 359–375

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Field-sensitive unreachability and non-cyclicity analysis

Enrico Scapin[a,*], Fausto Spoto[b]

[a] Department of Computer Science, University of Trier, Germany
[b] Department of Computer Science, University of Verona, Italy

HIGHLIGHTS

- We model a novel and sound data-flow analysis for Java bytecode.
- Our interprocedural definite analysis approximates two related heap properties.
- Abstract Interpretation is used to soundly approximate the program semantics.
- An abstract constraint graph of the program is built to compute the solution.
- The final aim is to improve other analyses by also considering program fields.

ARTICLE INFO

Article history:
Received 17 January 2014
Accepted 11 March 2014
Available online 8 April 2014

Keywords:
Data-flow analysis
Interprocedural static analysis
Constraint-based analysis
Field-sensitive analysis
Abstract interpretation

ABSTRACT

Field-sensitive static analyses of object-oriented code use approximations of the computational states where fields are taken into account, for better precision. This article presents a novel and sound *definite* analysis of Java bytecode that approximates two strictly related properties: field-sensitive unreachability between program variables and field-sensitive non-cyclicity of program variables. The latter exploits the former for better precision. We build a *data-flow* analysis based on *constraint graphs*, whose nodes are program points and whose arcs propagate information according to the semantics of each bytecode instruction. We follow *abstract interpretation* both to approximate the concrete semantics and to prove our results formally correct. Our analysis has been designed with the goal of improving client analyses such as *termination analysis*, asserting the non-cyclicity of variables with respect to specific fields.

1. Introduction

Static analysis builds compile-time approximations of the set of values, states or behaviors arising dynamically, at run-time *i.e.*, during the execution of a computer program. This is important to improve the quality of software by detecting illegal operations, such as divisions by zero or dereferences of `null`, erroneous executions, such as infinite loops, or security flaws, such as unwanted disclosure of information. In order to make static analysis computable, we follow *abstract interpretation* [1] here, a framework that lets one define approximated but sound static analyses from the formal specification of the properties of interest and of the semantics of the language.

In modern object-oriented languages such as Java, a typical problem related to the verification of real, large software programs is how the dynamic allocation of objects shapes the heap: namely, objects can be instantiated on demand and can reference other objects through *fields*, that can be updated at run-time. There are several articles in literature describing *memory*-related properties and providing *pointer analyses* that statically determine approximations of the possible run-time

* Corresponding author.
E-mail addresses: scapin@uni-trier.de (E. Scapin), fausto.spoto@univr.it (F. Spoto).

http://dx.doi.org/10.1016/j.scico.2014.03.012
0167-6423/© 2014 Elsevier B.V. All rights reserved.

values of a pointer [3]. *Shape analysis* [11] builds the possible shapes that data structures might assume at run-time; *aliasing analysis* [6] determines which variables point to the same location; *sharing analysis* [14] infers which variables are bound to overlapping data structures; *reachability analysis* [7] looks for paths between locations and *non-cyclicity analysis* [10] spots variables bound to non-cyclical data. In this context, we present here a *definite* data-flow analysis for field-sensitive unreachability and non-cyclicity. Namely, we build an *under-approximation* of the program fields that are never used in the paths between two variables or in a cycle bound to a variable, respectively. Under-approximations in the context of abstract interpretation have been studied in [13] through predicate transformers where the abstract transition function is a sound postcondition transformer of the state-transition function. A field-sensitive pointer analysis has been developed in [9], with a constraint-based approach as ours but not for object-oriented languages with dynamic memory allocation; instead, C and fields of structures are considered. Furthermore they extended a set constraint language and an inference system to model each field as a separate variable. Here instead, unreachability and non-cyclicity specify which fields cannot be used to establish the property. The work most related to ours is [2], that introduces an acyclicity analysis as the reduced product of abstract domains for reachability and cyclicity, over a semantics similar to ours. They highlight that cyclicity supports reachability *i.e.*, one can exploit unreachability information to improve non-cyclicity analysis. The main difference with our work is that we compute the fields not involved in reachability or cyclicity, getting higher precision. Furthermore, we have provided formal correctness proofs for the propagation rules of each bytecode instruction and method call, including its side-effects (see [12]).

Our analysis is designed with the goal of improving client analyses of the Julia analyzer for Java and Android bytecode (`http://www.juliasoft.com`). Namely, its *termination checker* finds method calls that might diverge at run-time, through the *path-length* property [16] *i.e.*, an estimation of the maximal length of a path of pointers rooted at each given program variable. For the Java instruction $\mathsf{x} = \mathsf{x.next}$, Julia estimates the path-length of x; in the original definition, it is decreasing only if it is possible to assert the non-cyclicity of x. With the analysis of this article, we can now assert it more precisely, by considering the accessed field: the path-length decreases if next belongs to the set of non-cyclical fields F_{x} for variable x.

2. Overview of the analyses

We provide here a high-level description of the analyses that we are going to define in the next sections.

Our definite unreachability and non-cyclicity analyses are built over the assumption that the program has been already processed into a graph of basic blocks. There is a subgraph for every method or constructor and those subgraphs are linked at method calls, where each call is bound to an over-approximation of the runtime targets of the call. Bytecodes are assumed to be typed. While Java bytecodes are often untyped, they are guaranteed to be typable by the traditional type inference Kindall algorithm [4]. The construction of the graph and the type inference is already performed in fully implemented tools, such as the Julia analyzer.

Once this preprocessing has been performed, actual static analyses can be performed. We assume that three preliminary static analyses are already available before we run our unreachability and non-cyclicity analyses. They are a definite aliasing analysis between program variables and possible sharing and reachability analyses between program variables. Note that these preliminary analyses do not use field names and are consequently much simpler to define and implement that the new analyses described in this article. They are actually all already available and highly optimized in the Julia analyzer. They are used for these reasons:

- definite aliasing analysis is used to determine variables of a caller method that are definite alias of parameters passed to a callee. If the parameter is not reassigned inside the callee, then its value at the end of the callee stands for the variable of the caller as well and can be used to reconstruct the side-effects of the callee on that variable;
- possible sharing analysis is used at method call, again, since the locations reachable from a variable of the caller that does not share with any parameter of the callee are unaffected by the call itself. This improves the approximation of the side-effects of the call;
- possible reachability analysis is used to clean-up our new unreachability analysis. If a variable does not possibly reach another, then the latter is unreachable from the former, for any set of fields that might be used to state unreachability. This removes spurious pairs of unreachable variables from the approximation and can be seen as the basis over which our new unreachability analysis builds, by providing more fine-grained information that considers the fields used for reachability as well.

These supporting analyses and our new analyses are plugged inside the same framework of analysis. Namely, the graph of basic blocks is translated into a graph where nodes stand for bytecodes and arcs propagate abstract information among nodes, in a monotonic way. Abstract information is propagated until fixpoint, by using any fixpoint strategy. The Julia analyzer includes a fixpoint strategy that uses a workset of arcs still to propagate. Arcs are sequentially picked up from the workset and propagated; other arcs are added to the workset when the approximation of the heads changes. Token of abstract information are kept inside bitsets, for compact representation and efficient propagation. This propagation is extremely efficient for relatively simple analyses such as definite aliasing, possible sharing and reachability. Instead, it might be expensive for complex analyses as those described in this article, that are still to be implemented. This complexity can