Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 322 (2016) 3–18

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Exploiting Linearity in Sharing Analysis of Object-oriented Programs

Gianluca Amato Maria Chiara Meo Francesca Scozzari

Dipartimento di Economia, Università di Chieti-Pescara, Pescara, Italy

Abstract

We propose a new sharing analysis of object-oriented programs based on abstract interpretation. Two variables share when they are bound to data structures which overlap. We show that sharing analysis can greatly benefit from linearity analysis. We propose a combined domain including aliasing, linearity and sharing information. We use a graph-based representation of aliasing information which naturally encodes sharing and linearity information, and define all the necessary operators for the analysis of a Java-like language.

Keywords: Sharing analysis, linearity, aliasing, object-oriented programming.

1 Introduction

In object-oriented languages, program variables are often bound to complex data structures which may overlap. This is the case for Java programs, whose objects are stored in a shared memory called heap. Discovering whether two data structures may overlap is the scope of sharing analysis. This information is used in program parallelization and distribution: data structures which do not overlap allow the execution of methods on different processors, using disjoint memory. Moreover, it is very useful for improving other kind of analysis, like shape, pointer, class and cyclicity analysis. Sharing properties has been deeply studied for logic programs (e.g., [10,9,12,8,5,2]) and the large literature on this topic has been the starting point for designing our enhanced abstract domain for sharing analysis. In particular, the use of a linearity property [7,9,12,11,3,4] has proved to be very useful when dealing with sharing information (see [6] for a comparative evaluation). We show how the same idea can be rephrased to enhance sharing analysis of object-oriented programs. We propose a new combined analysis of sharing, aliasing and linearity properties for

[1] Email: {gamato, cmeo, fscozzari}@unich.it

[2] The authors would like to thank Fausto Spoto for helpful suggestions.

http://dx.doi.org/10.1016/j.entcs.2016.03.002

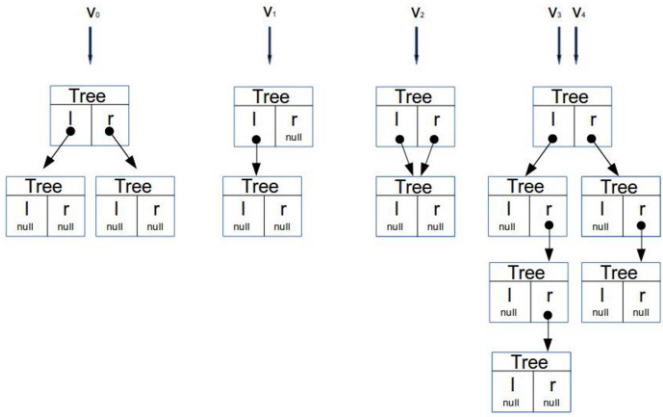

Fig. 1. A concrete state with variables v_0, v_1, v_2, v_3, v_4.

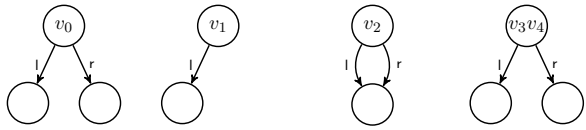

Fig. 2. Abstraction of the concrete state in Fig. 1.

Java-like programs based on abstract interpretation, inspired by the corresponding domains on logic programs.

A concrete state in a object-oriented program is usually described by a *frame*, which is a map from variables to memory locations (or `null`), and a *memory*, which is a map from locations to objects. Our idea is to abstract concrete states into a new kind of graphs we call ALPs graphs. For instance, given the class Tree with two fields l and r, the state in Fig. 1 is abstracted into the ALPs graph in Fig. 2. All ALPs graphs have at most two levels: nodes in the first level are labeled with one or more variables, while nodes in the second level are unlabeled. First-level nodes may have outgoing and incoming edges labeled with field names, while second-level nodes have no outgoing edges. Note that the data structures pointed by the variables v_0 and v_3 are abstracted in the same way, since we do not consider the whole data structure.

1.0.1 Aliasing and nullness.

ALPs graphs may encode *definite nullness* for variables and fields. A variable is definitively null if it does not appear as a label in the graph, while a field v_0.`f` is null when there is no edge labeled with `f` departing from the v_0 node. For example, v_1.`r` is definitively null according to Fig. 2.

The graph also encodes definite *weak aliasing*: two variables (or two fields) are weak aliased when they point to the same location (possibly `null`). In the ALPs graph, this means they are the same node. For instance, the variables v_3 and v_4 in Fig. 1 are in the same node. Moreover, the fields v_2.`l` and v_2.`r` are abstracted into a single node.