

CMSC 631
Composing Dataflow Analyses and Transformations
- Report

Polyvios Pratikakis
polyvios@cs.umd.edu

December 10, 2002

1 Introduction

This paper presents a framework that allows several independent dataflow analyses which perform optimizations to be composed and applied concurrently. This is done so that every analysis will benefit from the transformations made to the program by all other analyses.

Dataflow analyses are mostly used in optimizing compilers, to deduce transformations that can be applied on the original program, for optimization purposes, without changing the program's behavior. Obviously, different analyses can benefit from each other's results or transformations. Therefore, we want the analyses to be able to "communicate," in order to maximize the optimizations applied to the program. Instead of manually writing a "combined" analysis by hand, in [3] a framework is proposed that tries to automatically combine various analyses. That way, each analysis is independently implemented, and it's the framework's responsibility to combine the analyses in a correct way.

2 Dataflow analyses interaction

"Communication" between the separate modules of data flow analysis is achieved through the representation of the program as a Control Flow Graph. Specifically, the various analyses applied perform their transformations on the CFG of the program, by replacing nodes of the CFG with subgraphs.

One might think that the paper contradicts itself in that. This is because in presenting the mathematical model of the analyses it is asserted that the replacement of a node by a graph is syntactically valid, i.e. the graph has the same "interface" (input/output edges) as the node it replaces. However,

in the example of analysis given in the introduction, a node with two output edges (if..then..else) is replaced with a node with one output edge (goto).

This is not a problem if we relax the above assertion to the replacement graph having *at most* as many output edges as the replaced node, with the case of less output edges resulting into never executed code, which is removed by the *dead code elimination* supported by the compiler framework.

The “super analysis” is constructed by having the replacing subgraphs recursively analyzed using all other analyses, the fixpoint solution being the final replacing graph. “Recursively” means that for every node of the replacing graph, the “super analysis” is repeated, until the replacement graph reaches a fixpoint, where it is no longer transformed by any of the analyses constituting the “super analysis.”

This framework requires that the optimizing transformations that result from an analysis, are applied whenever possible during the analysis, at the time a CFG node is analyzed. This is modeled by the flow function used in the analysis having the option to return a replacement graph for that node, instead of the flow values of the node’s output edges.

3 Structure of the paper

In order to be able to define the “concurrent” (more like interleaved) application of all analyses on the CFG, the authors define in turn:

- The concept of performing the analysis before all transformations, which is the “common” type of data flow analysis application found in compilers.
- Integrating the transformations with the analysis, so that the graph is transformed by an analysis as this is computed iteratively over the CFG. This is achieved by augmenting the notion of a data flow function to be able to return a sub-graph instead of the output values for a node, in the cases when the node is to be replaced by that subgraph.
- Combining many analyses that perform transformations during computation, to create a “super analysis.” This is achieved by defining a framework that allows all the analyses to be carried simultaneously for every node scanned. In the case that one of the data flow functions of the analyses combined returns a replacement graph for a node, the graph is analyzed using *all* analyses, and the result is substituted into the common CFG used by *all* analyses, not just the one that resulted in the transformation. That way, all analyses know about a transformation as soon in the CFG as it “happens.”

For each of these steps, the soundness of the model is proven in [2], and termination of the algorithm is studied. Whether the “super analysis” terminates

depends on the termination of the separate analyses that it combines. Also, even when the various analyses in separate might terminate, their successive application might end in a “super analysis” that oscillates infinitely between two or more states, because of mutually cancelled transformations.

4 Results - Conclusions

The experimental results of the framework’s use show that with a lot less effort in writing independent and modular analyses, one can achieve comparable compilation and execution times, almost as if the “super analysis” was hand-written.

The framework presented in this paper can only handle intraprocedural analysis of programs, but an interprocedural approach is presented in [1]. The interprocedural approach “inlines” the procedures in every call site using intraprocedural analysis, avoiding re-computation of transfer functions for procedures that have already been analyzed.

References

- [1] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical report, University of Washington, November 2002.
- [2] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. Technical report, University of Washington, November 2001.
- [3] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282. ACM Press, 2002.