Program Slicing in the Presence of Pointers (Extended Abstract)

James R. Lyle
National Institute of Standards and Technology
jimmy@swe.ncsl.nist.gov

David Binkley
Loyola College in Maryland
National Institute of Standards and Technology
binkley@swe.ncsl.nist.gov

Index Terms--Program slicing, data flow analysis, debugging, pointers, software tools, software engineering.

I. Introduction

Program slicing is a family of program decomposition techniques based on extracting statements relevant to a computation in a program. Program slicing as originally defined produces a smaller program that reproduces a subset of the original program's behavior [16]. This is advantageous since, by excluding irrelevant statements, the slice can collect an algorithm that may be scattered throughout a program. It should be easier for a programmer interested in a subset of the program's behavior to understand the corresponding slice than to deal with the entire program. The utility and power of program slicing comes from the potential automation of tedious and error prone tasks. Program slicing has applications to program debugging [17, 18, 13, 19, 16], program testing, program integration [7], parallel program execution [16], software metrics [16, 14], reverse engineering [2], and software maintenance [5]. Several variations on this theme have been developed, including *program dicing*, *dynamic slicing* [1, 12] and *decomposition slicing*.

The most important goals of research in program slicing are the following:

- to compute slices faster,
- to find algorithms that produce smaller slices,
- to develop new applications of program slicing, and,
- to compute slices for a wider collection of language constructs.

There have been many papers on applications of program slicing, and improvements to algorithm speed and precision [4, 8], but few papers have addressed difficult language constructs such as pointers [9, 11]. Program slicing cannot become a useful tool until slicing algorithms can deal with real world program features such as pointers. This paper presents an algorithm for computing program slices for programming languages that use pointers. An early version of this pointer tracking algorithm has been implemented as part of **unravel**, a program slicing tool developed at NIST. The goal of our work is to produce a slicing algorithm that is useful on real code (ANSI C). We begin with a simple algorithm that is expensive in storage and computation time, and then improve its performance by exploiting features of good programming style.

A. Definitions

Definition: A *slicing criterion* for a program slice is a tuple $< i, V >$ where *i* is a statement in the program and *V* is a subset of the program variables. A slice of a program is computed *with respect to v, at statement i*.

Definition: A *program slice* of program *P* for slicing criterion, <i,V>, is an executable program obtained by deleting zero or more statements from P such that the values of the variables in *V* are the same when execution reaches statement *i* of both P and the slice of P [16].

Definition: *Refs(n)* is the set of variables referenced (the variable's value is used) at statement *n*.

Definition: *Defs(n)* is the set of variables defined (the variable is assigned a value) at statement *n*.

Definition: *Succ(n)* is the set of statements that could be executed immediately after statement *n*.

Definition: *Irefs(n)* is the set of pointer variables dereferenced that have values used at statement *n*. The set members are represented as ordered pairs of the form *(v,k)*, where *v* is a pointer variable and *k* is the level of dereferencing (i.e. *v* has *k* * operators applied).

Definition: *Idefs(n)* is the set of pointer variables dereferenced that have values assigned at statement *n*. The set members are represented as ordered pairs of the form *(v,k)*, where *v* is a pointer variable and *k* is the level of dereferencing (i.e. *v* has *k* * operators applied).

B. Background

It is not possible to keep track of exactly where a pointer in a linked list points. However, since the statements involved in the creation and manipulation of a linked list could operate on any of the nodes in the linked list, keeping detailed information about which list node each node points to may not enable a slicing algorithm to compute a smaller slice. We recognize two types of addresses, addresses of static objects and addresses of dynamic objects, each address type requires a different approach. The essential difference is that addresses of dynamic objects cannot easily be tracked precisely and an indirect assignment through a dynamic address must be treated like an assignment to an array element.

