Theoretical Computer Science 631 (2016) 43–72

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

A generic framework for heap and value analyses of object-oriented programming languages

Pietro Ferrara

Julia SRL, Verona, Italy

ARTICLE INFO

Article history:
Received 25 February 2015
Received in revised form 10 January 2016
Accepted 1 April 2016
Available online 11 April 2016
Communicated by D. Sannella

Keywords:
Static analysis
Abstract interpretation
Generic analyzers

ABSTRACT

Abstract interpretation has been widely applied to approximate data structures and (usually numerical) value information, but their combination is needed to effectively apply static analysis to real software. In this context, we introduce a generic framework that, given a heap and a value analysis, combines them, proving formally its soundness. We plug inside this framework a standard allocation site-based pointer analysis, a TVLA-based shape analysis, and standard existing numerical domains. As far as we know, this is the first sound generic automatic framework for statically typed object-oriented programming languages combining heap and value analyses that allows to summarize and materialize heap identifiers.

1. Introduction

Two major fields of static program analysis have been heap and (usually numerical) value abstractions. Many heap [1,2] and numerical value [3,4] analyses have been introduced. A priori, any value analysis could be combined with any heap analysis, since they deal with two different types of information, and it is widely accepted [5] that, if one wants to achieve a practical static analysis on *real* programs, this combination is necessary. Nevertheless, heap and value analyses have been studied as orthogonal problems so far, since their automatic combination is known to be a hard problem.

Two main approaches have been followed during the last decade in the static analysis of imperative and object-oriented programming languages: (i) analyzers focused on value information that preprocess the program applying a specific heap analysis, and replace heap accesses with symbolic variables (e.g., Clousot [6]), and (ii) heap abstractions (e.g., TVLA [7]) that do not track value information, or that have to be manually extended (e.g., with specific predicates) to track a particular type of value information [8,9]. As far as we know, existing analyzers that combine heap and value analyses are not both generic (that is, they are specific on a particular heap and/or value analysis) and automatic (that is, they require to provide some annotation, like instrumentation predicates).

1.1. Motivating example

Consider the motivating example in Fig. 1. Class ListInt represents a list of integers, with an integer field f (containing the value of an element) and a ListInt next field (pointing to the next element of the list, or to null if we are at the end). Method absSum(l) computes the sum of the absolute values of the elements in the list. Imagine that two clients

E-mail address: pietro.ferrara@juliasoft.com.

http://dx.doi.org/10.1016/j.tcs.2016.04.001
0304-3975/

call this method. client1 passes the list [1; 2][1] to absSum, where the two elements are allocated at different program labels (p1 and p2). Instead, client2 calls absSum with a list of n positive elements, where n is an input of the program. There are various properties and invariants we would like to prove and infer on such program. First of all, we would like to prove that we do not have any NullPointerException (property P1). In addition, we could discover that the value returned by absSum is positive (P2), or that it is greater than or equal to all the elements in the list pointed to by l (P3). These properties require to combine different heap and value analyses. P1 does not require any particular numerical analysis, and for both the clients a simple and efficient heap analysis based on the allocation sites [10] would be precise enough. Instead, P2 requires at least a numerical domain that tracks the sign of numerical variables, while P3 requires a relational domain like Octagons [4]. In addition, for client1 the allocation site-based heap abstraction would be precise enough both for P2 and P3. Instead, on client2 this abstraction would approximate all the nodes of the list with a unique summary node, and it would not be able to discover that the value added to sum is positive, since it cannot track precise information on the Boolean condition of the if statement. Therefore, we need a more precise heap abstraction that materializes the node pointed to by it (e.g., shape analysis [2]).

```
1 int absSum(ListInt l) {
2     int sum = 0;
3     ListInt it = l;
4     while(it != null) {
5         if(it.f < 0) sum = sum − it.f;
6         else sum = sum + it.f;
7         it = it.next;
8     }
9     return sum;
10 }
```

Fig. 1. The motivating example.

1.2. Contribution

The contribution of this work is the formalization of a sound generic analysis that allows the combination of various heap and value abstractions automatically for object-oriented programming languages. The heap analysis approximates concrete locations through *heap identifiers*, while the value analysis tracks information on these identifiers. In addition, our framework allows the heap analysis to *freely* manage heap identifiers, and in particular to merge and materialize them. These modifications are represented by *substitutions*, and they are propagated to the value analysis. This work targets programming languages where references and values are distinct. In particular, the heap analysis is aimed at abstracting the heap structure, while the value domain abstract the values of non-reference variables and heap locations. This is common in statically typed object-oriented programming languages like Java and C#, but it does not apply to other imperative programming languages like C where references are threated as values (e.g., with pointer arithmetic). For the most part, our approach relies on standard components of abstract interpretation-based sound static analyses, and we formally define and prove the soundness of their combination. In addition, we show how to instantiate our framework with a pointer and a shape analyses, as well as with numerical domains. This proves that our framework is expressive and flexible enough to be applied to some of the most common heap and value analyses.

Compared to existing approaches, the main novelties of our work are (i) the introduction of a neat distinction between the information about values and heap structures via a so-called split domain, and (ii) supporting both strong and weak updates on heap locations of generic data structures.

Novel material. This journal article revises and extends two previous conference papers [11,12]. In particular, this current work adds the following content:

1. The language presented in Section 2 has been extended to support Boolean conditions.
2. Sections 3 and 4 add to the formal definitions presented in Section 3 and 4 of [12] the full formal proofs of soundness of the split and abstract domain and semantics. In addition, the abstract semantics has been modified to support weak updates.
3. Section 5 presents the formal instantiation of the framework introduced by Sections 3 and 4 to a standard pointer analysis.
4. Section 6 recalls some definitions of Section 4 of [11] adding the formal definitions and the soundness proofs of the TVAL+ domain and semantics. In addition, [11] relied on ad-hoc formalization of heap identifiers and replacement functions, while Section 6 plugs it into the new framework introduced in Section 3 and 4 redefining several components.

1.3. Overview of the framework

Domains. Fig. 2 depicts the overall structure of our approach. On the left, we have standard object-oriented states composed by an environment and a store. On the right, we have our target abstract domain composed by a heap and a value abstract state. Here we represent the state of `client1` when calling method absSum in our motivating example. We adopt an allocation site-based heap abstraction [1] and the Interval domain [13]. Therefore, the heap analysis abstracts the list with two abstract nodes named p1 and p2, while the value analysis tracks that field f of p1 is [1..1], and field f of p2 is [2..2].

[1] [1; 2] is a shortcut to denote a list of two elements, with value 1 stored in the field f of the first element list, and 2 in the second one.