Slicing pointers and procedures (abstract)

Michael D. Ernst
Microsoft Research
1 Microsoft Way, Redmond, WA 98052 USA
Email: mernst@research.microsoft.com

January 13, 1995

Abstract

Program slicing restricts attention the components of a program relevant to evaluation of one expression, the slicing criterion. Our slicer, which explicitly represents the store as an aggregate value is the first to support arbitrary pointer manipulations and aggregate values, and is faster than more limited techniques. We also improve the asymptotic complexity of slicing in the presence of procedure calls, and of a preprocessing step for computing dependences of procedure returns on formals. Additionally, our interprocedural slices can be smaller than those produced by other techniques. We implement these techniques in the first slicer for an entire practical programming language (ANSI C, except `longjmp`).

1 Introduction

Program slicing [Wei84] is a technique for visualizing dependences and restricting attention to just the components of a program relevant to evaluation of the slicing criterion, which is the particular expression(s) of interest. Backward slicing reveals which parts of the program the slicing criterion's meaning depends on, while forward slicing determines which parts of the program depend on the meaning of the slicing criterion. While this paper explicitly discusses only backward slicing, the techniques are equally applicable to forward slicing.

In programming environments, slices aid program understanding, maintenance, testing, and debugging by demonstrating the relationships between parts of a program. Slices can also limit the amount of the program manipulated and assist with parallelization and comparison and integration of program versions. Programs can be specialized to produce fewer results (and run faster) by generating code from slices.

This paper addresses static slicing, which uses analysis to discover dependences and indicates the program components that *may* have an effect on (or may be affected by) the slicing criterion values. By contrast, a dynamic slice [AH90, Agr91, Ven93] gives definite information for some particular execution by maintaining an execution trace of a running program.

This paper presents a new approach to slicing based on a functional representation called the value dependence graph (VDG); in particular, improved techniques for dealing with pointers, structures, and procedures are presented.

We show how to slice in the presence of arbitrary pointer manipulations. Our method hinges on explicit representation of the store as an aggregate value with one component for each location. This representation permits dependences in terms of values rather than variables, and it allows aggregates to be sliced elementwise. Previous techniques have handled only call-by-reference parameters or limited pointer variables, and have not been as efficient as our method. Our slices include only the relevant parts of aggregate values, which were only partially supported by previous methods.

We improve the graph-based interprocedural slicing method [HRB90] in several ways. We compute summary dependences faster than previous methods (by a factor of at least $O(n)$) and via a simpler algorithm. Our representation is more compact, so in the absence of aliasing, our slicing algorithm runs in time proportional to the size of the original program, compared to quadratic time for previous algorithms. Aliasing degrades our algorithm by $O(n)$ but degrades previous algorithms exponentially. Our slices are more precise because they exclude more procedure calls and bodies by distinguishing between direct and indirect summary dependences.

These techniques are implemented in VDGomatic, a slicer that accepts input via mouse clicks and displays closure slices by highlighting portions of the program in the programmer's editor. It is the first slicer to support an entire practical programming language (ANSI C, with the exception of `longjmp`). Currently the system is limited to programs shorter than approximately 10,000 lines.

This paper first describes the value dependence graph (VDG), the intermediate representation used by our slicer, then gives an algorithm for slicing with respect to this representation. The next two sections extend the basic algorithm to properly account for pointers and procedures. Finally, the implementation is discussed. Other reports [Ern94a, Ern94b] provide more details and discuss our techniques for expression-oriented slicing (we improve the precision of slicing criteria, display of slices, and of slices themselves), for executable slicing (we bypass syntactic constraints in code generation and supply precise liveness information to the back end), and for slicing optimized code (which improves dependence information) while displaying results in terms of the original program.

2 Slicing the value dependence graph

This section briefly describes the value dependence graph (VDG), the program representation used by our slicing algorithm, then describes the basic slicing algorithm. Later sections further detail and enhance the algorithm.

2.1 The value dependence graph

The value dependence graph (VDG) [WCES94a, WCES94b] is a sparse, parallel, functional, dataflow-like program representation for imperative programs. It is sparse because it directly connects consumers to producers of values, enabling fast, incremental analyses and transformations. It is parallel because it makes no commitment regarding the relative order of noninterfering computations. It is functional because all values are explicitly represented, even those usually left implicit, such as the store.

The insight underlying the VDG's design is that only the values computed by a program matter. The original names and control flow are incidental to the underlying computation, so such artifacts of the program text should not be enshrined in the representation, as they are in traditional intermediate representations such as the control flow graph. The consequences of this decision are

- all values are explicit, even ordinarily-implicit ones such as I/O streams,
- selectors (resulting from `if` and `switch`) choose among values, not control paths, and
- non-sequential control flow is represented by function calls.

The VDG is composed of nodes which represent computation and arcs which carry values between computations. Figure 1 shows a procedure and its VDG representation. The `sum_product` procedure returns a modified I/O stream resulting from a call to `printf`. Some of the arguments to `printf` are themselves computed by a function call, where the function's body corresponds to

```
void sum_product(int n)
{
  int sum = 0;
  int product = 0;
  int i = 0;

  while (i++ < n)
    { sum = sum + i;
      product = product * i;
    }
  printf("Sum %d, product %d",
         sum, product);
}
```