A slicing criterion using a dynamic object can be ambiguous. For example, consider the slicing criterion, **<22,at->value>** on the program in Figure 1. The slice should consist of

```
 1   typedef List_struct List;
 2   struct List_struct {
 3       int  value;
 4       List     *next;
 5   } List;
 6
 7   main()
 8   {
 9       List     *at,*head = NULL,*node;
10       int  n;
11
12       n = 0;
13       while (n < 10){
14           node = (List *) malloc (sizeof(List));
15           node -> value = n++;
16           node -> next = head;
17           head = node;
18       }
19       at = head;
20       at -> value = 47;
21       at = at -> next;
22       at -> value = 8;
23   }
```

Figure 1.

statements: {**12, 13, 14, 16, 17, 19, 21, 21**}. However, if the assignment to **at->value** is treated as a scalar then lines 20 and 15 are not included in the slice. If the slicing criterion means *slice on the* **value** member that **at** *currently points to*, then lines 20 and 15 should not be included in the slice. However, if the slicing criterion means *slice on the any* **value** member that **at** *might point to*, then lines 20 and 15 should be in the slice.

C. Algorithm Overview

Our approach for ANSI C exploits how pointers are used in real world programs. There are several common patterns of usage that can be exploited to produce practical pointer slicing algorithms.

The algorithm requires a pre-pass over the program to compute lists of possible address values for each pointer at each statement in the program. Our algorithm is similar to one developed by [15]. The standard slicing algorithm must be modified to not only keep track of data but also addresses. These modifications can be made to either a control flow based slicing algorithm [16] or a PDG [4] based slicing algorithm.

II. Changes to the Slicing Algorithm

We present our changes in the context of a recursive formulation of program slicing for a language with no *if* statements or *loop* statements. The formulation can be easily extended for control structures by associating a list of statements required to be also included in the slice with each statement. This recursive formulation is convenient for discussing program slicing but, it is not a practical implementation algorithm.

Let v be some program variable name, n be some statement. Let $P_i(n, v)$ be a function that returns the set of addresses that *...*v (where there are i *'s) could point to at statement n. P_1 is understood if no subscript is written. Note that $P_0(n, v) = \{v\}$ and for $i > 1$, $P_i(n, v) = \{x | x \in P_1(n, y) \forall y \in P_{i-1}(n, v)\}$. Let *is_dynamic(a)* be a boolean function that returns *true* if a is dynamically created and returns *false* otherwise.

Recursive Slicing Algorithm:

$$S_{<m \in succ(n), v>} = \begin{cases} \{n\} & \text{if v} \in \text{defs(n) \& refs(n)} = \emptyset \\ S_{<n,v>} & \text{if v} \notin \text{defs(n)} \\ \{n\} \bigcup S_{<n,x>} \forall x \in refs(n) & \text{if v} \in \text{defs(n) \& refs(n)} \neq \emptyset \end{cases}$$

Definition: The *active set* at statement *n* is the union of all variables *x* found in slicing criterion on statement *n* induced by the recursion. The active set at statement *n* is the set of all variables that could influence any statement in the slice executed after statement *n*. Any variable not in the active set at statement *n* can be safely changed without affecting the computation of the slice.

We have two cases, how pointers change treatment of *ref* sets and how pointers change treatment of *idef* sets.

A. Ref sets

Suppose we have a statement such as:

n: A = *B + C;

Consider the slicing criterion: $S < m, X >$ where $m \in$ *succ(n)*. If $A \notin X$ this statement does not belong in the slice and can be passed by. If $A \in X$ & $A \in$ *defs(n)* the slice, $S < m, X >$, is the statement, *n*, unioned with the following:

$$\begin{aligned} S_{<m=succ(n),X>} &= \{n\} \\ &\bigcup S_{<n,v>} \forall v \in refs(n) \\ &\bigcup S_{<n,v>} \forall v \in X \ \& \ v \notin defs(n) \\ &\bigcup S_{<n,v>} \forall v \in P(n,B) \end{aligned}$$

The last member of the above union generalizes to *k* levels of indirection by:

$$S_{<n,v>} \forall v \in P_i(n,b) \forall i, 1 \le i \le k (b,k) \in irefs(n)$$

B. Def sets

Suppose we have a statement such as:

n: *B = A;

Consider the slicing criterion: $S < m, X >$ where $m \in$ *succ(n)*. If $X \cap P(n, B) \neq \emptyset$ statement *n* should be included in the slice, unioned with the following:

$$
\begin{aligned}
S_{<m=succ(n),X>} = \{\mathrm{n}\} \\
\quad \cup S_{<n,v>} \forall v \in refs(n) \\
\quad \cup S_{<n,v>} \forall v \in P(n, B) \\
\quad \cup S_{<n,B>}
\end{aligned}
$$

If there are at least two members of P(n,B), say, $y \notin X$ and$\mathrm{z} \in \mathrm{X}$, the slice $S_{<n,P(n,B)>}$, captures the case where B actually points to a variable (y in this example) not in X and z should remain in the active set since statement *n* does not change the value of z. If z is the only member of *P(n,B)* and if *is_dynamic(z)* is *false* then the slice on *z* should be omitted since statement *n* must change the value of *z*.

