Computer Languages, Systems & Structures 40 (2014) 99–111

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

A sparse evaluation technique for detailed semantic analyses[☆]

Yoonseok Ko[a], Kihong Heo[b], Hakjoo Oh[b,*]

[a] KAIST, South Korea
[b] Seoul National University, South Korea

ARTICLE INFO

Article history:
Received 4 October 2013
Received in revised form
28 April 2014
Accepted 8 May 2014
Available online 20 May 2014

Keywords:
Static analysis
Abstract interpretation
Sparse evaluation
Data-flow analysis

ABSTRACT

We present a sparse evaluation technique that is effectively applicable to a set of elaborate semantic-based static analyses. Existing sparse evaluation techniques are effective only when the underlying analyses have comparably low precision. For example, if a pointer analysis precision is not affected by numeric statements like `x:=1` then existing sparse evaluation techniques can remove the statement, but otherwise, the statement cannot be removed. Our technique, which is a fine-grained sparse evaluation technique, is effectively applicable even to elaborate analyses. A key insight of our technique is that, even though a statement is relevant to an analysis, it is typical that analyzing the statement involves only a tiny subset of its input abstract memory and the most are irrelevant. By exploiting this sparsity, our technique transforms the original analysis into a form that does not involve the fine-grained irrelevant semantic behaviors. We formalize our technique within the abstract interpretation framework. In experiments with a C static analyzer, our technique improved the analysis speed by on average 14 ×.

1. Introduction

In static analysis, the technique of sparse evaluation has been widely used to optimize the analysis performance [4,27,11,26,17,14]. Sparse evaluation is based on the observation that static analysis sometimes aggressively abstracts program semantics and therefore a number of program statements are irrelevant to the analysis. For instance, typical pointer analyses (e.g., [14,15]) have comparably low precision and not affected by numeric statements such as `x:=1`. The goal of existing sparse evaluation is to remove such irrelevant statements, which makes the analysis problem smaller and improves the analysis' scalability. In the literature, sparse evaluations have been effectively used to improve the performance of pointer analysis [14,30] and classical data-flow analyses [4,27].

However, existing sparse evaluation techniques are not effective for elaborate semantic analyses in general. Note that the basic assumption of existing sparse evaluation is that the given analysis problem is simple-minded and hence a number of program statements are irrelevant to the analysis. However, in general semantic analyses, it is not uncommon that the analysis is so detailed that the assumption of sparse evaluation does not hold. That is, such an analysis considers all types of values (e.g., including both numbers and pointers) and almost all statements in the program are relevant to the underlying analysis. For instance, consider an elaborate pointer analysis that considers not only pointers but also numeric values.

[☆] This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning (MSIP)/National Research Foundation of Korea (NRF) (Grant NRF-2008-0062609).
[*] Corresponding author.
E-mail address: pronto@ropas.snu.ac.kr (H. Oh).

http://dx.doi.org/10.1016/j.cl.2014.05.001
1477-8424/© 2014 Elsevier Ltd. All rights reserved.

Then statements like `x:=1`, which would be irrelevant to simple pointer analyses, are no longer irrelevant and cannot be removed by existing sparse evaluation techniques.

In this paper, we present a new sparse evaluation technique for such detailed static analyses. The intuition behind our technique is that, even though an analysis is detailed and a statement is relevant, the analysis is still sparse in a fine-grained way: analyzing the statement uses only small part of its abstract memory. For example, consider analyzing the statement `x:=1`. The abstract semantics for the statement would update only the value of `x` but other values, say `y`, are not involved in the analysis of the statement. To exploit this sparsity, we first reformulate the analysis problem into an equivalent form that is more fine-grained in that semantic equations are expressed explicitly in terms of individual abstract locations. We call this step decomposition. Then, we define two elimination procedures (which we call no-change and no-contribution eliminations, respectively) that remove the fine-grained irrelevant behaviors of the analysis. We present our technique in the abstract interpretation framework [8,6] and prove that the decomposition and elimination procedures are semantics-preserving, which means that our technique maintains the original analysis' soundness and precision.

