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 source-to-source translation framework and experimental results show that it is practical on real-world examples.

Introduction
The role of pointer analysis in understanding C programs has been studied for years, being the subject of several PhD theses and nearly a hundred research papers [9]. This type of static analysis has been used in a variety of applications such 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 [9] or hardware [13]. 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 tree or term-rewriting systems such as Stratego [15, 16]. 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 [17], which uses Stratego [15] as a back-end, and find that it works well in practice.

Existing Pointer Analysis Techniques
Many techniques have been proposed for pointer analysis of C programs [1, 3, 4, 6, 10, 12, 14, 18]. They differ mainly in how they group related alias information. Figure 1 shows a C fragment and the points-to sets computed by four well-known flow-insensitive algorithms. Some techniques encapsulate 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, points to 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 [5, 7, 8] or dataflow-based [6, 11, 12, 18]. Members of both groups usually define a minimal grammar for 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 [5, 7, 8] or dataflow equations [6, 11, 12, 18]. The algorithms consist of iterative applications of inference rules or dataflow equations on the statements of the program, during which pointer relationships are derived. This approach assumes that the C program only contains allowed statements. For instance, a=**b, with two levels of dereferencing in the right-hand side, is commonly parsed. It is difficult to employ such an approach to source-to-source 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 the code or, even worse, make incorrect changes. A flow insensitive 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.

Analysis Accuracy

Another source of approximation commonly found in today’s approaches is the adoption of the so-called non-visible variables [10], later renamed to invisible variables [6] or, alternatively, extended parameters [18]. When a function call takes place, a parameter 1 p of pointer type might point to a variable v that is not in the scope of the called function. To keep track of such pointer relationships, special symbolic names are created in the enclosing scope [6] [10] [18] 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 [6] [18]. If an indirect reference, say *x, can lead to an out-of-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 approximate as the chain of function calls gets larger. The following example shows how spurious aliases can be generated even though symbolic variable 1 a is not accessed within the called function. In the example, 1 a stands for more than one program variable, namely c and d, due to the double assignment to global variable a [6] [18]. When the statement b = &c is analyzed, local variable c has already been mapped to 1 a, thus b is assumed to possibly point to such symbolic variable. 1 b is then created to represent program variable e, and the assignment b = &e induces pointer b to be also associated with 1 b. When f returns, an additional relationship between program variables b and d has been created, even though only local variables were accessed within f . A pointer relationship to 1 a was generated because of a relationship to one of the variables 1 a stands for - not all the variables it represents3. 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 [18]. On the other hand, invisible variables can cause some imprecision, as illustrated in Figure 2. We believe that pointer analysis for source-to-source code transformation should be information-driven, i.e., precision of results should have a high priority. In this sense, we eliminate the use of invisible variables at the expense of (potentially) 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.

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)Ugen, 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. 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 [17], 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, iteratively 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.

Basic Dataflow Framework

In our approach, the dataflow equations are not taken from a set of templates, as is usually done, but are evaluated while traversing the AST of the program. In the figures that follow, we express a must relationship as a solid line, and a may relationship as a dotted line. In this sense, assume that the pointer relationships holding between some variables just before analyzing the statement **x=y Assuming both z and w are (uninitialized) pointers, which makes x of *** type, this pointer assignment generates four new triples: z,q,may, z,r,may, w,q,may, and w,r,may . An invariant in the points-to graph is that any node can have at most one outgoing must edge (it would be nonsensical to say that a pointer “must” be pointing to two or more locations at the same time). It then follows from the definition of the gen set in Figure 4 that mustT(x,a)AmustT(y,b) ⇒ |gen(e,T)| = 1. That is, when both pointer chains are each known to point to exactly one thing (i.e., a and b), exactly one new relationship is generated. In the example above, n = 2, m = 0, Xn(T) = {z,w},
Y_{m+1}(T) = \{q,r\}, \neg \text{must}_T(x,z), \neg \text{must}_T(x,w), \neg \text{must}_T(y,q) \text{ and } \neg \text{must}_T(y,r).\text{ If instead we had the assignment } ^*x=y, \text{ then } n = 1, X_n(T) = \{u\}, \text{ must}_T(x,u), \text{ triples } u,z,\text{ may and } u,w,\text{ may are killed, and triples } u,q,\text{ may and } u,r,\text{ may are generated. Since the locations found after } m + 1 \text{ dereferences from } y \text{ are being assigned to the locations found after } n \text{ dereferences from } x, \text{ the gen set is formed by the cross product of sets } X_n(T) \text{ and } Y_{m+1}(T). \text{ Each resulting triple } a,b,l \text{ has } l = \text{ must} \text{ only when } \text{must}_T(x,a) \text{ and } \text{must}_T(y,b) \text{ hold (i.e., when all the relationships along both simple paths are known exactly), and has } l = \text{ may otherwise. In the kill set computation, } \text{must}_T(x,a) \text{ requires } X_n(T) = \{a\} \text{ (e.g., the set } \{u\} \text{ in the assignment } ^*x=y). \text{ Location } a \text{ is guaranteed to be changed, so we remove the relations where a points to a variable from points-to information. So the kill set includes relationships about everything that a may or must point to prior to the assignment. If } \text{must}_T(x,a) \text{ does not hold, then existing triples } a,b,l \text{ cannot be removed, since the modification of } a \text{ is not guaranteed (i.e., a may not be reached when the assignment is executed).}

Conclusions and Future work

The main contribution of this paper is a pointer analysis algorithm that operates on the abstract syntax tree of a program—a necessity for source-to-source transformations, which strive to preserve as much about the program as possible. Our algorithm performs a flow-sensitive analysis using dataflow equations generated directly on-the-fly from the abstract syntax tree of the program. Our choice of a flow-sensitive analysis makes our algorithm slower than many existing techniques, but the extra precision it provides is useful in source-to-source transformations. Similarly, our choice of re-analyzing a function each time it is called is less efficient than techniques that, say, create a transfer function for each subroutine and re-apply it as necessary [18], but this increases precision. The algorithm presented in this paper fits the environment typical in source-to-source tools, although some coding optimizations are still needed to make it run faster. In the future, we plan to memoize functions that do not change the points-to sets, which should not affect precision. We also plan to build a visualization tool that displays the points-to sets graphically (presumably as a points-to graph). This might be useful for source code debugging. Partial support for this has already been built. Implementation This section presents some code that was extracted verbatim from our current implementation. The language we used was YATL— a brief explanation about its syntax and semantics was given in Section 9.

References

