Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates

Arnab De and Deepak D'Souza

Department of Computer Science and Automation,
Indian Institute of Science, Bangalore, India
{arnabde,deepakd}@csa.iisc.ernet.in

Abstract. The ability to perform strong updates is the main contributor to the precision of flow-sensitive pointer analysis algorithms. Traditional flow-sensitive pointer analyses cannot strongly update pointers residing in the heap. This is a severe restriction for Java programs. In this paper, we propose a new flow-sensitive pointer analysis algorithm for Java that can perform strong updates on heap-based pointers effectively. Instead of points-to graphs, we represent our points-to information as maps from access paths to sets of abstract objects. We have implemented our analysis and run it on several large Java benchmarks. The results show considerable improvement in precision over the points-to graph based flow-insensitive and flow-sensitive analyses, with reasonable running time.

1 Introduction

Pointer analysis is used to determine if a pointer may point to an abstract memory location, typically represented by an allocation site in languages like Java. A precise pointer analysis has the potential to increase the precision and scalability of client program analyses [29,17]. The precision of pointer analysis can be improved along two major dimensions: flow-sensitivity and context-sensitivity. A flow-insensitive pointer analysis [1,31] computes a single points-to information for the entire program that over-approximates the possible points-to relations at all states that the program may reach at run-time. A flow-sensitive analysis on the other hand takes the control flow structure of a program into account and produces separate points-to information at every program statement. A context-sensitive analysis aims to distinguish among invocations of the same function based on the calling contexts.

Traditionally researchers have focused on improving the scalability and precision of flow-insensitive [14,26,10] and context-sensitive analyses [25,34,33]. Flow-sensitive analyses were found to be expensive and gave little additional pay-off in client applications like memory access optimizations in compilers [16,15,17]. However in recent years, it has been observed that several client analyses like typestate verification [8], security analysis [5], bug detection [9], and the analysis of multi-threaded programs [28], can benefit from a precise flow-sensitive pointer analysis.
analysis. As a result there has been renewed interest in the area of flow-sensitive pointer analysis and the scalability of such analyses, particularly for C programs, has been greatly improved \cite{12, 22, 38, 21, 11, 18}.

Most of these techniques however compute the points-to information as a \textit{points-to graph} (or some variant of it), which as we explain below, can be a severe limitation to improvements in precision for Java programs. A node in a points-to graph can be a variable or an abstract object representing a set of dynamically allocated heap objects. Figure 1(b) shows an example points-to graph. Typically, allocation sites are used as abstract objects to represent all concrete objects allocated at that site. An edge from a variable to an abstract object denotes that the variable may point to that object. Similarly an edge from an abstract object \(o_1\) to an abstract object \(o_2\), annotated with field \(f\), denotes that the \(f\) field of object \(o_1\) may point to the object \(o_2\). Precision improvements of flow-sensitive pointer analyses come mostly from the ability to perform \textit{strong updates} \cite{21}. If the analysis can determine that an assignment statement writes to a single concrete memory location, it can \textit{kill} the prior points-to edges of the corresponding abstract memory location. It requires the lhs of the assignment to represent a single abstract memory location \textit{and} that abstract memory location to represent a single concrete memory location. As abstract objects generally represent multiple concrete objects, the analysis cannot perform a strong update on such objects. This situation is common in Java programs, where all indirect assignment statements (i.e. assignments whose lhs have at least one dereference) write to the heap, and hence traditional flow-sensitive algorithms cannot perform any strong updates for such assignments.

We illustrate this problem using the program fragment of Figure 1(a). The points-to graph before statement L1 is shown in Figure 1(b), where variables \(p\) and \(r\) point to the abstract heap location \(o_1\), \(q\) points to \(o_3\), and field \(f\) of object \(o_1\) points to the object \(o_2\). If the abstract object \(o_1\) represents multiple concrete objects, traditional flow-sensitive algorithms cannot kill the points-to information of field \(f\) of \(o_1\) after the assignment statement at L1 – it may unsoundly kill the points-to information of \(r.f\). Hence the analysis concludes that \(t_1\) may point to either \(o_2\) or \(o_3\) at the end of the program fragment (Figure 1(c)), although in any execution, \(t_1\) can actually point to only \(o_3\). In general, \(p\) could have pointed to multiple abstract memory locations, which also would have made strong updates impossible for traditional flow-sensitive analyses.

In this paper we propose a different approach for flow-sensitive pointer analysis for Java programs that enables us to perform strong updates at indirect assignments effectively. Instead of a points-to graph, we compute a map from \textit{access paths} to sets of abstract objects at each program statement. An access path is a local variable or a static field followed by zero or more field accesses. In the program fragment of Figure 1(a), the points-to set of the access path \(p.f\) can be strongly updated at L1 regardless of whether \(p\) points to a single concrete memory location or not. On the other hand, the points-to set of \(r.f\) must be weakly updated at L1 as \(r.f\) may alias to \(p.f\) at that program statement.

\footnote{All analyses considered in this paper are \textit{field-sensitive} \cite{27}.}