Pointer Analysis for Source-to-Source Transformations

Marcio Buss[*] Stephen A. Edwards[†]
Department of Computer Science
Columbia University
New York, NY 10027
{marcio,sedwards}@cs.columbia.edu

Bin Yao Daniel Waddington
Network Platforms Research Group
Bell Laboratories, Lucent Technologies
Holmdel, NJ 07733
{byao,dwaddington}@lucent.com

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.

1 Introduction

The role of pointer analysis in understanding C programs has been studied for years, being the subject of several PhD thesis and nearly a hundred research papers [10]. 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 [10] or hardware [14]. 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.

```
p=&x; p=&y; q=&z; p=q; x=&a; y=&b; z=&c;
```

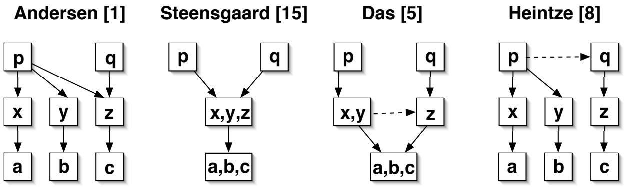


Figure 1. Results of various flow-insensitive pointer analysis algorithms.

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 [16, 17].

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 [18], which uses Stratego [16] as a back-end, and find that it works well in practice.

2 Existing Pointer Analysis Techniques

Many techniques have been proposed for pointer analysis of C programs [1, 3, 5, 7, 11, 13, 15, 19]. 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.

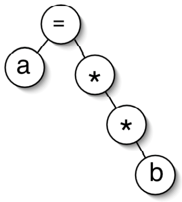
Arrows in the figure represent pointer relationships between the variables in the head and tail nodes: an arc from *a* to *b* means that variable *a* points-to variable *b*, or may

[*] supported in part by CNPq Brazilian Research Council, grant number 200346/01-6

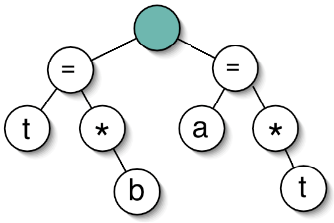
[†] supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and by New York State's NYSTAR program

point-to that variable, depending on the specific algorithm. 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, 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 [6, 8, 9] or dataflow-based [7, 12, 13, 19]. 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 [6, 8, 9] or dataflow equations [7, 12, 13, 19]. 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 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-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.

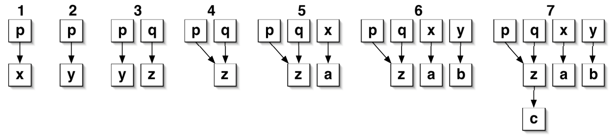
For source-to-source transformations, we want to perform the analysis close to the source level. It is particularly useful to directly analyze the ASTs and annotate them with the results of the analysis. Hence, we need to be able to handle arbitrary compositions of statements.

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.

3 Analysis Outline

Following the approach of Emami et al. [7], 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.

p=&x; ① p=&y; ② q=&z; ③ p=q; ④ x=&a; ⑤ y=&b; ⑥ z=&c; ⑦



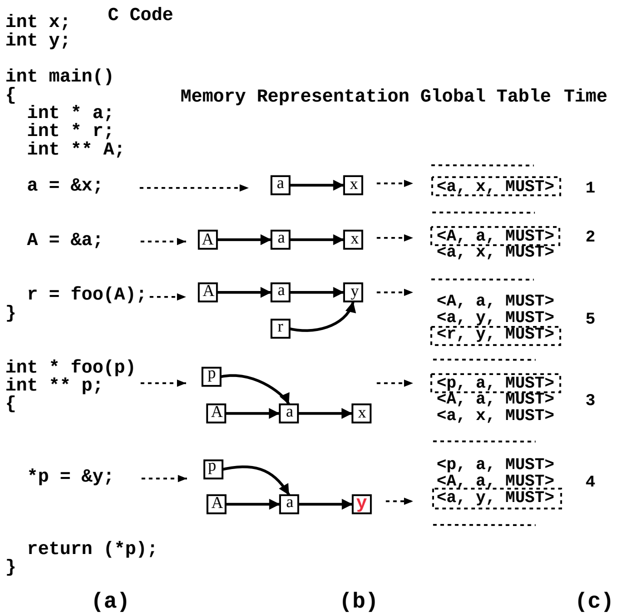
By operating directly on the AST, we avoid building the control-flow graph for each procedure or the call-graph for the whole program. Clearly, the control-flow graph can still be built if desired, since it simply adds an extra and relatively thin layer as a semantic attribution to the AST. Thus, from 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 [18], 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 $\langle x, y, q \rangle$, 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.



```
int x;
int y;

int main()
{
  int * a;
  int * r;
  int ** A;

  a = &x;
  A = &a;
  r = foo(A);
}

int * foo(p)
int ** p;
{

  *p = &y;

  return (*p);
}
```

```
<a, x, MUST>                                1

<A, a, MUST>
<a, x, MUST>                                2

<A, a, MUST>
<a, y, MUST>
<r, y, MUST>                                5

<p, a, MUST>
<A, a, MUST>
<a, x, MUST>                                3

<p, a, MUST>
<A, a, MUST>
<a, y, MUST>                                4
```

