Report on "**Composing Dataflow Analyses and Transformations**"
by S. Lerner, D. Grove and C. Chambers

Report by Konstantinos Bitsakos, Dec. 5 2002

This paper addresses the problem of writing dataflow analyses in a modular way and combining them in order to create a super-analysis. Their approach is to export the results of an individual analysis to all other analyses using flow graph transformations. So, depending on the input values and the current node of the program graph, the result (flow function) of each individual analysis can be a set of abstract values or a new sub graph to replace the current node. Applying the super-analysis recursively to the new sub graph can lead to further graph transformations (by the same or other analyses). When this procedure reaches a fixed point, then the current state of the graph is saved. The most common statement (node) transformations are:

- *ReplaceResult*: replacement with another statement (e.g. constant-folding)
- *ReplaceGraphResult*: replacement with an entire subgraph of statements (e.g. inlining)
- *DeleteResult*: deletion (e.g. dead code elimination)

Inside the paper they say that "a replacement graph must have the same number of input and output edges as the node it replaces and its nodes and edges must be unique". At the same time they give an example, where they replace an "if" statement with a "goto" statement. It seems to me, that this replacement is not valid according to their requirements.
From a mathematical point of view, they start with the formal definition of a single Analysis followed by Transformations (*AT*), next they integrate the Analysis and the Transformation (*IT*) and at the final stage they combine multiple analyses to create a super-analysis (*CA*). At each step they formulate the soundness criteria using results from previous steps and they prove that their framework is sound.
It is clear from the context, that their framework can only integrate analyses with the same direction (forward or backward).

Interesting points:

- Their framework is independent of the exact representation of the program. Most of their work is done in the Vortex compiler, which uses a standard CFG representation. Recently they experimented with Whirlwind compiler, which uses a dataflow graph (DFG) representation with control-edges.

- How do they select between different replacement graphs (what is the PICK function they mention)? Inside a technical report they say that clients of the composed analysis interface, specify the order of component analyses when they create a composed analysis and according to this order the first component analysis to choose a transformation takes precedence over any later analyses. They also mention that the experimental results show that there is a natural ordering of transformation passes.

- *Intraprocedural analysis*. In this paper they don't say much about intraprocedural analysis. In a referenced report they say that intraprocedural analysis determines whether the analysis is flow-sensitive or insensitive, but they don't specify how this is done.

- *Interprocedural analysis*. Interprocedural analysis is based on the intraprocedural framework. The interesting thing about it, is that you can define various levels of context sensitivity, by defining how many input-output pairs the framework keeps for each function. For example, if it keeps only one pair per function, then the analysis will be context-insensitive. In the presence of recursion the framework initially selects the most optimistic approximation for a recursive function and replaces each subsequent function call with the same calling context with the optimistic output. If at the end of the analysis, the real output is

different from the optimistic one, the analysis is performed again with the real output. Eventually the analysis reaches a fixpoint solution.

Limitations

- They do NOT guarantee that the procedure of graph transformations will eventually terminate. Even if each single analysis terminates, the interaction between multiple analyses can lead to an infinite loop of transformations.

- They only support local graph transformations, so some optimizations like loop-invariant code motion and instruction scheduling cannot be performed in their framework. This is a very serious drawback in my opinion, because most of the times it's almost impossible for the programmer to guess the optimum ordering of program statements for that particular machine, due to the variety/complexity of the hardware (pipelined super scalar CPU, virtual/cache memory etc). So the programmer expects from the compiler to optimize his code. In a later technical report they describe a way to implement an intraprocedural loop invariant code motion optimization in their framework prior to the super analysis. The way they do it, is by executing an iterative analysis to identify loop-invariant calculations and insert them in loop preheaders. When the super-analysis performs common subexpression elimination and copy propagation, it eliminates or reduces the cost of the redundant calculations inside the loop.

- Even if they don't require the flow function to be monotonic in order for their framework to be sound, they do require it, in order to guarantee precise results. In fact each intermediate solution is monotonic, because they take the meet of the current solution with the previous one. So, if in one iteration the result is "top", then there is no way to go down the lattice. The fixed point solution will be "top" as well.

Experimental results

For their experiments they performed class analysis, splitting, inlining, constant propagation and folding, common sub-expression elimination, removal of redundant loads and stores and symbolic assertion propagation in 10 Cecil small to medium (up to 50KLOC) programs. According to the performance numbers given, their approach compared with the modular-once approach produce 3-13 times better results (in the same compilation time), while compared with the modular-iterated approach produce the same results but the compilation time was much less. This leads us to believe that most of the benchmark programs exhibit non-trivial mutually beneficial interactions, but they don't take advantage of the "on the fly"/parallel transformation capabilities of their approach. They only give us the relative compilation/running time of the tests. It would be interesting to include the absolute compilation time as well.

Extensions

- Implement "*snooping*", that allows flow functions of an analysis to look at the dataflow values produced by other analyses. This is useful mainly for pure analyses.

- Develop an incremental version of the interprocedural analysis framework in order to support selective reanalysis and recompilation after incremental program changes during interactive program development.

- Use some methods to speed up analyses such as denser bit-vector-based representations, copy-on-write tables and sparser dataflow analysis model.