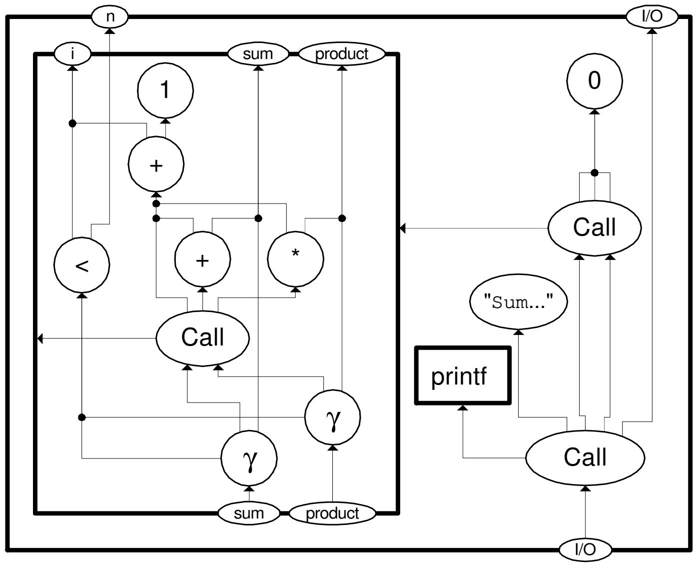


Figure 1: A procedure and its VDG representation. The arrows point from consumers to producers of values. A γ node is a selector whose output is its left or right input, depending on whether its boolean argument (attached to the side of the γ node) is true or false. The heavy boxes represent function bodies, with arguments along the top edge and return values along the bottom edge.

the loop body in the original program. Depending on the value of a less-than (`<`) comparison, that function either returns its arguments (the loop terminates) or does some computation and makes a recursive call (another loop iteration is invoked). (Section 3.1 introduces lookup and update nodes, which represent operations on the store.)

The VDG is a practical representation. The VDG's size is proportional to that of the original program (certain optimizations can make the VDG quadratic in the size of the original program). A program's VDG can be computed in linear time, even for programs with irreducible control flow. Furthermore, efficient code can be generated from the VDG [WCES94b, Ste93].

2.2 Basic slicing algorithm

The basic slicing algorithm [OO84] on the VDG is extremely simple. A slice consists of all computations encountered in a graph traversal starting at the slicing criterion, which is a particular dependence (value) arc. A backward slicing traversal follows consumer-producer arcs; at formal parameters, the traversal proceeds at corresponding actuals on call sites in the slice, and at call results, the traversal continues both at the appropriate return nodes in the callee and at all actuals corresponding to formals in the slice. Because each VDG arc is visited at most once, this algorithm runs in time linear in the size of the slice, which is no larger than the VDG.

Figure 2 shows a slice of the `sum_product` procedure of Figure 1. The slicing traversal starts at the slicing criterion, an arc representing one of the arguments to `printf`. When the traversal encounters a call node, it continues at the corresponding procedure return (annotated `sum`). That return node (transitively) depends on the left-hand result of the recursive call (the corresponding return for which has already been processed), on formal `n` of `sum_product`, and on formals `i` and `sum` of the inner procedure. Encountering the latter two formals causes the traversal to continue at the corresponding actuals for both calls; it proceeds to the addition nodes and the constants. The traversal ends when no more unreached nodes are encountered.

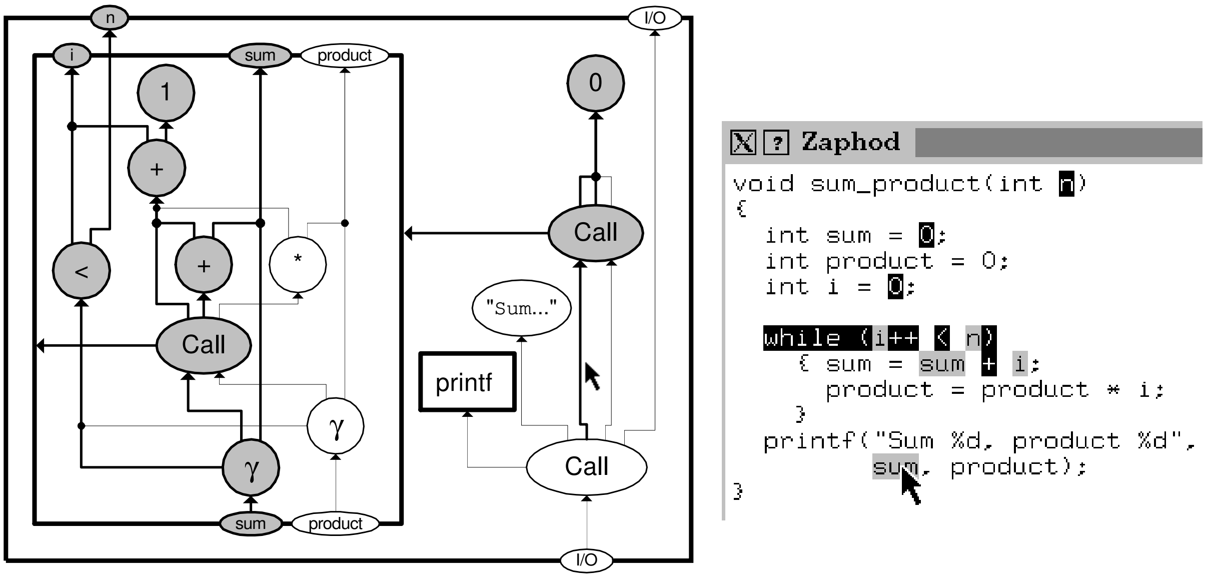


Figure 2: A backward slice indicates which computations can affect the run-time value of the slicing criterion. The textual slicing criterion is the use of `sum` in the call to `printf`; its VDG translation is an edge connecting two call nodes. In the VDG, edges and nodes in the slice are shown thicker and shaded, respectively. The screen dump on the right shows how our system indicates, in the programmer's editor, the computations in the slice. The dark highlighting indicates computation; the light highlighting indicates other names for the values in the slice. The user interface issues of communicating a VDG slice to the user [Ern94a, Ern94b] are outside the scope of this paper.

The advantages of our slicing approach accrue both from the VDG's features and their exploitation by our algorithms. The VDG's fine granularity makes slicing criteria and results more precise than with statement-based representations. The graph directly links value producers with consumers, so all dependences are made explicit in a single graph. Because all values and computations are explicitly represented, the algorithm need not account for hidden constraints and side effects.