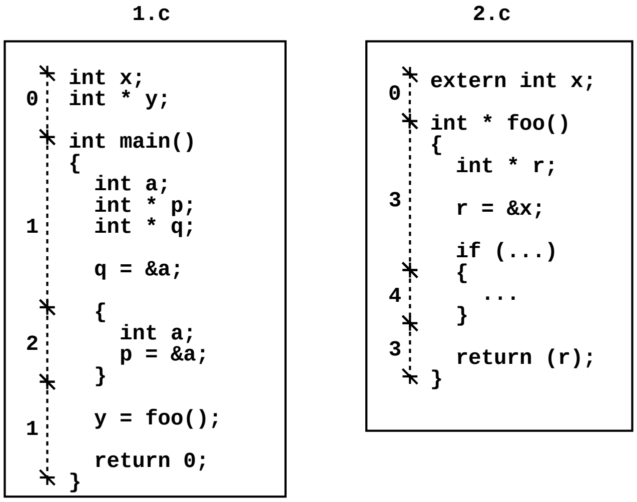
On the left is the source code for two procedures; in the center are the memory contents during the analysis; and on the right are the points-to sets generated by each statement. Note that each location of interest is represented by an abstract signature and that each pointer relationship holding between 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 (denoted by ⋓ in Figure 6) 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.

3.2 Abstract Signatures

Each location of interest in the program is represented by a unique signature of the form

$$(\mathit{function\text{-}name}, \mathit{identifier}, \mathit{scope})$$

where *function-name* is the name of the function in which the variable or parameter is declared or a special keyword for global variables; *identifier* is the syntactic name given by the programmer or specially-created names for heap locations; and *scope* is a unique integer assigned to each distinct scope in the program (the scope associated with a given signature is the integer assigned to the scope where the variable is declared). The numbers to the left of each source program below show a possible set of scopes. The dashed lines delimit their ranges.



```
  int x;
0 int * y;

  int main()
  {
    int a;
    int * p;
1   int * q;

    q = &a;

    {
      int a;
2     p = &a;
    }

1   y = foo();

    return 0;
  }
```

```
  extern int x;
0
  int * foo()
  {
    int * r;

3   r = &x;

    if (...)
    {
4     ...
    }
3
    return (r);
  }
```

The signatures created for q and a while analyzing the statement q=&a are (main,q,1) and (main,a,1). The signatures for p and a in the statement p=&a, are $(\mathrm{main}, \mathrm{p}, 1)$ and $(\mathrm{main}, \mathrm{a}, 2)$ (a is redeclared in scope 2). Signatures are generated on-the-fly to avoid pre-processing.

3.3 Pointer Relationships Representation

Once everything has a unique signature, we adopt the relations *must* and *may* points-to as follows.

By definition, variable x *must* point to variable y at program point p if, at that program point, the address of y is in the set S of possible locations that x may point to and $|S| = 1$. Also, all possible execution paths to program point p must have assigned y's address to x prior to p, and that address assignment must not have been killed since then. This is denoted by the triple $\langle x', y', must\rangle$, where x' and y' represent the abstract signatures for x and y.

Similarly, variable x *may* point to variable y at program point p if, at that program point, the address of y is in the set S of possible locations that x may point to and either $|S| > 1$ or there exists some execution path P_i to p that does not assign y's address to x. This is denoted by the triple $\langle x', y', may\rangle$, where x' and y' are the signatures for x and y.

Intuitively, an assignment $x = \&y$ at point p inside the *then* branch of an *if* statement implies that x must point to y from p to the point where both execution paths merge, assuming x is not redefined in between; x may point to y after this in case the path that goes through the *else* part does not assign y's address to x.

Figure 2 shows a code fragment and snapshots of the entire table at four distinct moments during the analysis (for clarity, *must* is written "M" and *may* is written "m"). Point 1, for example, corresponds to the instant after the analysis has traversed the *if* statement at lines 10–13, the assignment at line 15, the call site at line 16, and is about to analyze foo.

