Pointer Analysis

B. Steensgaard: **Points-to Analysis in Almost Linear Time.** POPL 1996

M. Hind: **Pointer analysis: haven't we solved this problem yet?** PASTE 2001

Presented by Ronnie Barequet
23.03.2014

Pointer Analysis is a field of algorithms aimed to analyze the locations a pointer refers too. There is a large variety of algorithms in this field, focusing differently on precision, running-time, and other dimensions. Pointer analysis constructs a **points-to graph**. In this graph, nodes are program variables, and an edge (a,b) represents a points to relation from 'a' to 'b'.

Pointer analysis is motivated by several applications. Foremost, it computes the regions in memory the current portion of the code accesses. This is essential to liveness analysis, which computes which memory locations will never be accessed at given time. Liveness analysis is valuable for garbage collection at runtime, compile-time optimization, and improvement in efficiency and accuracy of further static analysis, even pointer analysis [Khedker-Mycroft-Rawat 12]. Pointer analysis is also beneficial to other runtime optimizations. For example, computation of related pages, which decides which pages should be cached together.

Pointer analysis is the basis of more elaborate static analysis, such as data flow analysis, alias analysis (aimed to examine when two expressions are equal), typestate analysis, and formal verification.

In the field of pointer analysis, there are many dimensions of precision, which are limited due to complexity requirements. A given algorithm may or may not be **interprocedural**, i.e., analyze passage of data through procedures.

An analysis may be required to be **flow sensitive**, that is, compute state of program (points-to graph, in our case). **Flow insensitive** analyses computes a last upper bound of all states. **Context sensitive** approaches treat each function call separately, at the expense of high running time. A **context-insensitive** algorithm merges information from all callsites entering a function.

Another important dimension of the problem is the abstract representation of memory, used to model usage of dynamic memory. A common solution is to introduce one node for each alloc statement. We note that more delicate methods can be used.

We observe that pointer analysis, and other static analyses, target to solve undecidable problems. Therefore, we apt to solve an approximate problem instead, by abstracting the dimensions of the problem. **Sound static analysis** over approximates, that is, identifies all properties, but may

introduce more false ones. On the other hand, **complete analysis**, reports only actual properties, but might not report all properties. We will discuss primarily sound analysis.

Steensgaard's algorithm is a very basic pointer analysis. It is flow and context insensitive. Essentially, it views memory slots as abstract types in a program, and restricts each pointer to point at only one type, thereby introducing classes of equivalency of memory regions, which the analysis seeks to compute. Using a union-find data structure, this algorithm can be implemented quite efficiently, and can even be modified to be more sensitive to context. However, it is very course.

Alternatively, **Andersen** suggested another flow and context insensitive algorithm based on subset constraints on points-to-sets, where the points-to set of a variable is the set of regions it may point to, and is portrayed by the points-to graph. Constraints are obtained by translating operations to relations between points-to sets. A minimal points-to graph satisfying all constraints can be computed polynomially, and further optimizations where shown make the analysis of Linux kernel tractable.

In practice, sensitivity to fields inside structs, arrays, etc. is a very important granularity of pointer analysis. Context insensitivity is mostly not very costly in precision. However, "star" procedures, which are called from many sources (such as memcpy, for example), must be analyzed with greater care. Such procedures are usually easy to find.

Measuring and comparing the different pointer analysis algorithms is important, but hard because the different implementations use different abstractions of memory. A classic technique is to score the average size of may-equal set. Another technique is to compare to dynamic (obtained by fuzzer) reports. However, the bottom line is that the specific client's precision is the goal, and different clients may require diverse areas of focus.

A large amount of research has been made yielding variations of the mentioned pointer analysis methods, yielding more precise yet still tractable algorithms. Das's algorithms uses Andersen's more expensive yet more precise method selectively, and applies Steensgard's elsewhere. Milanova suggested the concept of object sensitivity, in which each method call is treated separately according to its invoking instance of the class. Plenty more ideas were researched over the years.

In conclusion, Pointer analysis is a complex and highly researched area in static analysis. It is useful in both optimization and program analysis. There are many dimensions to this problem, and different clients may require different special emphasis. The flow insensitive analysis presented is feasible nowadays for large programs, and may readily serve basis for more elaborate analyses. It seems that the challenges lie in adding more sensitivity to the calling context, or even taken path. Another major issue is the modeling of the heap, from which emerged shape analysis research.

Discussion

During the discussion after the presentation the following points came up:

- Context sensitivity –
    - Seems vital, but hard to measure to which extent is necessary

    - Das found that one level of context sensitivity achieves most desired sensitivity (in C programs).
- Field sensitivity
    - Important issue.
    - Explicit offsets are easy to analyze
      Variable offset are hard to analyze precisely. How to abstract?
        - A possible solutions is to unite access to variable offset ‘i’.
        - Range analysis may be a non-expensive method which can improve precision.
- Pointer analysis is useful for optimization:
    - parallelization of independent computations
    - garbage collection hints

References

- B. Steensgaard: **Points-to Analysis in Almost Linear Time**. POPL 1996: 32-41
- M. Hind: **Pointer analysis: haven't we solved this problem yet?** PASTE 2001: 54-61
- U. P. Khedker, A. Mycroft, P. Singh Rawat: **Liveness-Based Pointer Analysis**. SAS 2012: 265-282
- S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, E. Geay: **Effective typestate verification in the presence of aliasing**. ACM Trans. Softw. Eng. Methodol. 17(2) (2008)
- B. Hardekopf, C. Lin: **The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code.** PLDI 2007: 290-299
- O. Lhoták, K. Chung: **Points-to analysis with efficient strong updates**. POPL 2011: 3-16
- A. Milanova, A. Rountev, B. Ryder: **Parameterized object sensitivity for points-to and side-effect analyses for Java.** ISSTA 2002: 1-11
- M. Das: **Unification-based pointer analysis with directional assignments**. PLDI 2000: 35-46
- M. Shapiro, S. Horwitz: **Fast and Accurate Flow-Insensitive Points-To Analysis**. POPL 1997: 1-14
- A. Rountev, B. Ryder, W. Landi: **Data-Flow Analysis of Program Fragments**. ESEC / SIGSOFT FSE 1999: 235-252