The basic slicing algorithm is simple, fast, and easy to understand, but imprecise: the slices it produces are larger than necessary. The next two sections extend this algorithm to address pointers and aggregate values and procedure calling context.

3 Pointers and aggregate values

Modern programming languages depend on the explicit or implicit use of pointers for organizing data, and even in the absence of pointer variables, function calls can introduce aliasing, which complicates slicing. A start has been made on these problems by suggesting techniques for addressing by-reference function parameters [HDC88, HRB90, LC92] and providing support for pointer variables [JZR91, LR94]. We provide the first discussion and implementation of slicing techniques for handling arbitrary pointer manipulations (including arithmetic, casting, and function pointers). We then discuss the similar challenges raised by aggregate values, which are also fully supported by VDGomatic.

A strategy for slicing in the presence of pointers has three components: a representation for pointers or aliased values; an analysis to determine possible pointer values or possible alias relations; and a slicing algorithm that takes advantage of the representation and the analysis results.

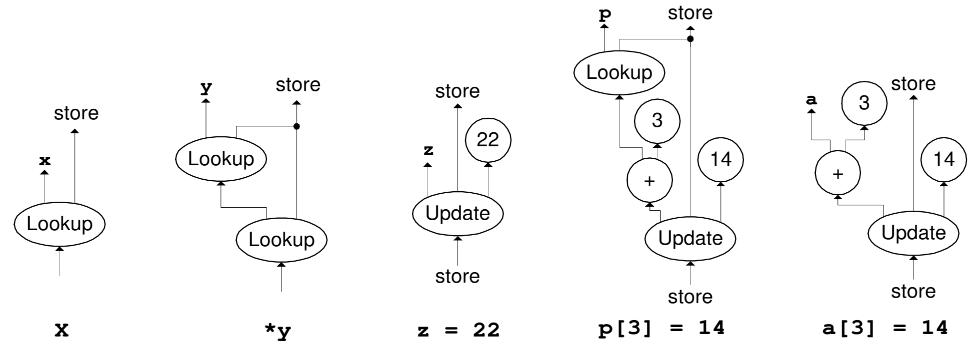


Figure 3: The VDG represents memory references by lookup nodes and memory stores by update nodes. This figure specially indicates store values and uses variable names in typewriter font for variable addresses. Variables `y` and `p` contain pointers; `a` is an array.

3.1 Representing pointers

The VDG, a functional representation, explicitly represents all values, including locations and stores. Two types of node operate on stores. Lookup nodes represent variable references and pointer dereferences; they take two arguments—a location and a store—and return the contents of the location in the store. Update nodes represent assignments to memory; they take three arguments—a location, a store, and a value—and return a modified store in which the location's content has been set to the value. For ordinary variable references and assignments, the location is a constant resolved at link time. In general, the location may be produced by an arbitrary expression, such as another lookup or an arithmetic operation. Figure 3 shows the representation for various memory references and assignments.

Stores are represented as aggregates, because store locations can be manipulated independently; typical store operations manipulate only a small portion of the store. It is sometimes helpful to view a store arc as a set of component arcs, one for each value contained in the store. A pointer value specifies a location in a store. Explicit representation of stores permits dependences to be expressed in terms of values rather than variables.

When the location argument to an update node is the same as that of a lookup node that (eventually) consumes the update's result, and the location is not set between the update and the lookup, then the update's value input can be routed directly to the consumer(s) of the lookup. The lookup (and possibly the update as well) becomes dead and can be removed from the graph. Figure 4 demonstrates this "short-circuiting" transformation, which, along with arity raising [WCES94b, Ruf94b], eliminated all store operations from Figure 2. The transformation depends only on location equality, not on the location argument being a constant. (Arbitrary pointer manipulations complicate proving that the location is not modified between the update and the lookup.)

Explicit representation of locations and stores in the graph simplifies and clarifies slicing and other analyses, as there are no hidden effects or values to account for. Additionally, information about pointers is easy to reflect in the graph (by short-circuiting (Figure 4), store splitting [SW93, Ste94b], and other transformations), making it immediately available to all analyses.

3.2 Pointer analysis

VDGomatic makes use of a points-to analysis [CWZ90, HEGV93, Ruf94b], which indicates the possible values for each location-valued expression. (In our representation a points-to analysis is more natural than an alias analysis, which produces a set of possible alias pairs.) In the worst case,

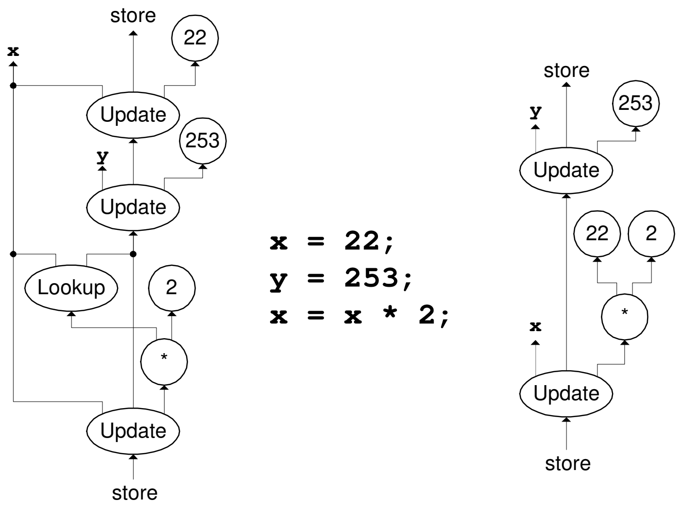


Figure 4: "Short circuiting" of a lookup node and an update node setting the looked-up location eliminates the lookup and can make the update dead as well.

the analysis indicates that a location-returning expression could return any location at run time, but in practice, on average fewer than two values are possible [LRZ93, EGH94, Ruf94a, Ste94a]. The details of the analysis are irrelevant to the slicing algorithm.

