Pointer Analysis for C Programs

Mohit Bansal
Student, Departnment Computer Science Engineering
Dronacharya College of Engineering, Gurgaon

Abstract- **We present a pointer analysis algorithm designed for source-to-source transformations. Existing techniques for pointer analysis apply a collection of inference rules to a dismantled intermediate form of the source program, making them difficult to apply to source-to-source tools that generally work on abstract syntax trees to preserve details of the source program. Our pointer analysis algorithm operates directly on the abstract syntax tree of a C program and uses a form of standard dataflow analysis to compute the desired points-to information. We have implemented our algorithm in a sourceto-source translation framework and experimental results show that it is practical on real-world examples.**

I. INTRODUCTION

The role of pointer analysis in understanding C programs has been studied for years, being the subject of several PhD Thes is and nearly a hundred research papers . This type of static analysis has been used in a variety of applicationssuch as live variable analysis for register allocation and constant propagation, checking for potential runtime errors (e.g., null pointer dereferencing), static schedulers that need to track resource allocation and usage, etc. Despite its applicability in several other areas, however, pointer analysis has been targeted primarily at compilation, be it software or hardware . In particular, the use of pointer analysis (and in fact, static analysis in general) for automated source code transformations remains little explored. We believe the main reason for this is the different program representations employed in source-to-source tools. Historically, pointer analysis algorithms have been implemented in optimizing compilers, which typically proceed by dismantling the program into increasingly lower-level representations that deliberately discard most of the original structure of the source code to simplify its analysis. By contrast, source-to-source techniques strive to preserve everything about the structure of the original source so that only minimal, necessary changes are made. As such, they typically manipulate abstract syntax trees that are little more than a structured interpretation of the original program text. Such trees are often manipulated directly through treeor term-rewriting systems such as Stratego .In this paper, we present an algorithm developed to perform pointer analysis directly on abstract syntax trees. We implemented our algorithm in a source-to-source tool called Proteus , which uses Stratego as a back-end, and find that it works well in practice.

II. EXISTING POINTER ANALYSIS TECHNIQUES

Many techniques have been proposed for pointer analysis of C programs . They differ mainly in how they group related alias information. Figure1 shows a C fragment and the points-to sets computed by four well-known flow-insensitive algorithms.

Arrows in the figure represent pointer relationships between the variables in the head and tail nodes: an arc from

a to *b* means that variable *a* points-to variable *b*, or may point-to that variable, depending on the specific algorithm

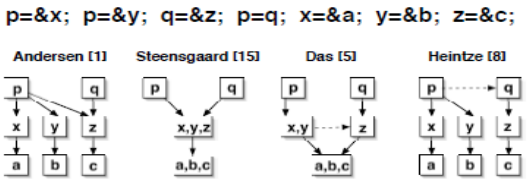
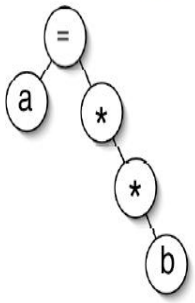


Figure 1. Results of various flow-insensitive pointer analysis algorithms.

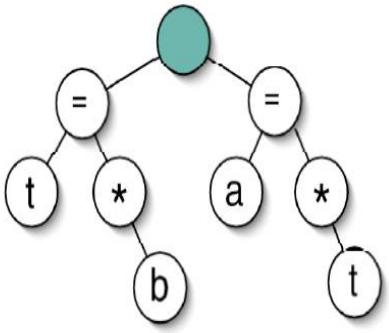
Some techniques encapusulate more than one variable in a single node, as seen in Steensgaard's and Das's approaches, in order to speed-up the computation. These methods trade precision for

running time: variable *x*, for instance, pointsto *a*, *b* and *c* on both techniques, although the code only assigns *a*'s address to *x*.

Broadly, existing techniques can be classified as constraint-solving or dataflow-based . Members of both groups usually define a minimal grammarfor the source language that includes only basic operators and statements. They then build templates used to match these statements. The templates are cast as inference Rules or dataflow equation. The algorithms consist of iterative applications of inference rulesor dataflow equations on the statements of the program, duringwhich pointer relationships are derived. This approach assumes that the C program only contains allowed statements. For instance, a=**b, with two levels of dereference in the right-hand side, is commonly parsed



Existing techniques generally require the preceding statement to be dismantled into two sub-expressions, each having at most one level of dereference.



