Points-to analysis for C and Java

Dimitris Vardoulakis

@inproceedings{steens,
    author = {Bjarne Steensgaard},
    title = {Points-to Analysis in Almost Linear Time},
    booktitle = {Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages},
    pages = {32--41},
    year = {1996}
}

This paper presents a flow-insensitive, monovariant points-to analysis for C based on unification. The analysis produces one points-to graph for the whole program and runs in almost linear time. Nodes in the graph represent locations and edges represent points-to relations from source to sink nodes.

The main difference of this analysis from its predecessors is its running time. Andersen's analysis [1], also flow-insensitive and monovariant, is cubic. Polyvariant analyses [2, 4] are more precise but have exponential complexity, though in practice this rarely happens. The linear running time allows us to make a simple observation about the analysis precision: the points-to graph must be linear in the size of the program. Thus, the out-degree of each node must be bounded by a constant. Therefore, all locations pointed to by the same location must be merged in one node in the graph, to avoid a quadratic number of edges.

@inproceedings{fahndrich,
    author = {Manuel Fahndrich and Jeffrey Foster and Zhendong Su and Alexander Aiken},
    title = {Partial Online Cycle Elimination in Inclusion Constraint Graphs},
    booktitle = {Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation},
    pages = {85--96},
    year = {1998}
}

This paper presents an efficient implementation of Andersen's analysis [1]. The authors observe that set-expressions in cyclic constraints can be represented
by a single node in the constraint graph, which greatly reduces the work of the analysis. They also note that using the inductive-form representation of a graph makes the analysis faster than using the standard form.

Andersen’s analysis is based on set constraints. It produces a points-to graph that can have a quadratic number of edges. The main difference from Steensgaard’s algorithm is the treatment of assignments. For \( x = y \), Andersen makes the points-to set of \( y \) a subset of the points-to set of \( x \). Essentially, this adds edges from \( x \) to all \( y \)'s successors in the graph. If the points-to set of \( y \) grows later in the analysis, the assignment statement has to be revisited, which makes the running time cubic. Andersen’s analysis is more precise than Steensgaard’s.

An extension of Andersen’s algorithm to Java. The analysis generates constraints for reachable methods only, so it does not analyze dead code. It discovers targets of calls on the fly, using the current points-to results. Library code is analyzed once and used for many client programs. The analysis works on Java bytecode.

This paper uses Fähndrich et al.’s formulation of Andersen’s algorithm. It extends inclusion-constraint graphs by annotated edges and uses cycle elimination. The language of set-expressions is extended with projections, which makes it more intuitive than the language of Fähndrich et al.

This paper presents a framework for points-to analysis parameterized over Andersen’s or Steensgaard’s algorithm. The analysis works on bytecode. It does
not incorporate reachability, but analyzes the whole user program instead. Libraries are treated as unanalyzed code, for which the analysis makes conservative assumptions. It uses Java’s type information to track objects of type $t$ passed to unanalyzed code. Then, when unanalyzed code returns an object of type $t$ the analysis updates the points-to sets with objects of that type that have escaped.

In my opinion, the most important contribution of this paper is the treatment of unanalyzed code, which is earlier than and similar to Reppy’s modular analysis for ML [3]. Other than that, the analysis framework is fairly standard, it is on-the-fly constraint generation in disguise. The authors implement both algorithms in their framework and observe similar running times. They attribute this to the imprecision of Steensgaard’s algorithm, which cascades when finding possible target methods at call sites. However, the authors fail to spot another factor that slows down Steensgaard’s algorithm. They observe that unification can give ill-typed points-to edges and modify the assignment rule to do conditional merging using type information. But the symmetric treatment of assignments is key to making Steensgaard’s algorithm linear; by being selective their implementation is doing more search.

Another extension of Steensgaard’s and Andersen’s algorithms to Java. The authors try two approaches for handling of fields. The first approach distinguishes between the same field of different objects (also used by the other two Java papers). The second does not, instead it uses one variable per class field. To deal with dynamic dispatch, the authors try three techniques: class-hierarchy analysis (CHA), rapid type analysis (RTA) and using the points-to results on the fly (also used by the other two Java papers). According to their benchmarks, RTA is faster than on-the-fly resolution and only slightly less precise. They also use heuristics to avoid computing a points-to set for this at method calls. Last, for Collections and Maps the authors propose that the user provide a model for how these classes should behave. This model is used during points-to analysis.

Their alternative approach to field handling is efficient, because it creates points-to sets for fewer variables than the standard approach. The speed gain comes with only a slight loss in precision. Because of encapsulation in Java programs, fields are usually modified in the code of the class. Since analyses approximate
all runtime objects that can be created at the same program point with one abstract object, encapsulation lowers the precision of standard field handling (or equivalently, increases the precision of the alternative approach). A drawback of this work is the user-defined models for Collections and Maps; it requires a lot of effort from the user.

References