C Code

```
 1 int y;
 2 int *z,*w;
 3
 4 int main()
 5 {
 6   int *a,*r;
 7   int ** A;
 8   int *** B;
 9
10   if (...)
11     A = &a;
12   else
13     A = &w;
14
15   B = &A;
16   r = foo(B); /* 4 */
17 }
18
19 int * foo(int *** p1)
20 {
21   int k;      /* 1 */
22   **p1 = &y;  /* 2 */
23   z = &k;     /* 3 */
24   return (z);
25 }
```

Global Table

```
Point 1:
<("main","A",1),("main","a",1),m>
<("main","A",1),(Global,"w",0),m>
<("main","B",1),("main","A",1),M>
<("foo","p1",2),("main","A",1),M>

Point 2:
<("main","A",1),("main","a",1),m>
<("main","A",1),(Global,"w",0),m>
<("main","B",1),("main","A",1),M>
<("foo","p1",2),("main","A",1),M>
<("main","a",1),(Global,"y",0),m>
<(Global,"w",0),(Global,"y",0),m>

Point 3:
<("main","A",1),("main","a",1),m>
<("main","A",1),(Global,"w",0),m>
<("main","B",1),("main","A",1),M>
<("foo","p1",2),("main","A",1),M>
<("main","a",1),(Global,"y",0),m>
<(Global,"w",0),(Global,"y",0),m>
<(Global,"z",0),("foo","k",2), M>

Point 4:
<("main","A",1),("main","a",1),m>
<("main","A",1),(Global,"w",0),m>
<("main","B",1),("main","A",1),M>
<("main","a",1),(Global,"y",0),m>
<(Global,"w",0),(Global,"y",0),m>
<("main","r",1),("foo","k",2), M>
```

Figure 2. Example program.

Starting at the body of the main function, the *if* statement at lines 10–13 assigns the addresses of local variable `a` and global variable `w` to `A`. According to the definition of *may*, `A` may point to either location after the statement, and this is represented by the first two entries in the table for point 1 (in fact, since these pointer relationships are not killed anywhere in the program, they will persist throughout the entire analysis). The other two entries at point 1 come from the assignment of `&A` to `B` in line 15, and the function call at line 16 (point 4 at line 16 happens after `foo` returns). Specifically, the parameter passing in `r = foo(B)` makes `p1` point to whatever locations `B` points to, namely `A`.

At point 2, `p1` is dereferenced twice. The first dereference leads to `A` and the second dereference leads to either `a` or `w`. Accordingly, both locations are marked as “may point to `y`.” Two new entries are created at point 2 (highlighted in the figure), indicating that both `a` and `w` may point to `y`.

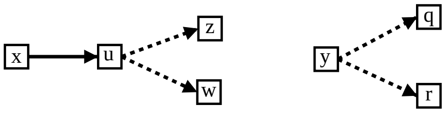
Note that both `A` and `a` (but not `w`) fall out of scope when `foo` is called, although they can be indirectly accessed through `p1`. Existing techniques create a set of “invisible” variables, or extended parameters [7, 19], in which symbolic names are used to access out-of-scope variables reached through dereferences of a local pointer. We handle distinct scopes more transparently, as seen by the effects of the statement `**p1=&y`. Furthermore, avoiding invisible variables may increase the accuracy of the analysis results, especially on a chain of function calls, since a single symbolic name may end up representing more than one out-of-scope variable in some cases [7, 19].

The statement at line 23, `z=&k`, adds a new triple to point 3 (highlighted), and the *return* at line 24 causes `r` to refer to where `z` points. But note that prior to the *return*, `z` points to a local variable of the called function, and this causes `r` to refer to an invalid location. By using our naming scheme, the highlighted triple in point 4 reveals the violation. In the analysis, we can use the name of the closing function to detect such invalid triples. This potential bug was not detected by *lint* or Gimpel’s *FlexeLint*.

During the analysis, the same idea is used each time a scope closes (using the scope information in the signatures) to perform a limited type of escape analysis [2], or to delete certain triples. The latter is seen at point 4, where $\langle(\mathrm{foo},\mathrm{p1},2),(\mathrm{main},\mathrm{A},1),\mathrm{M}\rangle$ was deleted since `p1` would be removed from the stack at runtime upon function return.

4 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` are as follows:



Assuming both `z` and `w` are (uninitialized) pointers, which makes `x` of `***` type, this pointer assignment generates four new triples: $\langle z,q,may\rangle$, $\langle z,r,may\rangle$, $\langle w,q,may\rangle$, and $\langle w,r,may\rangle$. The resulting relationships are

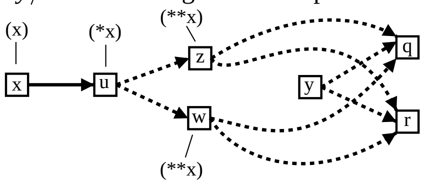


Figure 3 shows the formal definition of the dataflow equations for an assignment. Here, $X_n(T)$ is the set of locations reached after n dereferences from x in T, the table, $Y_{m+1}(T)$ is the set of locations reached after $m+1$ dereferences from y in T, and the predicate $\mathrm{must}_T(v_1,v_2)$ is true only when all the relationships along the path from v_1 to v_2 in T are *must*.

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 3 that

$$\mathrm{must}_T(x,a)\wedge\mathrm{must}_T(y,b)\Rightarrow|\mathrm{gen}(e,T)|=1.$$

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.