It is difficult to employ such an approach to source-tosource transformations because it is difficult to correlate the results calculated on the dismantled program with the original source. Furthermore, it introduces needless intermediate variables, which can increase the analysis cost. For source-to-source transformations, we want to perform the analysis close to the source level. It is particularly useful to directly analyze the ASTs and annotate them with the results of the analysis. Hence, we need to be able to handle arbitrary compositions of statements.

Precision is another issue in source-to-source transformations: we want the most precise analysis practical because

otherwise we may make unnecessary changes to thecode or, even worse, make incorrect changes. A flowinsensitive analysis cannot, for example, determine that a pointer is initialized before it is used or that a pointer has different values in different regions of the program. Both of these properties depend on the order in which the statements of the program execute. As a result, the approach we adopt is flow-sensitive

2.1 Analysis Accuracy

Another source of approximation commonly found in today's approaches is the adoption of the so-called *non-visible*

variables , later renamed to *invisible variables* or, alternatively, *extended parameters* . When a function call takes place, a parameter1 *p* of pointer type might point to avariable *v* that is not in the scope of the called function. To

keep track of such pointer relationships, special symbolicnames are created in the enclosing scope and then manipulated in place of *v* whenever *p* is dereferenced. When the function call returns to the caller, the information

kept in the symbolic name is 'mapped' back to *v*. For example, for a variable *x* with type int**, symbolic names

1 x and 2 x with types int* and int would be created . If an indirect reference, say *x, can lead to an outof-scope variable *w*, the corresponding symbolic name 1 x is used to represent *w*.

There are some drawbacks with this approach: it adds an overhead in the analysis due to this 'mapping' and 'unmapping' of information, and it can become too approximateas the chain of function calls gets larger. The following example shows how spurious aliases can be generatedeven though symbolic variable 1 a is not accessed within the called function.

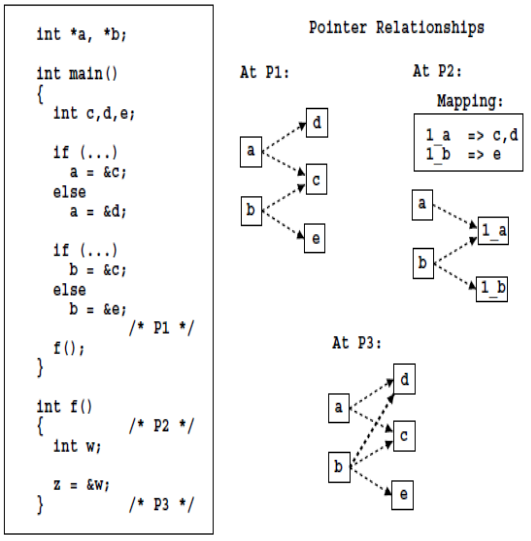
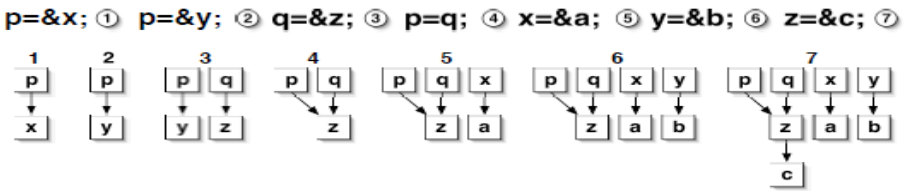


Figure 2. Innacuracy due to invisible variables.

The adoption of invisible variables, however, is of relevant importance if one's priority is the efficiency of the
interprocedural pointer analysis. The use of invisible variables facilitates summarization of the effects of a procedure
in the pointer relationships, and this enables the analysis to avoid re-evaluating a function's body in some particular
cases . On the other hand, invisible variables can cause some imprecision. We believe that pointer analysis for source-to-source code transformationshould be *information*-driven, i.e., precision of results should have a high priority. In this sense, we eliminatethe use of invisible variables at the expense of (potentiallly) having to re-evaluate a function's body multiple times.We rely on specially created 'signatures' in order to maintain pointer relationships across function calls, and handle the parameter passing mechanism as regular assignments.