3.3 Slicing algorithm

We now extend the slicing algorithm to process lookup nodes, to traverse store edges, and to process update nodes.

Upon encountering a lookup node, a slicing traversal proceeds to the lookup's location argument in the usual way. The slice should not include the entire store argument, however, because only some operations that contribute to the store are relevant to the lookup node's value.

Unlike scalar values, stores can be partially included in a slice. (An aggregate-valued slicing criterion optionally specifies a component of that aggregate [Ern94a, Ern94b].) A slicing traversal along a store specifies a location, and the result includes only operations that can set that location. At a lookup node, the slicing traversal proceeds independently for each location that may be looked up—that is, each location specified by the points-to analysis for the lookup node's location argument. Like scalar values, store components are each visited only once, so a store arc is traversed no more often than the number of locations it contains.

Update nodes encountered when slicing on location l in a store fall into three categories:

strong update of l: the location argument always returns l.[1] This is a killing definition that definitely sets the value in l. The slicing traversal proceeds to the update node's value and location inputs, but not to its store input, since previous store values are irrelevant.

weak update of l: the location argument may return l. This is a preserving definition that might set the value in l but might leave it unchanged. The slicing traversal proceeds to the update node's value and location inputs, and also proceeds along its store input with respect to l, in order to include the location's previous value.

[1]Some locations returned by the points-to analysis (such as those for `malloc` calls and for arrays, which are treated as single elements) represent multiple runtime locations and cannot be strongly updated; updates to them are always weak.

```
int main()
{
  int x, y, z;
  int *xy1, *xy2, *xy3;
  int *yz1, *yz2;

  srand(time(0)); /* init RNG */
  xy1 = rand() & 1 ? &x : &y;
  xy2 = rand() & 1 ? &x : &y;
  xy3 = rand() & 1 ? &x : &y;
  yz1 = rand() & 1 ? &y : &z;
  yz2 = rand() & 1 ? &y : &z;

  x = 1;
  y = 2;
  z = 3;
  *xy1 = 4;
  *yz1 = 5;
  y = 6;
  *xy2 = 7;
  *yz2 = 8;
  printf("%d\n", *xy3);
  return(*xy3);
}
```

Figure 5: Slicing in the presence of pointer manipulation. Pointer `xy`n points to either `x` or `y`, and `yz`n points to `y` or `z`. The final value pointed to by `xy3` is 1, 4, 6, 7, or 8, depending on the values of four of the calls to `rand`, the pseudorandom number generator. Highlighting of the `int` declaration indicates that the addresses of `x`, `y`, and `z` are manipulated.

other: the location argument never returns l. This isn't a definition of the value in l at all, so it is irrelevant for this slicing criterion. The slicing traversal proceeds along the update node's store input with respect to l.

Figure 5 shows the result of slicing a program containing pointer manipulations.

VDGomatic does not actually consider one location at a time, then union the results. Instead, each traversal of a store edge specifies a set of locations. Encountering a strong update causes locations to be removed from the set, but the traversal continues if the set is nonempty. Sets are combined when the traversal simultaneously reaches a node from multiple consumers. (The algorithm queues values needing to be processed, combining queue entries for a single value.) This strategy also enables optimizations that take advantage of pointer equality in addition to the points-to analysis [Ern94b].

3.4 Aggregate values

The slicing algorithm treats stores as aggregate values; the same mechanisms can be applied to other aggregates, such as C `structs`. An edge may be traversed a number of times equal to the number of components in the value carried by the edge. Each traversal specifies which components of the aggregate are being demanded and sliced upon, and those components need not be revisited if the traversal returns to the aggregate value.

The VDG representation of structures is similar to that of stores. Lookup-struct and update-struct nodes operate on structured values; member specifiers, which give offset and size information with respect to a base pointer, indicate parts of structured values.

For analysis, aggregates are broken into quanta small enough that no member specifier only partially includes some quantum. C `unions` may cause structure elements to overlap, resulting

```
typedef struct { int slot1; int slot2; } Twoslots;

Twoslots foo(Twoslots a, int b)
{
  Twoslots r;
  r.slot1 = a.slot2;
  r.slot2 = a.slot1 * b;
  return r;
}

int main()
{
  Twoslots y = { 22, 23 };
  Twoslots x = foo(y, 44);
  return x.slot1 + x.slot2;
}
```

```
typedef struct { int slot1; int slot2; } Twoslots;

Twoslots foo(Twoslots a, int b)
{
  Twoslots r;
  r.slot1 = a.slot2;
  r.slot2 = a.slot1 * b;
  return r;
}

int main()
{
  Twoslots y = { 22, 23 };
  Twoslots x = foo(y, 44);
  return x.slot1 + x.slot2;
}
```

Figure 6: The first slot of `foo`'s return value depends directly on the second slot of formal `a`, and the second slot of the result value depends indirectly on the first slot of formal `a` and indirectly on formal `b`. Slices on `x.slot1` and `x.slot2` are shown. While the traditional method includes parts of `foo` in a slice on `x.slot1`, we omit it by default because there is no dependence on its body.

in more quanta for the union than either alternative has slots; member specifiers for overlapped fields encompass multiple quanta, just as pointer operations can reference or assign to multiple store locations. Every bit in a quantum is guaranteed to be identically operated upon, and each quantum is independently treated by the slicing algorithm. Casting, which effectively makes gives union types to arbitrary variables, is similarly handled [Ern94b].

While pointers are more expressive (and member specifiers can be resolved at compile time), C's `union` construct enables aliasing, so many of the same issues are raised for structures as for pointers. Since aggregates may be nested (consider a struct in a union in a struct in a store), the algorithms are mechanically more complicated, but the concepts and algorithmic complexity are the same. Figure 6 shows an example of slicing on aggregate components.

