Journal of Logical and Algebraic Methods in Programming 95 (2018) 103–127

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Field-sensitive sharing

Damiano Zanardini

Universidad Politécnica de Madrid, Spain

A R T I C L E I N F O

Article history:
Received 14 February 2017
Received in revised form 31 August 2017
Accepted 2 October 2017
Available online xxxx

A B S T R A C T

In static analysis of programming languages with dynamic memory, *sharing analysis* tries to infer if two variables point to data structures which are not disjoint. I.e., two variables *share* at a certain program point if there is a memory cell which can be accessed from both via two *converging paths* in the heap.

Sharing information is used as an auxiliary component in a number of static analysis techniques: to know that two variables *do not share* any memory cell allows to guarantee that any modification to the first variable has no effect on the data structure accessible from the second. On the other hand, if it cannot be guaranteed that the data structures accessible from two variables x and y are disjoint, then a loss of information occurs in that any update to x must be considered as a *possible update* of y, thus making the inference of interesting program properties much harder.

This paper introduces a novel sharing analysis which takes into account the *fields* involved in converging paths. For every two variables and every program point, a *propositional formula*, called a *path-formula*, is computed, that describes which fields may or may not be traversed by converging paths in the heap. Let x point to an object representing a phone contact, x.f point to the beginning of a single-linked list (the user's phone numbers), and x.g point to other information. On the other hand, let y point to the second element of the number list. In this case, x and y share, so that existing analyses based on sharing have to admit that an update of x.g may modify y, thus invalidating previous information about it. However, field-sensitive information like "for every two converging paths π_x and π_y from x and y, respectively, π_x does not traverse field f" allows to infer that the data structure pointed to by y will not be modified by updating x.g.

Besides improving existing static analysis techniques, the field-sensitive sharing analysis is interesting in itself, and can be formalized in the framework of *abstract interpretation* as a refinement of traditional sharing.

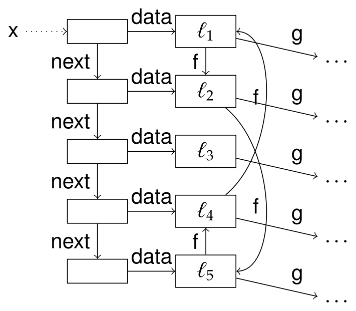
1. Introduction

Programming languages with dynamic memory allocation, such as Java or C++, allow creating and manipulating linked data structures in the *heap*. This feature threatens most approaches to static analysis since keeping track at compile time of the possible runtime configurations of the heap is quite challenging.

One of the main sources of precision loss in static analysis of the heap is the possibility that the data structures accessible from two different variables are not disjoint: in this case, the variables are said to *share* some part of the heap, i.e., there are heap locations which can be accessed from both. This is quite a common picture in heap-manipulating object-oriented

E-mail address: damiano@fi.upm.es.

https://doi.org/10.1016/j.jlamp.2017.10.005
2352-2208/© 2017 Elsevier Inc. All rights reserved.



```
1 while (x≠null) do {
2   y := x.data;
3   y.f := ...
4   y.g := ...
5   x := x.next;
6 }
```

Fig. 1. A list storing mutually connected data, and a loop traversing it.

languages. The problem for static analysis techniques stems from the risk that a modification to one variable affects the data structure pointed to by the other, although this is not directly visible in the source code. *Sharing analysis* aims at proving that two variables do not share in order to rule out the possibility that an update to one breaks some interesting properties of the other. This information has been successfully used in a number of program analysis techniques and is indeed one of the fundamental components of most approaches to static analysis of heap-manipulating languages.

The present paper introduces a sharing analysis which is not tied (as many shape analyses are) to a pre-defined set of data structures or *shapes*, and can infer precise information about the fields involved in paths in the heap. It is true that most programs exhibit quite simple sharing patterns [23], so that, in many cases, there seems to be little need for a very precise sharing analysis (remember that precision is usually a major threat to scalability). However, precision *does* matter, and a significant part of the research on static analysis has always been devoted to improve the inference of interesting properties of programs (which has often become scalable in a later time). To the best of our knowledge, no existing analysis is able to infer such information, although, clearly, there exist techniques which can prove other interesting properties of the heap.

Consider a loop on the data structure drawn in Fig. 1. Traditional sharing correctly infers that y shared with x after line 2. Therefore, the alteration undergone by y in lines 3 and 4 is usually considered as potentially modifying the data structure pointed to by x. This makes it impossible to prove the *termination* of the loop since (1) there is a cycle on y (locations ℓ_1, ℓ_2, ℓ_5, ℓ_4); and (2) this cycle is accessible by x. Cycles in data structures usually make it impossible to prove the termination of loops traversing[1] the structure, because no decreasing *ranking function* can be found [5,13,1,35].

However, a closer look allows to observe that any location shared by x and y is reachable (1) from x, by traversing either (1a) data; (1b) next and data; (1c) data and f; or (1d) next, data and f; and (2) from y, by traversing either (2a) no fields (i.e., the shared location can be exactly the location directly pointed to by y); or (2b) just f. A termination analyzer could use this information to infer that, by executing x:=x.next, x never accesses a cycle because field data is never traversed. Consequently, the termination of the whole loop could be proved.

The present field-sensitive sharing analysis aims at providing such kind of information about *how* variables share. This information is computed by an *abstract semantics* that manipulates *propositional formulæ* representing, for a pair of variables, the fields which have to be traversed by heap paths reaching the same location (*converging paths*). For example, suppose that next, data, f and g are the only fields in the program: the above sharing information between x and y (i.e., the pair (x, y)) can be represented as

$$\mathsf{data}^{\searrow} \land \neg\, \mathsf{g}^{\searrow} \land \neg \mathsf{data}^{\swarrow} \land \neg \mathsf{next}^{\swarrow} \land \neg\, \mathsf{g}^{\swarrow}$$

meaning that (1) any path from the first variable x (downward arrows from left to right correspond to the first variable in the pair) to a shared location has to traverse data (as indicated by $\mathsf{data}^{\searrow}$) but not g (as specified by $\neg\,\mathsf{g}^{\searrow}$), and, implicitly, may traverse either next or f, or both; and (2) any path from y (corresponding to arrows from right to left) can only possibly traverse f, the traversal of the other fields being explicitly forbidden ($\neg \mathsf{data}^{\swarrow} \land \neg \mathsf{next}^{\swarrow} \land \neg\, \mathsf{g}^{\swarrow}$).

Following the well-known theory of *abstract interpretation* [14], an *abstract domain* representing field-sensitive sharing is defined (Section 3), and a sound *abstract semantics* is built on it (Section 4), which computes the desired information. A prototypical implementation of the abstract semantics is presented in Section 5.

Main contributions

As discussed in Section 1.1, a closely-related work [38] introduces field-sensitive reachability and cyclicity analysis. The present paper builds on that work and adds the following contributions:

[1] To "traverse" a field, a path in the heap or a whole data structure has to be understood as field dereference: the path from the location pointed to by v to the one pointed to by $v.f$ traverses φ.