Science of Computer Programming 98 (2015) 645–679

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Two techniques to improve the precision of a demand-driven null-dereference verification approach

Amogh Margoor, Raghavan Komondoor *

Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India

ARTICLE INFO

Article history:
Received 3 May 2013
Received in revised form 17 August 2014
Accepted 12 September 2014
Available online 8 October 2014

Keywords:
Dataflow analysis
Weakest pre-conditions

ABSTRACT

The problem addressed in this paper is sound, scalable, demand-driven null-dereference verification for Java programs. Our approach consists conceptually of a base analysis, plus two major extensions for enhanced precision. The base analysis is a dataflow analysis wherein we propagate formulas in the backward direction from a given dereference, and compute a necessary condition at the entry of the program for the dereference to be potentially unsafe. The extensions are motivated by the presence of certain "difficult" constructs in real programs, e.g., virtual calls with too many candidate targets, and library method calls, which happen to need excessive analysis time to be analyzed fully. The base analysis is hence configured to skip such a difficult construct when it is encountered by dropping all information that has been tracked so far that could potentially be affected by the construct. Our extensions are essentially more precise ways to account for the effect of these constructs on information that is being tracked, without requiring full analysis of these constructs. The first extension is a novel scheme to transmit formulas along certain kinds of def–use edges, while the second extension is based on using manually constructed backward-direction summary functions of library methods. We have implemented our approach, and applied it on a set of real-life benchmarks. The base analysis is on average able to declare about 84% of dereferences in each benchmark as safe, while the two extensions push this number up to 91%.

1. Introduction

Null-dereferences are a bane while programming in pointer-based languages such as C and Java. In this paper, we describe a *sound*, *context-sensitive*, *demand-driven* technique to verify dereferences in Java programs via over-approximated weakest pre-conditions analysis. A *weakest pre-condition* $wp(p, C)$ is the weakest constraint on the *initial state* of the program that guarantees that the program state will satisfy the condition C every time control reaches the point p. We define the notion of *weakest at-least once pre-condition*, denoted as $wp_1(p, C)$, as the weakest constraint on the initial state of the program that guarantees that execution will reach p at least once in a state that satisfies C. Note that for any (p, C), $wp_1(p, C) = \neg wp(p, \neg C)$. We can use the weakest at-least once pre-condition to check if a selected dereference of a variable or access-path v at a given program point p is *always* safe. This can be done by checking if $wp_1(p, v = null)$ is *false*. However, the weakest at-least-once pre-condition is in general not computable precisely in the presence of loops or recursion; hence, our approach uses an abstract interpretation [1] to compute an over-approximation of it. After our analysis

* Corresponding author. Tel.: +91 80 22932368.
E-mail addresses: amogh.margoor@csa.iisc.ernet.in (A. Margoor), raghavan@csa.iisc.ernet.in (R. Komondoor).

http://dx.doi.org/10.1016/j.scico.2014.09.006
0167-6423/

```
1: foo(a,b,c) {
2:   if(a ≠ null) {  ⟨b.f = null, a ≠ null, a = null⟩  ≡  false
3:     b = c;         false
4:     t = new...;    ⟨b.f = null, b ≠ c, a ≠ null⟩
5:     c.f = t;       ⟨b.f = null, b ≠ c, a ≠ null⟩,  ⟨t = null, b = c, a ≠ null⟩
6:   }
7:   d=a;             ⟨b.f = null, a ≠ null⟩
8:   if(d ≠ null)     ⟨b.f = null, d ≠ null⟩
9:     b.f.g = 10;    ⟨b.f = null⟩
10:}
```

Fig. 1. Example to illustrate the base analysis. The dereference of field `f` in line 9 is being verified.

terminates, we check if the computed over-approximation of $wp_1(p, v = null)$ is *false*; if yes, it would imply that the precise solution is also *false*, implying the safety of the dereference. On the other hand, if the over-approximation is satisfiable, we declare the dereference as *potentially* unsafe.

Each element in the lattice that we use for abstract interpretation is a representation of a formula in disjunctive normal form, with each literal being a predicate that compares an access path with another access path or with *null*. An access path is a variable, or a variable followed by fields, e.g., $v.f_1.f_2...f_k$, that points to an object (i.e., is not of primitive type). The lattice elements are ordered by implication, where weaker formulas dominate stronger formulas; our join operation basically implements logical *or*. We illustrate our lattice as well as our analysis using an example, shown in Fig. 1. Our notation is to show the formula that holds at the point above any statement to the right of the statement. Also, we enclose each *disjunct* in a formula (except the disjuncts *true* and *false*) within angle brackets, and indicate both conjunctions of predicates within a disjunct as well as disjunctions of disjuncts using commas. Note in the example that there are two disjuncts at the point above line 5, and a single disjunct at all other points. The underlining of certain predicates in the example can be ignored for now, and will be addressed later.

The input to our approach is a dereference that needs to be verified, which we refer to as the *root* dereference. In the example, the root dereference is that of `b.f` in line 9. Therefore, the first step in the approach is to initialize the formula at the point above line 9 to $\langle \texttt{b.f} = null \rangle$, as shown to the right of line 9. The analysis proceeds by propagating formulas in a backwards direction, using conservative transfer functions which over-approximate the weakest pre-condition semantics of each statement. The final result of this propagation at all points is shown in the figure. Assuming that the method `foo` is the entire program, the computed over-approximation of $wp_1(\textit{line 9}, \langle \texttt{b.f} = null \rangle)$, which is shown adjacent to line 2, is *false*; hence, the root dereference is declared safe. We postpone a detailed discussion and illustration of our analysis to subsequent sections in the paper.

1.1. Challenges

The obvious advantage that a backwards analysis such as ours has over a forward counterpart is that it is *demand-driven*, meaning a single selected dereference can be verified. This is a very useful feature in a real world setting, where most changes are incremental and affect only a small part of a program. Thus, the developer will be able to verify the dereferences in the part of code that is modified, without paying the price of analyzing all dereferences in the program. There are several reasons why analyzing a single dereference in the backwards direction can be much more efficient than analyzing all the dereferences in the program using a forwards analysis; we postpone a detailed discussion of this to Section 2.5.

This said, a backwards analysis poses its own set of challenges. The first problem is that in order to obtain high precision we would need to perform *strong updates* on formulas that refer to fields of objects when they are propagated back through "put field" statements that write to fields of objects. However, techniques for performing strong updates on formulas have been proposed in the literature only for forward analyses. These techniques do not carry over naturally to the backward setting. The second problem is the resolution of *virtual calls* in large object-oriented programs. While a forward analysis could potentially use path-specific points-to information [2] to derive a precise set of targets for a virtual call, a backwards analysis would need to rely on imprecise *may points-to* information to identify an over-approximation of the candidate targets at a virtual call.

There are other problems we face that are shared by forward counterparts, too. Java programs make extensive use of libraries; entering and analyzing all library methods would take a heavy toll on the scalability of the technique. The usage of recursive data structures such as linked lists and trees, and the usage of arrays, pose challenges to any analysis, because code that uses these structures is hard to analyze precisely in an efficient manner. *Shape analysis* [3] is a sophisticated technique that has been proposed to handle recursive data structures, but it does not scale to programs of sizes we are interested in its current state of evolution. Finally, context-sensitivity and path-sensitivity are typically required for precision, but can be complex or expensive to implement.

1.2. The base analysis

Our approach consists of two parts: 1) the base analysis, and 2) an extended analysis, which is the base analysis plus two major extensions. The base analysis was originally proposed, discussed, and evaluated by Madhavan and Komondoor [4]. The