Our work provides a general and flexible alternative to the recent sparse analysis framework [23]. Although the goal of ours and the sparse analysis in [23] is the same (i.e., making the analysis sparse), the techniques used are different: the technique in [23] constructs def-use dependencies while we eliminate unnecessary dependencies. This difference makes our technique flexible in controlling the sparseness of the final analysis. We discuss this point in Section 8 in more detail. Furthermore, we generalize the idea of sparse technique in the abstract interpretation framework with arbitrary trace partitioning.

We show the effectiveness of our technique in a realistic setting. We implemented our technique on top of Sparrow, an interval domain-based abstract interpreter [18,16,21,22,24]. In experiments, our technique improved the analysis speed from 2 to 59 times, on average 14 times, and reduced peak memory consumption by 29–80%, on average 56%, for a variety of open-source C benchmarks (6K–111K LOC).

Overview: We illustrate our technique with an example. Suppose that we analyze the program in Fig. 1(a) with a non-relational analysis: the abstract state of the analysis is a map from the set of abstract locations (simply variables x and y in this example) to abstract values, say numeric intervals [8]. During the analysis, the first statement defines the value of x, the second statement defines y, and the last statement updates y with the value of x. Observe that existing sparse evaluation techniques remove no statements in this example, because all the three statements have some effects on the analysis. On the other hand, our technique works as follows:

1. We reformulate the analysis (Fig. 1(a)) into the decomposed form (Fig. 1(b)) in which values of each abstract location are computed separately: each instruction c in Fig. 1(a) is split into (c, x) and (c, y) that hold the values of x and y, respectively. For instance, at node $(1,x)$, the value of x at instruction 1 is stored and value for y at instruction 1 is stored at $(1,y)$. Edge $(c,x)\to(c',x')$ means that the value of x' at c' may depend on the value of x at c.
2. In Fig. 1(b), we eliminate nodes that have no effect on the analysis, which we call no-change elimination. Note that, among the six nodes in Fig. 1(b), values are actually updated at nodes $(1,x)$, $(2,y)$, and $(3,y)$. At other nodes (dotted ones in Fig. 1(b)), values are not changed. The goal of this step is to remove such "no-change" nodes from the analysis. We simply remove the nodes and short-circuit their in/outflows, which results in Fig. 1(c). Observe that, after no-change elimination, the value of x from $(1,x)$ is directly propagated to $(3,y)$.
3. In Fig. 1(c), we remove irrelevant input flows, which we call no-contribution elimination. In Fig. 1(c), only the value of x is necessary to update the value of y at $(3,y)$; the value of y is not used at $(3,y)$. So, we remove the "no-contribution" flow $(2,y)\to(3,y)$. Other unnecessary flows are also removed, leading to Fig. 1(d).

With our technique, the original analysis problem (Fig. 1(a)) is reduced to a smaller problem (Fig. 1(d)). Obviously, fixpoint computation for the smaller problem will be cheaper than the original analysis.

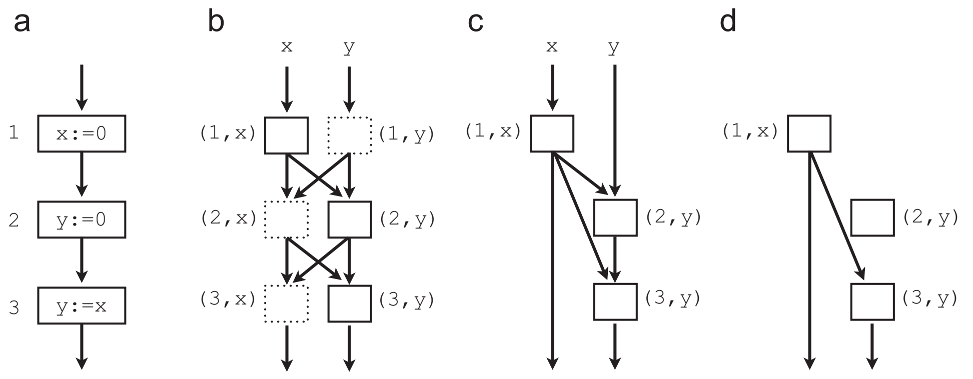

Fig. 1. Example: (a) original analysis, (b) after decomposition, (c) after no-change elimination, and (d) after no-contribution elimination.