If *is_dynamic(z)* is *true* then z must be sliced on since there could be several instances of *z*. *B* could point to one of several possible addresses other than *z* and therefore z must remain in the active set for statement n.

III. Capturing The Pointer State

The extension to program slicing requires that it be possible to determine where pointer variables might be pointing at each program statement. Addresses are introduced into the pointer state from two sources, applying the *address of* operator (&) to a declared variable and the address of storage allocated from the heap returned by **malloc** class functions. The information on addresses is collected by identifying statements that introduce addresses into a program and the addresses are then propagated through the program to obtain the pointer state for each statement. This is somewhat like symbolic execution [10, 3] in that possible values of variables (pointer variables only) are determined by static program analysis rather than by actual program execution. It is useful to observe that embedded within the program flow graph is a subgraph that represents generation and manipulation of addresses. Every program statement does not have to be analyzed since only the statements corresponding to nodes of this subgraph need to be analyzed to determine the pointer state. This subgraph is smaller, usually much smaller, than the original program.

Definition: The *Pointer State Subgraph* (PSS) is obtained for a control flow graph by deleting nodes that correspond to statements that do not assign a pointer value and by deleting any branch node and corresponding join node if all nodes from the branch node to the corresponding join node have been deleted.

First, we consider the pointer state for addresses of declared variables.

A. Addresses of Declared Variables

Given that n is a statement, v is a variable, and $Q(n, v)$ = {all currently known variables that v might point to after statement n executes}. The function Q is used to accumulate information about the pointer state. Eventually Q becomes P, the pointer state function used by the slicing algorithm. Q_i is defined analogously to P_i.

The algorithm for capturing the pointer state is the following:

1. Construct the PSS
2. Scan the PSS for address creation
3. Propagate created addresses
4. Repeat until no more changes occur

The pointer state of each node is propagated from the node to all successor nodes by taking the union of the pointer state for all variables not defined at the node with any changes required by the propagation rule for the kind of statement corresponding to the node of the PSS. The following table gives the basic address propagation rules.

Statement		Propagation Rule
A =	&x	Q(n,A) = {x}
A =	B	Q(n,A) = Q(n,B)
A =	*B	Q(n,A) = $\bigcup Q(n, X) \forall X \in Q(n, B)$
A =	**B	Q(n,A) = $\bigcup Q(n, X) \forall X \in Q(n, Y) \forall Y \in Q(n, B)$
A =	*...*B	Q(n,A) = $Q_k(n, B)$ where k is indirection level
*A =	&x	Q(n,Y) = Q(n,Y) $\bigcup$ {x}, $\forall Y \in Q(n, A)$
**A =	&x	Q(n,Z) = Q(n,Y) $\bigcup$ {x}, $\forall Z \in Q(n, Y) \forall Y \in Q(n, A)$
*...*A =	&x	Q(n,Z) = $Q_i(n, A) \bigcup$ {x}, where k is indirection level

B. Addresses From The Heap

Consider line 14 of Figure 1:

```
14    node = (List *) malloc (sizeof(List));
```

The variable *node* is receiving the value of a different address of storage on the heap each pass through the loop at lines 13--18. It is not possible to determine by static analysis how many times storage is dynamically allocated from the heap or to track assignments to individual locations. This implies that an approximation that lumped together several dynamically allocated storage locations should be introduced. For example, the heap could be treated as a single array with assignments to individual members treated as both an assignment and a reference or tracked with methods such as [6]. However, treating the heap as a single array is too conservative; by partitioning the heap into several arrays we can do much better. The heap should be partitioned by object type so that all objects of the same type are treated as if they have been collected into a single array. Then the entire array of objects of a single type from the heap can be treated as a single object of a given type with a change to a dynamic object treated as an update to an array.