3.5 Complexity

The basic slicing algorithm runs in time linear in the number of arcs or nodes in the VDG, whose size is proportional to that of the original program. Extended to account for pointers and aggregates, the algorithm runs in time linear in the number of values in the graph, where aggregate components are counted individually. In the worst case, the number of such values is quadratic in the size of the original program, because a single store containing all the values in the global store can be threaded through every computation.

In practice, the graph is little larger than the original program; only widespread aliasing, which is rare, results in VDGs containing many large stores. In the absence of aliasing, the number of values computed is proportional to the size of the original program (equivalently, the number of nodes or edges in the VDG). VDG construction makes stores smaller and propagates them through fewer computations by splitting stores into noninterfering pieces, by removing elements from stores (converting them into standalone scalars), and by eliminating store operations. Neither these transformations nor support for slicing aggregates or pointers increases the size of the VDG (beyond the storage requirements of the points-to analysis).

3.6 Related work

Two previous slicers provide limited support for pointer variables (but no arithmetic, casting, function pointers, etc.). CPS [JZR91] adds dummy variables and literals for pointer dereferences and address-of operations; pointer references and assignments count as uses and modifications of

```
int g1, g2;

void set_globals(int a, int b)
{
  g1 = a;
  g2 = b;
}

int main()
{
  int tmp;
  set_globals(10, 20);
  tmp = g1;
  set_globals(30, 40);
  return(tmp + g2);
}
```

Figure 7: Calling context determines which actual parameters should be included: an accurate slice includes different parameters for the two calls to `set_globals`. VDGomatic permits users to omit procedure bodies where no computation occurs: the slice shown clearly indicates that the result is the sum of 10 and 40. Both the lighter highlighting and including portions of the body (as in Figure 11) help indicate which variables carry dependences between calls, when that information is desired.

the corresponding dummy variables. Ghinsu [LR94] uses an extended storage shape graph to add data dependence edges to the program representation.

Call-by-reference function parameters can be addressed by duplicating each procedure for every possible alias pattern among its parameters [HDC88, HRB90, LC92], which increases the program size exponentially, or by assuming all possible aliases can occur simultaneously, which degrades slice quality [HRB90].

The only slicer besides VDGomatic which addresses structures is Ghinsu [LR94]. It handles basic operations on C's `struct`, but its type system cannot accommodate `union`, casting, and other operations, and no details are given regarding how the alias analysis and slicer deal with structures.

4 Interprocedural slicing

The basic slicing algorithm of Section 2.2 has two flaws with respect to interprocedural slicing. First, the same set of actuals is demanded at each call site in the slice. If a slice on one call of a procedure forces inclusion of its first parameter, and elsewhere in the slice another call of the same procedure induces a demand on its second parameter, then the resulting slice includes both actual parameters at each call site. For instance, in Figure 7, the basic slicing algorithm would include both arguments of each call to `set_globals`. Second, when the slicing criterion appears in a procedure, the algorithm can fail to include procedures in the call chain—that is, procedures that may (transitively) call the procedure containing the criterion. This is a result of ascending from formals only to corresponding actuals at call sites in the slice. Figure 8 demonstrates the proper behavior, and Figure 12 shows slicing in the presence of both pointers and procedure calls.

Our solution to these problems is patterned after Horwitz's [HRB90]. Before slicing, a summary of the dependences of each procedure return on each formal parameter is computed. This summary permits the slicing traversal to proceed directly from a call result to the actuals that affect that result. Appropriate portions of the procedure body are added to the slice separately, by performing a traversal which starts at the procedure return corresponding to the call result and ends at the procedure's formals (the slice has already proceeded at the corresponding actual arguments).

A slicing traversal beginning inside a procedure proceeds from formals to the corresponding actuals at *all* call sites (which themselves appear inside procedures on the call chain), because actuals to any of those calls could affect (an instance of) the slicing criterion value.

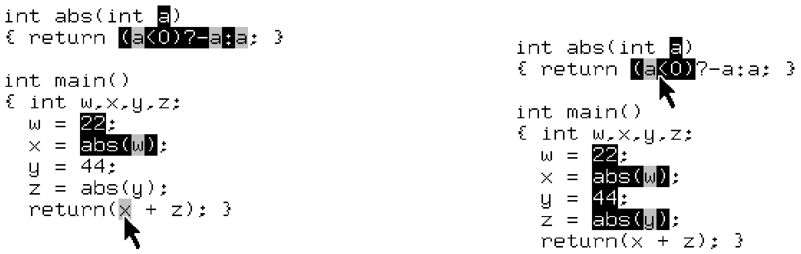


Figure 8: A slice including a particular call to `abs`, as on the left, contains actuals from that call and elements of the procedure body. A slice that starts inside the function body, as on the right, includes both of the calls to the function, and their actuals, since any of those actual parameters can affect the run-time value of the expression in the function body.

This algorithm runs in time linear in the number of values computed by the program, since each edge need be traversed at most once for each component of the value it carries. Thus, the asymptotic complexity is the same as that of the naive slicing algorithm which does not account for calling context. Bookkeeping details such as depth limiting, direct summary dependences, and aggregate values complicate the code but do not affect its algorithmic complexity.

We present several improvements on previous work on interprocedural slicing. We give a faster (by an order of at least $O(n)$), clearer algorithm for computing summary dependences. In the absence of aliasing, our slicing algorithm works in time linear in the size of the original program, compared to quadratic time for previous algorithms; aliasing degrades our method less than other methods. Our slices exclude more procedure calls and bodies by distinguishing between direct and indirect summary dependences. We produce better executable slices by providing exact liveness information (including demand on call results) to the code generator [Ern94a, Ern94b].

4.1 Summary dependences

Precomputed summary dependences for a procedure indicate, for each return value, which formal parameters it depends on. They permit a slicing traversal which encounters a procedure call result to proceed at the demanded actual parameters of the call without entering the procedure body.