For an assignment e of the form

$$\underbrace{*\cdots*}_{n}x=\underbrace{*\cdots*}_{m}y$$

$$\begin{aligned}
\text{gen}(e,T) &= \left\{\langle a,b,l\rangle : a\in X_n(T)\wedge b\in Y_{m+1}(T)\wedge l=\begin{cases} \textit{must} & \text{if } \text{must}_T(x,a)\wedge\text{must}_T(y,b)\\ \textit{may} & \text{otherwise}\end{cases}\right\}\\
\text{change}(e,T) &= \left\{\langle a,b,l\rangle : a\in X_n(T)\wedge\langle a,b,l'\rangle\in T\wedge l=\begin{cases} \textit{must} & \text{if } \text{must}_T(x,a)\\ \textit{may} & \text{otherwise}\end{cases}\right\}\\
\text{kill}(e,T) &= \big\{\langle a,b,l\rangle : a\in X_n(T)\wedge\langle a,b,l\rangle\in T\wedge\text{must}_T(x,a)\big\}\\
T' &= \big(T-\{\langle a,b,\text{must}\rangle : \langle a,b,\text{may}\rangle\in\text{change}(e,T)\}\big)\cup\text{change}(e,T)\\
T'' &= \big(T'-\text{kill}(e,T)\big)\cup\text{gen}(e,T)
\end{aligned}$$

Figure 3. Dataflow equations for an assignment.

In the example above, $n=2$, $m=0$, $X_n(T)=\{z,w\}$, $Y_{m+1}(T)=\{q,r\}$, $\neg\text{must}_T(x,z)$, $\neg\text{must}_T(x,w)$, $\neg\text{must}_T(y,q)$ and $\neg\text{must}_T(y,r)$. If instead we had the assignment `*x=y`, then $n=1$, $X_n(T)=\{u\}$, $\text{must}_T(x,u)$, triples $\langle u,z,may\rangle$ and $\langle u,w,may\rangle$ are killed, and triples $\langle u,q,may\rangle$ and $\langle u,r,may\rangle$ are generated.

Since the locations found after $m+1$ dereferences from y are being assigned to the locations found after n dereferences from x, the *gen* set is formed by the cross product of sets $X_n(T)$ and $Y_{m+1}(T)$. Each resulting triple $\langle a,b,l\rangle$ has $l=must$ only when $\text{must}_T(x,a)$ and $\text{must}_T(y,b)$ hold (i.e., when all the relationships along both simple paths are known exactly), and has $l=may$ otherwise.

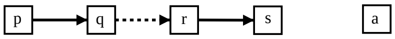
In the *kill* set computation, $\text{must}_T(x,a)$ requires $X_n(T)=\{a\}$ (e.g., the set $\{u\}$ in the assignment `*x=y`). Location a is guaranteed to be changed, so we remove the relations where a points to a variable from points-to information. So the *kill* set includes relationships about everything that a may or must point to prior to the assignment. If $\text{must}_T(x,a)$ does not hold, then existing triples $\langle a,b,l\rangle$ cannot be removed, since the modification of a is not guaranteed (i.e., a may not be reached when the assignment is executed).

The *change* set contains relationships that must be demoted from *must* to *may*. Section 5 demonstrates this with an example.

The definitions in Figure 3 apply for any number of dereferences in an assignment, and we extend this basic idea in our analysis for compositions of C statements. We calculate such *gen*, *kill*, and *change* sets using a recursive traversal of the abstract syntax tree of the program (we describe an example in the next section). The dataflow equations match the semantics of pointer dereferences in C, and the treatment of related operators such as address-of and field-dereference (e.g., `p->q`) follows a similar rationale.

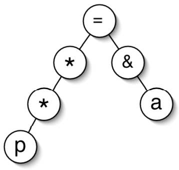
5 An Example

Consider the statement `**p=&a`, where `a` is a non-pointer variable, and assume that pointers `p`, `q`, `r`, and `s` have the following relationship:

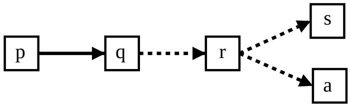


```
Global Table
<p,q,MUST>
<q,r,MAY>
<r,s,MUST>
```

The AST of this assignment is

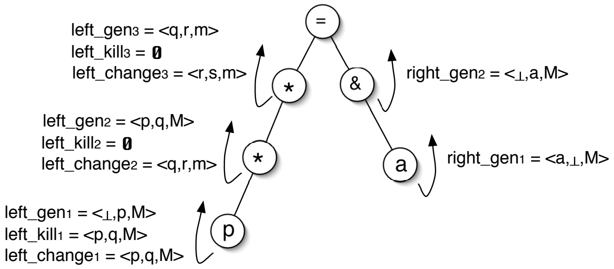


This assignment adds the triple $\langle r,a,may\rangle$ and changes the triple $\langle r,s,must\rangle$ to $\langle r,s,may\rangle$. The point-to relationships after the assignment are



```
Global Table
<p,q,MUST>
<q,r,MAY>
<r,s,MAY>
<r,a,MAY>
```

To determine this, our algorithm independently traverses the left and right sides of the assignment, collecting information on the way. This process is shown below:



We construct the *gen* set by combining information from the sets labeled left_gen and right_gen, collected from both the left and right sides of the assignment. By contrast, the *kill* and *change* sets are computed from the left side of the assignment only —from the left_kill and left_change sets— because an existing points-to relationship can only be affected through assignment (i.e., by the lvalue).

