Slicing Programs with Arbitrary Control Flow

THOMAS BALL* SUSAN HORWITZ*
tom@cs.wisc.edu horwitz@cs.wisc.edu
Computer Sciences Department/University of Wisconsin/1210 W. Dayton St./Madison, WI 53706/USA

Abstract

Program slicing is a program transformation that is useful in program debugging, program maintenance, and other applications that involve understanding program behavior. Given a program point *p* and a set of variables *V*, the goal of slicing is to create a *projection* of the program (by eliminating some statements), such that the projection and the original program compute the same values for all variables in *V* at point *p*.

This paper addresses the problem of slicing programs with arbitrary control flow. Previous slicing algorithms do not always form semantically correct program projections when applied to such programs. We give the first algorithm for slicing programs with complex control flow and a proof of its correctness. Our algorithm works for programs with completely arbitrary control flow, including irreducible control flow.

1. INTRODUCTION

Program slicing, a program transformation originally defined by Mark Weiser [12], is useful in program debugging [9], program maintenance [6], and other applications that involve understanding program behavior [8]. Given a program point *p* and a set of variables *V*, the goal of slicing is to create a *projection* of the program (by eliminating some statements), such that the projection and the original program compute the same values for all variables in *V* at point *p*.

Example. The program shown in Figure 1(a) computes the sum and product of the numbers from 1 to *N*[1]. Figure 1(b) shows the result of slicing the example program with respect to the statement **output**(*prod*) and the variable *prod*. For any value of *N*, the example program and the program projection compute the same value for variable

(a) Example Program	(b) Result of slicing with respect to **output**(*prod*)

begin
  sum := 0
  prod := 1
  i := 0
  while *i* < *N* **do**
    i := *i*+1
    sum := *sum*+*i*
    prod := *prod***i*
  od
  output(*sum*)
  output(*prod*)
end
	begin
	
	  prod := 1
	  i := 0
	  while *i* < *N* **do**
	    i := *i*+1
	
	    prod := *prod***i*
	  od
	
	  output(*prod*)
	end

Figure 1. An example program, and the result of slicing with respect to **output**(*prod*).

* This work was supported in part by the National Science Foundation under grant CCR-8958530, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from Xerox and 3M.

[1]In the example program, variable *N* is used without being explicitly initialized. It is assumed that such variables get their values from an initial state on which the program is executed.

From: Proc. of the 1st International Workshop on Automated and Algorithmic Debugging (1993): LNCS Vol 749

prod in the **output** statement. □

This paper addresses a problem that has not been discussed in the literature on program slicing [12[10, 11]7]. The problem is how to slice programs with unstructured control flow, *i.e.*, programs that include constructs such as **break**, **continue**, and **goto**. Previous algorithms do not slice such programs correctly. We give a program-slicing algorithm that correctly handles such programs, and we prove that the program projections produced by our algorithm meet the semantic goal of program slicing: Both the original program and the projection compute the same values at the point of the slice. Our algorithm works for programs with completely arbitrary control flow, including irreducible control flow [1]. (As given here, our algorithm has a slight technical restriction: A program is sliced with respect to a point *p* and the set of variables used or defined at *p* rather than an arbitrary set of variables. Extending the algorithm is straightforward.)