The dependence summary is precomputed for all functions simultaneously by finding a fixed point solution for the dataflow equations of Figure 9. A solution to the equations consists of FORMALS(v) for every value v in the program—indicating which formal parameters v depends on—and SUMMARY(ret) for every return node ret in the program—indicating which formal parameters ret depends on. The rules can be understood as follows:

- A return node depends on whatever formals its input depends on.
- A formal node's output depends only on that formal.
- Except for return, formal, and call nodes, a node's output value depends on every formal depended on by any of its inputs.
- The dependences of a call result *res* are determined in two steps. First, let F be the formals of the called procedure which are depended on by the formal return corresponding to *res*. Then, *res* depends on every formal depended on by an actual parameter corresponding to a formal $f \in F$. When calling through a function pointer, this procedure is followed for every possible callee (as indicated by the points-to analysis) and the results are unioned together.

An efficient optimistic analysis solves the dataflow equations of Figure 9. All FORMALS and SUMMARY properties are initialized to the empty set, then each node in the program is placed on a

Notation:

$$
\begin{aligned}
\textsc{summary}(ret) &= \text{the formals of return node } ret\text{'s procedure on which } ret \text{ depends}\\
\textsc{formals}(v) &= \text{the formals of value } v\text{'s enclosing procedure on which } v \text{ depends}\\
in(n) &= \text{the inputs of node } n\\
out(n) &= \text{the outputs of node } n\\
\text{callees}(c) &= \text{procedures that may be called by call node } c\\
ret(res, p) &= \text{the formal return in procedure } p \text{ corresponding to result } res \text{ of a call to } p\\
act(f, c) &= \text{the actual parameter to call node } c \text{ corresponding to formal } f \text{ of a callee of } c
\end{aligned}
$$

Dataflow equations for node n:

return node: $\textsc{summary}(n) = \textsc{formals}(in(n))$

formal node: $\textsc{formals}(out(n)) = \{n\}$

call node: for each $res \in out(n)$

$$\textsc{formals}(res) = \bigcup_{p \in \text{callees}(n)} \; \bigcup_{f \in \textsc{summary}(ret(res,p))} \textsc{formals}(act(f, n))$$

otherwise: $\textsc{formals}(out(n)) = \bigcup_{i \in in(n)} \textsc{formals}(i)$

Figure 9: Dataflow equations for computing dependences on formal parameters. The equations use *in*, *out* as a singleton when the set is known to contain just one element. For simplicity, the modifications to support aggregate values, direct dependences, and depth-limiting are not shown in the figure.

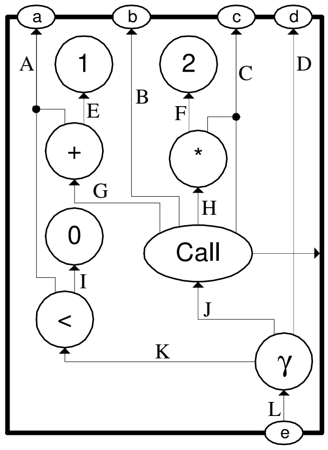
worklist. Nodes are removed from the worklist one at a time and processed. When a new SUMMARY property (different from the previous approximation) is computed, all callers of the procedure are placed on the worklist. When a new FORMALS property is computed for a value, its consumers are placed on the worklist. When the worklist is empty, the least fixed point solution has been computed.

Summary dependences for codeless procedures are looked up in a database which is filled in when the function is analyzed (in a separate compilation phase) or by hand (for library routines whose text is unavailable); pessimal assumptions are made for procedures not in the database. Free values defined outside a procedure but used inside without being passed in parameters (such as the use of `n` in the loop of Figure 1) can be either addressed directly or converted into parameters [Ern94b]. For efficiency, VDGomatic does not propagate whole FORMALS properties, but only differences from previous values.

Aggregate values such as structures and stores require two modifications to the dataflow equations. First, dependence on part of an aggregate formal does not imply dependence on all of the formal. Therefore, each element of the FORMALS and SUMMARY sets may be either a scalar formal or an aggregate formal and a location or member specifier. Second, different parts of an aggregate may depend on different formals. The FORMALS and SUMMARY properties are extended to be either a set (for scalar values) or a tuple of sets (one set for each element of an aggregate value). These tuples are combined elementwise when the equations call for a union operation. The dependence summary for the return value of procedure `foo` in Figure 6 is $\langle\{a.slot2\}, \{a.slot1, b\}\rangle$.

VDGomatic also marks each dependence on a formal as direct or indirect. A direct dependence indicates that the formal is passed through unchanged (and that no other formal is depended upon).

```
int foo(int a, int b, int c, int d);
{ if (a < 0)
    return(foo(a+1, b, c*2, c));
  else
    return(d);
}
```



Node	New value	Reprocess
a	FORMALS(A) $= \{a\}$	+,<
b	FORMALS(B) $= \{b\}$	call
c	FORMALS(C) $= \{c\}$	*,call
d	FORMALS(D) $= \{d\}$	γ
e	SUMMARY(e) $= \{\}$	
1	FORMALS(E) $= \{\}$	+
2	FORMALS(F) $= \{\}$	*
+	FORMALS(G) $= \{a\} \cup \{\} = \{a\}$	call
*	FORMALS(H) $= \{\} \cup \{c\} = \{c\}$	call
0	FORMALS(I) $= \{\}$	<
call	FORMALS(J) $= \{\}$	
<	FORMALS(K) $= \{a\} \cup \{\} = \{a\}$	γ
γ	FORMALS(L) $= \{a\} \cup \{\} \cup \{d\} = \{a, d\}$	e
e	SUMMARY(e) $= \{a, d\}$	call
call	FORMALS(J) $= \{a\} \cup \{c\} = \{a, c\}$	γ
γ	FORMALS(L) $= \{a\} \cup \{c\} \cup \{d\} = \{a, c, d\}$	e
e	SUMMARY(e) $= \{a, c, d\}$	call
call	FORMALS(J) $= \{a\} \cup \{c\} \cup \{c\} = \{a, c\}$	

Figure 10: Computing summary dependences. VDG values have been labeled with uppercase letters. The table steps through the algorithm, assuming the initial worklist order is a,b,c,d,e,1,2,+,*,0,call,<,γ. Nodes needing to be reprocessed are added to the worklist, if they are not already there; the last five lines of the table reprocess previously encountered nodes.