The traversal on the left side of the AST starts at the first `*` node and goes down recursively until reaching the identifier `p`. The figure above shows the three sets returned at this point. A table query is performed to compute left_kill$_1$ and left_change$_1$.

For the next node up as the recursion unwinds, the table is accessed and the returned sets correspond to the locations pointed to by the expression `*p`. Note that the *may* relation between `q` and `r` leaves left_kill$_2$ empty. The topmost dereference is then reached and the final sets left_gen$_3$, left_kill$_3$ and left_change$_3$ represent the sets for `**p`. Note that left_change$_3$ contains triple $\langle r, s, may\rangle$ although the current relationship between r and s is *must*. This is because a *may* relation was crossed on the way up the recursion—a `*` node in the AST correspond to a "qualified" dereference that takes into account qualifiers already seen.

Similarly, the traversal on the right starts at the `&` node and stops at the identifier `a`. The base case on the right is slightly different than on the left. An identifier on the right is an rvalue, and a lookup in the table does the dereference. Since `a` is a non-pointer variable, we assume it points to an undefined location (expressed as $\perp$ in right_gen$_1$). The address-of operator results in right_gen$_2$.

The final *gen* set is obtained by merging left_gen$_3$ and right_gen$_2$. Given a triple $\langle x, y, f\rangle$ in left_gen and $\langle z, w, g\rangle$ in right_gen, the *gen* set for the assignment includes the triple $\langle y, w, f \wedge g\rangle$ (i.e., the relationship is *must* only if both the left and right sets were *must*, otherwise it is *may*). In the example, this triple is $\langle r, a, may\rangle$.

The triple $\langle r, s, must\rangle$ is changed to $\langle r, s, may\rangle$. It would just have been killed were it not for the double dereference from `p` crossing a *may* relation. This fact is captured in left_change$_3$, which contains $\langle r, s, may\rangle$. This implies $\langle r, s, may\rangle$ should replace $\langle r, s, must\rangle$, since it is not guaranteed that r will be left unchanged by the assignment `**p=&a`. At the end of the recursion, we compare what we have computed for left_change with what actually holds in the table, and update T where they disagree.

6 The Algorithm

This section presents our algorithm. Additional details can be found in our technical report [4].

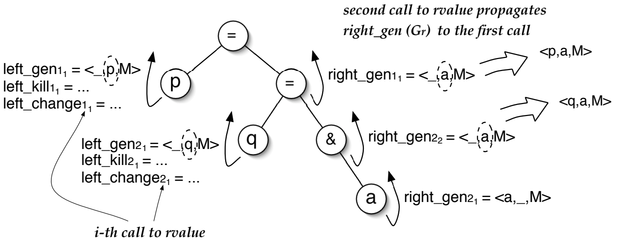
6.1 Expressions, Function Calls, and Assignments

The function in Figure 4 calculates the *gen* set for an expression (an rvalue of an assignment), and Figure 5 calculates the *gen*, *change*, and *kill* sets for the lvalue of an assignment. Together, they handle C's expressions.

Both functions take as parameters *e*, the sub-expression being analyzed (i.e., a node in the AST), and T, the current table. Both proceed by recursing on the structure of the expression, with separate rules for pointer dereferencing, function calls, and so forth.

In Figure 4, the rule for an assignment expression is fairly complicated. It first calls the lvalue function in Figure 5 to build the *gen*, *change*, and *kill* sets for the left-hand side of the assignment, calls itself recursively to calculate the *gen* set for the right-hand side, then merges the results of these two calls and uses them to update the table T.

Note that these functions handle nested assignments. Consider the expression `p=q=&a`. The rvalue function identifies the assignment to `p` and calls itself recursively on the assignment `q=&a`. In addition to updating the table with the effects of this expression, the G_r set is returned to the outer call of rvalue. Ultimately, $\langle p, a, must\rangle$ and $\langle q, a, must\rangle$ are added. This recursive behavior is shown below.



6.2 Pointer Dereferencing

Both Figures 4 and 5 use a function—dereference—that performs (qualified) pointer dereference by querying the table T and combining qualifiers already seen. It takes a set of triples, S, and returns a set of triples that is the union of all locations pointed-to by elements in S. Precisely,

$$\begin{aligned}&\text{dereference}(S, T) = \\ &\quad \{\langle y, z, f \wedge g\rangle : \langle x, y, f\rangle \in S \;\wedge\; \langle y, z, g\rangle \in T\},\end{aligned}$$

where $f \wedge g = must$ if $f = must$ and $g = must$, otherwise $f \wedge g = may$.