III. ANALYSIS OUTLINE

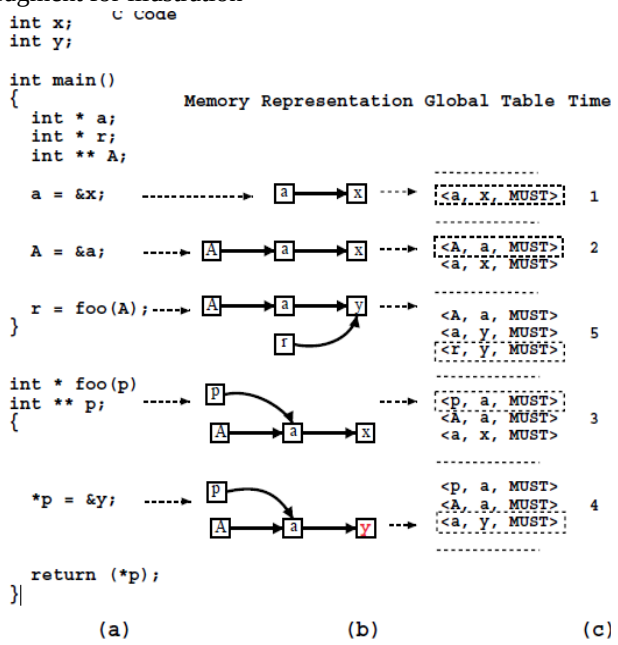
Following the approach of Emami et al. [6], our analysis uses an iterative dataflow approach that computes, for
each pointer statement, the points-to set generated (*gen*) and removed (*kill*) by the statement. The net effect of each statement is $(in - kill) \cup gen$, where *in* is the set of pointer relationships holding prior to the statement. In this sense, it is flow-sensitive and results in the following points-to sets for each sequence point in the code fragment of Figure 1.



By operating directly on the AST, we avoid building the control-flow graph for each procedure or the call-graph for
the whole program. Clearly, the control-flow graph can still be built if desired, since it simply adds an extra and relatively thin layer as a semantic attribution to the AST. Thus, from this specific point of view, ASTs are not a necessity for the iterative computation and handling of the program's control structure.
We assume the entire source code of the subject application (multiple translation units, multiple files) is resolved into a large AST that resides in memory , so that we are able to jump from one procedure to another through tree
queries. The analysis starts off at the program's *main* function, iterating through its statements. If a function call is
encountered, its body is recursively analyzed taking into account pointers being passed as parameters as well as global pointers. When the analysis reaches the end of the function, it continues at the statement following the function call. Below, we give an overview of some aspects of the implementation.

3.1 Points-to Graph Representation

We represent the points-to graph at a particular point in the program using a table. Entries in the table are triples
of the form _*x,y,q*_, where *x* is the source location pointing to *y*, the destination location, and *q* is the qualifier, which can be either *must* or *may*, which indicates that either *x* is definitely pointing to *y*, or that *x* merely may point to *y* (e.g., it may point to something else or be uninitialized). Pointer relations between variables in distinct scopes are encoded as regular entries in the table by relying on unique signatures for program variables. Below is a C fragment for illustration



On the left is the source code for two procedures; in the center are the memory contents during the analysis; and on
the right are the points-to sets generated by each statement. Note that each location of interest is represented by an abstract signature and that each pointer relationship holding between two locations is represented by an entry in the table. For an *if* statement, our algorithm makes two copies of the table, analyzes the statements in the true and false branches separately, then merges the resulting tables. The merge operation is a special union wherein a *must* triple has its qualifier demoted to *may* in case only one of the branches generates (or fails to kill) the triple. *For* and *while* statements are handled with a fixed-point computation—a copy of the table is made, the statements are analyzed, and the resulting table is compared to the initial one. The process is repeated until the two tables are the same.