An indirect dependence indicates that the formal may participate in computation to determine the return value. Formal outputs depend directly on the formal. Propagation through any other node makes a dependence indirect (e.g., the result of an addition depends *indirectly* on any formal depended on by any of the summands), with three exceptions: an assignment to a known location preserves directness, dependences on unaffected aggregate components are not made indirect, and a call preserves direct dependences when the corresponding return depends directly on a formal. This enhancement does not affect the algorithmic complexity, only the implementation complexity.

When a return value depends directly on a formal, VDGomatic can omit or include the procedure body from slices that include the corresponding call result. Omitting the body more clearly indicates an expression's value and what computations it depends on, while including elements of the procedure body indicates both computations depended upon and how the value was copied among storage locations. Previous slicers support only the latter view, which appears in Figure 11.

```
main(int a, int b)
{
  int sum = swapsum(&a, &b);
  printf("%d %d %d", a, b, sum);
}

swapsum(int *x, int *y)
{
  int z;
  z = *x;
  *x = *y;
  *y = z;
  z = (*x) + (*y);
  return(z);
}
```

```
typedef struct { int slot1; int slot2; } Twoslots;

Twoslots foo(Twoslots a, int b)
{
  Twoslots r;
  r.slot1 = a.slot2;
  r.slot2 = a.slot1 * b;
  return r;
}

int main()
{
  Twoslots y = { 22, 23 };
  Twoslots x = foo(y, 44);
  return x.slot1 + x.slot2;
}
```

Figure 11: Display of direct dependences due to procedure calls. These figures repeat the slices on the left-hand sides of Figures 12 and 6, but with code passing values directly through procedures highlighted.

```
main(int a, int b)
{
  int sum = swapsum(&a, &b);
  printf("%d %d %d", a, b, sum);
}

swapsum(int *x, int *y)
{
  int z;
  z = *x;
  *x = *y;
  *y = z;
  z = (*x) + (*y);
  return(z);
}
```

```
main(int a, int b)
{
  int sum = swapsum(&a, &b);
  printf("%d %d %d", a, b, sum);
}

swapsum(int *x, int *y)
{
  int z;
  z = *x;
  *x = *y;
  *y = z;
  z = (*x) + (*y);
  return(z);
}
```

Figure 12: Interprocedural slicing with pointers. Jiang's implementation [JZR91, §3.2.2] of Horwitz's algorithm [HRB90] includes the entire program when slicing on `a` or `sum`; our slices are shown above. Note that there is no dependence on any variable addresses.

4.1.1 Complexity

The summary algorithm works by processing nodes and propagating new information along edges to other nodes. Because each edge traversal adds to the set of dependences (if there is no new information, the traversal is stopped), the algorithm crosses each scalar VDG edge at most f times, where f is the combined number of formals and free values for the enclosing procedure. Edges carrying aggregate values can be traversed up to f times for each value carried by the edge, or up to $O(n^2)$ times, since both f and the number of aggregate components are bounded by n. The VDG contains $O(n)$ edges, so the total number of edge traversals is $O(n^3)$. This result can also be derived by replacing each aggregate edge with an appropriate number of scalar edges, resulting in a total of $O(n^2)$ VDG edges, each of which can be traversed up to f times.

The assertion that a VDG contains $O(n^2)$ edge components, where aggregates and stores are counted once for each element they contain, is easily justified for an unoptimized VDG. (An unoptimized VDG also contains $O(n)$ edges and nodes. While optimization can increase the number of nodes and edges in the VDG, it does not increase the number of edge components, which is a more accurate measure of VDG size [Ern94b].) Each variable reference becomes a store lookup, and each assignment is represented by an update-store node. No value is used by more than one consumer. Each program operation becomes (a constant number of) VDG nodes. The machine state (global store, I/O streams, and so forth) is threaded through each computation; these values appear as formals and returns of each procedure, in addition to user-specified formals and returns. The only free values are program constants, such as the addresses of global variables. Such a VDG contains $O(n)$ nodes and $O(n)$ edges, and each procedure has $O(n)$ formals and free values.

Each node is processed at most once for each time the slicing traversal reaches it. Node processing and computation of differences from previous results costs $O(n)$ for each component with modified dependences. (Storing summary dependences per formal as well as per return can prevent processing nodes when no output dependences would be changed.) Therefore, the total theoretical cost of the algorithm is $O(n^4)$, though in practice most edges are traversed no more than a few times, and few values have $O(n)$ components or depend on $O(n)$ formals.

4.2 Related work

Horwitz, Reps, and Binkley [HRB90] compute summary dependences from an attribute grammar that models procedure-call structure and from the subordinate characteristic graphs of the grammar's nonterminals. If n is the size of the original program, constructing the subordinate characteristic graphs requires $O(n^5)$ computation [HRB90, §5.1], and the entire summary algorithm performs $O(n^7)$ steps [RHSR94, §4]. An improved algorithm [RHSR94] finds same-level realizable paths in the graph that end at a procedure's formal-out (return) vertices, which induce summary edges at call sites. The algorithm can extend paths up to $O(n^4)$ times, and each path extension incurs set operations on sets of size $O(n)$. Livadas and Croll [LC92] suggest incrementally constructing and refining an "extended call sequence graph." They have not implemented the technique (Ghinsu handles function calls via inlining [LR94]) or determined its complexity (they do note that it multiply processes looping statements).