This definition for dereference combines the qualifiers f and g to comply with the behavior of the must$_T$ predicate in the dataflow equations (Figure 3). The idea behind dereference is to incrementally follow paths in the points-to graph induced by the expression while propagating the "intersection" of the qualifiers.

6.3 Address-of Operator

The address_of function returns a set of triples that correspond to every variable that points to something in a set of triples. Precisely,

$$\text{address\_of}(S) = \{\langle \perp, x, l\rangle : \langle x, y, l\rangle \in S\}$$

7 Interprocedural Analysis

The parameter passing mechanism—the rule for functions in Figure 4—behaves initially like a sequence of simple assignments. For example, our algorithm treats the call

```
foo(a1, a2, a3); /* call */
foo(f1, f2, f3) {  /* definition */ }
```

```
function rvalue($(e,T)$) returns (table, gen)
  case $e$ of
  $\&e_1$ :                                              Address-of
    $(T,G) = \text{rvalue}(e_1,T)$
    return $(T, \text{address\_of}(G))$

  $*e_1$ :                                               Dereference
    $(T,G) = \text{rvalue}(e_1,T)$
    return $(T, \text{dereference}(G,T))$

  $v$ :                                                  Identifier
    if $v$ is a global variable then
      $s = (\text{\_Global}, v, 0)$                      signature for global
    else
      $f$ = name of the function where $v$ is declared
      $d$ = number of scope where $v$ is declared
      $s = (f, v, d)$                                    signature for local
    if there is at least one $\langle s,x,l \rangle \in T$ then
      return $(T, \{\langle s,x,l \rangle : \langle s,x,l \rangle \in T\})$
    else
      return $(T, \{\langle s, \perp, must \rangle\})$

  $f(a_1, a_2, \ldots)$ :                                Function call
    $d$ = outermost scope for the body of function $f$
    for each actual parameter $a_i$ do
      $(T,G) = \text{rvalue}(a_i,T)$
      $v_i$ = formal parameter for $a_i$
      $p_i = (f, v_i, d)$                                signature for the formal
      for each $\langle \_, q, l \rangle \in G$ do
        add $\langle p_i, q, l \rangle$ to $T$
    $T$ = statement(body of $f, T$)
    Remove local variables declared in $f$ from $T$
    return $(T, G_r)$                  $G_r$ is computed at return stmts

  $l = r$ :                                              Assignment
    $(T, G_l, C_l, K_l) = \text{lvalue}(l,T)$
    $(T, G_r) = \text{rvalue}(r,T)$
    $G = \emptyset$
    $K = K_l$
    for each triple $\langle x,y,l_1 \rangle \in G_l$ do
      for each triple $\langle z,w,l_2 \rangle \in G_r$ do
        if $l_1 = must \wedge l_2 = must$ then
          Add $\langle y,w,must \rangle$ to $G$
        else
          Add $\langle y,w,may \rangle$ to $G$
    for each triple $\langle x,y,f \rangle \in C_l$ do
      if $\langle x,y,must \rangle \in T \wedge f = may$ then
        Replace $\langle x,y,must \rangle$ with $\langle x,y,may \rangle$ in $T$
    $T = (T - K) \cup G$
    return $(T, G_r)$

  $e_1\, op\, e_2$ :                                     Arithmetic operators
    $(T,G) = \text{rvalue}(e_1,T)$
    $(T,G) = \text{rvalue}(e_2,T)$
    return $(T,G)$
```

Figure 4. The function for expressions.

```
function lvalue($(e,T)$) returns (table, gen, change, kill)
  case $e$ of
  $\&e_1$ :                                              Address-of
    $(T,G,C,K) = \text{lvalue}(e_1,T)$
    return $(T, \text{address\_of}(G), \text{address\_of}(C), \text{address\_of}(K))$

  $*e_1$ :                                               Dereference
    $(T,G,C,K) = \text{lvalue}(e_1,T)$
    Remove all triples like $\langle x,y,may \rangle$ from $K$
    return $(T, \text{dereference}(G), \text{dereference}(C), \text{dereference}(K))$

  $v$ :                                                  Identifier
    if $v$ is a global variable then
      $s = (\text{\_Global}, v, 0)$                      signature for global
    else
      $f$ = name of the function where $v$ is declared
      $d$ = number of scope where $v$ is declared
      $s = (f, v, d)$                                    signature for local
    $G = \{\langle \perp, s, must \rangle\}$
    if there is at least one $\langle s,x,l \rangle \in T$ then
      $C = K = \{\langle s,x,l \rangle : \langle s,x,l \rangle \in T\}$
    else
      $C = K = \{\langle s, \perp, must \rangle\}$
    return $(T,G,C,K)$
```

Figure 5. The function for lvalues.

as a series of assignments `f1=a1; f2=a2; f3=a3;`. Each assignment, which may have an arbitrary expression on the right, is treated like an assignment expression, although we only compute the *gen* set for each since formal parameters are guaranteed to be uninitialized before the call. The scopes of the actual expressions differ from those of the formal arguments; our rule for signatures ensures this.