It is not easy to identify the type of object that a chunk of dynamically allocated storage is being used to contain since routines such as **malloc** only returns an address to a block of storage of the requested size. This address may be directly assigned to a pointer of the matching type or may be assigned by a cast operation to a pointer of another type or even to an integer variable. Such *pointer laundering*, casting a pointer to another pointer type or even non-pointer type so that the true type of the pointer is obscured, requires that all assignments of values that are used as pointers must be tracked to ensure that the pointer state is correctly captured.

To capture the pointer state, introduce a pseudo-variable address for each address allocation point in a program. This is usually every call to a **malloc** class function, however, it might be the case that a program has its own memory allocation function to allocate storage for program objects. The memory allocation function calls **malloc** or directly obtains heap storage from the run time environment. For such a program, each memory allocation function invocation would be where to introduce a pseudo-variable.

IV. Summary

The contributions of this paper are the changes required to incorporate pointers in program slicing algorithms, and the informal definition of the pointer state subgraph. Slicing programming languages such as C require efficient treatment of pointers. The pointer state subgraph provides a significantly smaller graph for analysis to capture the pointer state than the entire program flow graph.

References

[1] H. Agrawal and J. R. Horgan, “Dynamic Program Slicing,” *Proceedings of the ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation*, pp. 246-256 (June 20-22, 1990).

[2] J. Beck and D. Eichmann, *Program and Interface Slicing for Reverse Engineering,* Proceedings Working Conference on Reverse Engineering, Baltimore, Maryland (May 21-23, 1993).

[3] L. A. Clarke, “A System to Generate Test Data and Symbolically Execute Programs,” *IEEE Transactions on Software Engineering* **SE-2** p. 215 222 (Sept. 1977).

[4] J. Ferrante, K. Ottenstein, and J. Warren, “The Program Dependence Graph and its Use in Optimization,” *ACM TOPLAS* **9**(3) pp. 319-349 (July 1987).

[5] K. B. Gallagher and J. R. Lyle, “Using Program Slicing in Software Maintenance,” *IEEE Transactions on Software Engineering* **17**(8) pp. 751-761 (August 1991).

[6] R. Hamlet, B. Gifford, and B. Nikolik, *Exploring Dataflow Testing of Arrays,* Proceedings 15th International Conference on Software Engineering, Baltimore, Maryland (May 21-23, 1993).

[7] S. Horwitz, J. Prins, and T. Reps, “Integrating Noninterfering Versions of Programs,” *ACM TOPLAS* **11**(3) pp. 345-387 (July 1989).

[8] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural Slicing Using Dependence Graphs,” *ACM TOPLAS* **12**(1) pp. 26-60 (January 1990).

[9] S. Horwitz, P. Pfeiffer, and T. Reps, “Dependence analysis for pointer variables,” *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices* **24**(7) pp. 28-40 (July 1989).

[10] W. E. Howden, “Symbolic Testing and the DISSECT Symbolic Evaluation System,” *IEEE Transactions on Software Engineering* **SE-3** pp. 266-278 (1977).

[11] J. Jiang, X. Zhou, and D. Robson, *Program Slicing For C -- The Problems In Implementation,* Proceedings Conference on Software Maintenance 1991, Sorrento, Italy (October 15-17, 1991).

[12] B. Korel and J. Laski, “Dynamic Program Slicing,” *Information Processing Letters* **29**(3) pp. 155-163 (October 1988).

[13] J. R. Lyle and M. Weiser, *Automatic Program Bug Location by Program Slicing,* Proceedings of Second International Conference on Computers and Applications, Peking, China (June 1987).

[14] L. Ott and J. Thuss, *Slice Based Metrics for Estimating Cohesion,* Proceedings First International Software Metrics Symposium, Baltimore, Maryland (May 21-23, 1993).

[15] William E. Weihl, “Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables,” *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages,* pp. 83-94 (January 1980).

[16] M. Weiser, “Program Slicing,” *IEEE Transactions on Software Engineering,* **SE-10**(4) pp. 352-357 (July 1984).

[17] M. Weiser, “Programmers Use Slices When Debugging,” *CACM* **25**(7) pp. 446-452 (July 1982).

[18] M. Weiser and J. R. Lyle, “Experiments on Slicing-Based Debugging Aids,” pp. 187-197 in *Empirical Studies of Programmers*, ed. E. Soloway and S. Iyengar,Ablex Publishing Corporation, Norwood, New Jersey (1986).

[19] M. Weiser, “Program Slicing,” *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439-449 (March 1981).