These methods represent dependences due to calls via a system dependence graph (SDG). After summary computation, slicing is linear in the size of the SDG, which (in the absence of aliasing) is quadratic in the size of the original program. The blowup in size results from the numerous special nodes and edges added to model procedure calls, and from the representation of dependence summaries at every call site instead of just at the procedure definition. By contrast, an alias-free interprocedural VDG is linear in the size of the program, so the slicing algorithm remains linear-time, and computing summaries for such a procedure takes at most $O(n^3)$ time. Support for arbitrary pointer manipulations (see Section 3) increases VDG size to quadratic; support for call-by-reference parameters either increases the SDG size to exponential or results in degraded performance.

The NCTU slicer [HDC88] computes summaries via an exponential time [HRB90] technique. At each call, the called procedure is processed and the results propagated back to that call site. The method is made to terminate in the presence of recursion by computing a series of slices in which the recursion depth is limited: initially no recursive calls are followed, then one level, and so forth, until no change in results occurs.

5 Implementation

VDGomatic was first operational in late 1993 as part of the Zaphod environment for program understanding and debugging being built at Microsoft Research. It is written in Scheme and integrated with GNU Emacs for input and output and implements the techniques of this paper.

Currently the system can accommodate only programs of modest size. Creating a 13,000-node VDG from a 7,000-line program (the Free Software Foundation's `bc` calculator) takes approximately 150 seconds on an Iris Indigo 2 (MIPS R4400 processor). Computing dependence summaries takes under 30 seconds, slicing requires less than a second, and display of slices takes a second or two, depending on the size of the slice. (The full paper will contain more extensive timings.) The Zaphod system has not been optimized for speed and many obvious improvements are possible; for instance, VDGomatic incurs an $O(n)$ penalty by using an analysis framework rather than directly implementing traversals and value combination.

It is difficult to quantitatively compare our slicing results with previous ones. Previous work has not included hard results on timings or slice sizes, and in any event numbers for different program representations would be difficult to compare. Additionally, static slices vary greatly in size. slices on program results can be large, since interesting results tend to depend directly or indirectly on most of the previous computation, but other slices can be quite small. While we do not present slice size statistics here, future work will include the creation of a slicing test suite (for both closure and executable slicing) and measurements of our system on those test cases.

Acknowledgements

Daniel Weise, Ellen Spertus, Dave Binkley, Daniel Jackson, Bjarne Steensgaard, Todd Knoblock, Erik Ruf, Ted Biggerstaff, and Roger Crew. provided stimulating discussions and helpful comments on drafts of this paper. The Program Analysis group at Microsoft Research (Crew, Ernst, Ruf, Spertus, Steensgaard, and Weise) created the Zaphod programming environment, of which VDGomatic is one component. Ellen Spertus and Roger Crew independently suggested the name VDGomatic, which is pronounced "vegomatic."

References

[Agr91] H. Agrawal. Towards automatic debugging of computer programs. Technical Report SERC-TR-103-P, Software Engineering Research Center, Purdue University, September 1991.

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, White Plains, NY, June 20-22, 1990.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, NY, June 20-22, 1990.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256. ACM Press, June 1994.

[Ern94a] Michael D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 26, 1994.

[Ern94b] Michael D. Ernst. Slicing the value dependence graph. Technical Report MSR-TR-94-22, Microsoft Research, Redmond, WA, November 1994.

[HDC88] J. C. Hwang, M. W. Du, and C. R. Chou. Finding program slices for recursive procedures. In *Proceedings COMPSAC 88: The Twelfth International Computer Software and Applications Conference*, Chicago, October 1988. IEEE Computer Society.

[HEGV93] Laurie J. Hendren, Maryam Emami, Rakesh Ghiya, and Clark Verbrugge. A practical context-sensitive interprocedural alias analysis framework for C compilers. ACAPS Technical Memo 72, McGill University School of Computer Science, Advanced Compilers, Architectures, and Parallel Systems Group, Montreal, Quebec, July 24, 1993.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[JZR91] Jingyue Jiang, Xiling Zhou, and David J. Robson. Program slicing for C — the problems in implementation. In *Proceedings, Conference on Software Maintenance 1991*, pages 182–190, Sorrento, Italy, October 15-17, 1991. IEEE Computer Society Press.

[LC92] Panos E. Livadas and Stephen Croll. Program slicing. Technical Report SERC-TR-61-F, Computer and Information Sciences Department, University of Florida, Gainesville, FL, October 1992.

[LR94] Panos E. Livadas and Adam Rosenstein. Slicing in the presence of pointer variables. Technical Report SERC-TR-74-F, Computer and Information Sciences Department, University of Florida, Gainesville, FL, June 1994.

[LRZ93] William A. Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67. ACM Press, June 1993.

[OO84] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pages 177–184, Pittsburgh, Pennsylvania, April 1984.

[RHSR94] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. Submitted for publication, June 1994.

[Ruf94a] Erik Ruf. Context-insensitive alias analysis reconsidered. Submitted for publication, November 1994.

[Ruf94b] Erik Ruf. Optimizing sparse representations for dataflow analysis. Submitted for publication, September 1994.

[Ste93] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, Redmond, WA, August 1993.

[Ste94a] Bjarne Steensgaard. Points-to analysis in almost linear time. Submitted for publication, November 1994.

[Ste94b] Bjarne Steensgaard. Sparse functional stores for imperative programs. Manuscript, October 1994.

[SW93] Bjarne Steensgaard and Daniel Weise. A sparse representation for programs with pointers. Manuscript, November 1993.

[Ven93] G. A. Venkatesh. Experimental results from slicing C programs. Submitted for publication, October 19, 1993.

[WCES94a] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings of the Twenty First Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310, Portland, OR, January 1994.

[WCES94b] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. Technical Report MSR-TR-94-03, Microsoft Research, Redmond, WA, April 13, 1994.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.