Once the assignments are performed, the statements in the function body are analyzed and may produce an updated table since they might modify existing pointer relationships (e.g., Figure 2). Additionally, if the function itself returns a pointer, we collect the potential return values at the *return* statements and merge all of them as the G_r set for the function call.

Return statements are fairly subtle. To process a return statement, our algorithm collects return values in case the function returns a pointer and merges the points-to information reaching the *return* statement with the points-to information reaching other *return* statements in the function. At the end of the function, the set from the *return* statements is merged with the points-to information reaching the end of the function to combine all potential outputs. Our technical report [4] describes this in more detail.

Our current implementation handles recursive functions in a simplistic way. Basically, we keep a stack data structure that resembles the function call stack, containing the name

```
function statement($(s, T)$) returns table
  case $s$ of
  an expression $e$ :                                   Expression
    $(T, \_) = \text{rvalue}(e, T)$

  $s_1; s_2; s_3; \ldots$ :                             Compound statement
    for each statement $s_i$ do
      $T = \text{statement}(s_i, T)$

  if ($e$) $s_1$ else $s_2$ :                           If-else statement
    $(T, \_) = \text{rvalue}(e, T)$
    $T_1 = T$
    $T_1 = \text{statement}(s_1, T_1)$
    $T_2 = T$
    $T_2 = \text{statement}(s_2, T_2)$
    $T = T_1 \Cup T_2$

  while ($e$) $s_1$ :                                   While statement
    $(T, \_) = \text{rvalue}(e, T)$
    $T' = T$
    $T'' = \emptyset$
    while $T' \neq T''$ do
      $T'' = T'$
      $T' = \text{statement}(s_1, T')$
    $T = T \Cup T'$

  for ($i$ ; $c$ ; $n$) $s_1$ :                         For statement
    $(T, \_) = \text{rvalue}(i, T)$
    $(T, \_) = \text{rvalue}(c, T)$
    $T' = T$
    $T'' = \emptyset$
    while $T' \neq T''$ do
      $T'' = T'$
      $T' = \text{statement}(s_1, T')$
      $(T', \_) = \text{rvalue}(n, T')$
    $T = T \Cup T'$

  return $T$
```

Figure 6. The function for statements.

of the functions being analyzed in the current chain of calls. At every new call site, we check if the called function's name is in the stack. If so, we skip the function call and continue to the next statement. If not, we add the function's name to the top of the stack and jump to its first statement to continue the analysis. This method is clearly not very precise, but is a reasonable initial trade-off. We plan to extend the recursive function handling in a future implementation of our algorithm by performing a fixed-point computation. Function pointers are also handled by our algorithm. Since we perform a flow-sensitive, context-sensitive, interprocedural points-to analysis, the set of functions invocable from a function pointer call-site is a subset of the set of functions that the function pointer can point to at the program point just before the call-site. The analysis assumes that all these functions are invocable from the site, and merges their output sets to compute the points-to information at the program point after this call. As previously mentioned, this requires a flow-sensitive analysis due to its dependence on statement ordering.

8 Experimental Results

We have implemented the algorithm presented in this paper (along with additions for handling the rest of C) in a Linux-based source-to-source framework called Proteus [18]. Proteus uses Stratego [16] as its back end and thus employs tree-rewriting for code transformations. To write transformations in Proteus, the user writes a program in the YATL language, which is compiled to an Stratego file. Thus, we used a transformation language to implement our pointer analysis algorithm. One can view it as an "annotation" transformation that traverses the ASTs of the subject program, analyzing pointer statements without actually rewriting the code.

As an example, the following YATL fragment, taken verbatim from our implementation, checks if the term being analyzed is an *if* statement and, if so, analyzes both branches of the conditional and merges the results.

```
match(IfElseStmt: {=$cnd}<cond>,{=$th}<then>,{=$el}<else>)
{
  // Analyze the condition expression //
  analyze_expression($cnd, $t);

  // Create two copies  of  the  current //
  // points-to set, t, and hand them  to //
  // the two branches of the  "if"  stmt //

  $thenSet = int-to-string(uuid-int());
  set_copy($thenSet, $t);

  $elseSet = int-to-string(uuid-int());
  set_copy($elseSet, $t);

  analyze_generic_stmt($th, $thenSet);
  analyze_generic_stmt($el, $elseSet);

  // Merge "thenSet" and "elseSet" //
  set_merge($thenSet, $elseSet);
  set_copy($t, $thenSet);

  // Free unused memory //
  set_destroy($thenSet);
  set_destroy($elseSet);
}
```

The *match* construct in the above code means if the term being analyzed—the root of the current subtree—is an *if* statement, to bind the subtree representing the conditional expression to variable `$cnd`, the subtree corresponding to the true branch to variable `$th`, and the subtree corresponding to the else branch to `$el`. Since `$cnd` can be an arbitrary expression, it can include a pointer assignment (`if ((p=malloc(...))!=NULL)` is typical). The call to analyze_expression (rvalue function) han-

name	lines of code	number of files	parsing time	analysis time	max. memory
stanford	885	1	17s	48s	16Mb
compress	1933	3	27s	< 1m	27
mpeg2dec	9830	20	< 2m	< 7m	24
jpeg	27966	85	< 7m	< 32m	65

Table 1. Experimental results.

dles the conditional, which might update `$t`, a string that holds a unique name for the table: a “pointer” to it.

Two copies of the table are made—`$thenSet` and `$elseSet`. These are two unique names generated by `uuid-int` and converted to strings by `int-to-string`. The statements in the two branches of the *if* are then analyzed, each branch with its own copy of the initial table `$t`. After this is done, the resulting tables are merged by `set_merge`, ($\Cup$ in Figure 6) and the final set overwrites `$t` (the first parameter in `set_merge` also represents the destination; `set_copy(a,b)` means $a \leftarrow b$). Finally, the memory used for the temporary sets is freed.

With the support from the tool to build ASTs, resolve multiple files, and provide the front-end language, the pointer analysis algorithm takes less than four thousand lines of code, yet covers almost the entire C language (we currently do not handle *goto* statements).

8.1 Experiments

We tested our procedure on a set of benchmarks ranging in size from about 800 to 30 000 lines of code (including whitespace and comments). We report four test cases: stanford, compress, mpeg2dec, and jpeg. Stanford is a collection of algorithms such as a solution to the eight-queens problem and Towers of Hanoi. Compress, mpeg2dec, and jpeg are well-known file compression, MPEG video decoder, and JPEG encoder/decoder libraries. We slightly modified the source of each example to remove *goto* statements (we duplicated code) and correct prototypes.

To analyze a program, our system first parses all its source files and constructs a single AST in memory. We list the time taken for this in the *parsing* column of Table 1. Then our analysis runs: traverses the AST starting from *main*, constructs tables, etc. The times for this phase are listed under *analysis*. We ran these experiments on a 512 Mb, 2.4GHz Pentium 4 running Linux.

Not surprisingly, the time required for our analysis grows with the size of the program, as the price for precision in the form of flow-sensitiveness and function body re-analysis is paid in efficiency. Thirty-two minutes of analysis time for the largest example may seem excessive, but our objective has been precision, not speed, and as such we have not attempted to make our implemention more efficient. Compared to traditional pointer analysis algorithms, ours is flow-sensitive and interprocedural, up to multiple translation units and multiple files.

We believe that for source-to-source transformations, however, this magnitude of execution time is acceptable. This type of static analysis could automate a source code transformation that would take days or weeks to perform manually. For instance, inserting the minimal amount of null pointer checking in the source code might be done by first performing a pointer analysis and then inserting checks wherever a pointer may be null. Although slower, flow-sensitivity is of paramount importance to this type of checking, since verifying whether a pointer is initialized before it is used depends on the order of the statements. We are currently applying our analysis to porting a legacy application that assumed a big-endian architecture to a little-endian architecture. To perform this, we augment the points-to sets with type information—a simple, but very useful modification.

In Table 1, we list memory usage, which includes the space needed to store ASTs, symbol table(s), as well as the space used for temporary points-to tables. Although the source of the compress example is smaller, it requires about as much memory as the larger mpeg2dec example because the code is more pointer-intensive and may include more conditionals, which tends to increase the number of copies of the points-to table.

9 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 [19], 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.

References

[1] L. O. Andersen. Program analysis and specialization for the C programming language. PhD thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

[2] B. Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–37, 1998.

[3] M. Burke, P. Carini, J. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Lecture Notes in Computer Science, 892, Springer-Verlag, Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250, 1995.

[4] M. Buss, S. Edwards, B. Yao, and D. Waddington. Pointer analysis for source-to-source transformations. Technical Report CUCS-028-05, Department of Computer Science, Columbia University, 2005.

[5] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 233–245, 1993.

[6] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 35–46, 2000.

[7] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 242–256, 1994.

[8] M. Fahndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 253–263, 2000.

[9] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 254–263, 2001.

[10] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, 2001.

[11] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 235–248, 1992.

[12] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[13] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 13–22, 1995.

[14] L. Semeria, K. Sato, and G. D. Micheli. Synthesis of hardware models in C with pointers and complex data structures. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(6):743–756, 2001.

[15] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[16] E. Visser. Stratego xt. http://www.stratego-language.org.

[17] E. Visser and Z. Benaissa. A core language for rewriting. http://www.elsevier.nl/locate/entcs/volume15.html.

[18] D. Waddington and B. Yao. High fidelity C++ code transformation. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA)*, 2005.

[19] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 1–12, 1995.