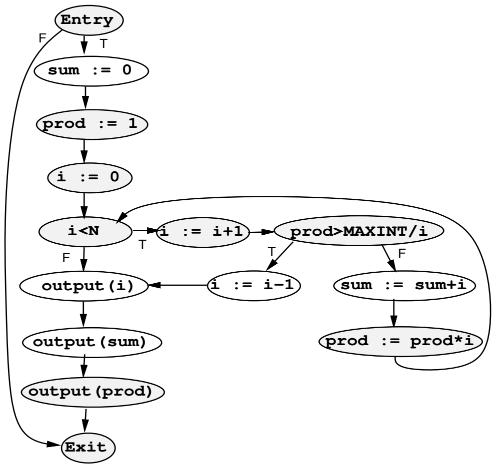
Algorithms for slicing programs with *structured* control flow have been defined by Weiser [12] and by the Ottensteins [10]. Neither of these algorithms works correctly for programs with unstructured control flow. We focus on the Ottensteins' algorithm and consider the problems that arise if one tries to apply this algorithm to programs with unstructured control flow. The Ottensteins' algorithm makes use of two program representations: the control flow graph [1] and the program dependence graph [4]. Vertices in the control flow graph represent statements and predicates in the program and edges represent the flow of control in the program. The program dependence graph has the same vertex set as the control flow graph. Its edge set is defined in terms of relationships in the control flow graph (control dependences and flow dependences). There are two steps to the algorithm. In Step 1, edges are traversed in the reverse direction in the program dependence graph from the vertex corresponding to program point *p* (we refer to this step as *backwards-closure*), identifying a set of vertices *S* in the program dependence graph. Step 2 produces the program projection by eliminating from the original program all components that do not correspond to a vertex in *S*.

If one uses the standard control flow graph for programs with unstructured control flow (*i.e.*, a graph in which a jump gives rise to a single control flow edge—see Figure 2(b)), the program projections computed by the Ottensteins' algorithm may fail to meet the semantic goal of program slicing; that is, the projections may compute different values than the original program at the point of the slice. The problem is that the algorithm does not correctly detect when unconditional jumps in the program (such as the **break**) are required in the program projection in order to meet the semantic goal of slicing.

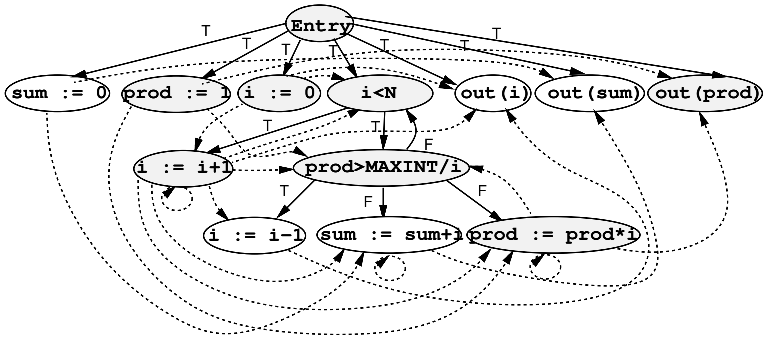
Example. Consider the program shown in Figure 2(a). Figure 2(b) shows the standard control flow graph for this program, and Figure 2(c) shows the program dependence graph of this control flow graph. In the two graphs, shading is used to indicate the vertices that would be identified by backwards-closure in the program dependence graph with respect to "**output**(*prod*)". Figure 2(d) shows the program projection obtained by eliminating all components not identified by the slicing algorithm. Because the **break** statement is absent from the projection, it does not satisfy the semantic goal (*i.e.*, for some values of *N* and *MAXINT*, different final values of *prod* will be output by the original program and by the projection). A projection that does satisfy the semantic goal (and that would be produced by the slicing algorithm defined in this paper) is shown in Figure 2(e). □

```
begin
  sum := 0
  prod := 1
  i := 0
  while i < N do
     i := i+1
     if prod > MAXINT/i then
        i := i−1
        break
     fi
     sum := sum+i
     prod := prod*i
  od
  output(i)
  output(sum)
  output(prod)
end
```

(a) Example program



(b) Control flow graph



(c) Program dependence graph

```
begin
  prod := 1
  i := 0
  while i < N do
     i := i+1
     if prod > MAXINT/i then
     fi
     prod := prod*i
  od
  output(prod)
end
```

(d) Incorrect projection

```
begin
  prod := 1
  i := 0
  while i < N do
     i := i+1
     if prod > MAXINT/i then
        break
     fi
     prod := prod*i
  od
  output(prod)
end
```

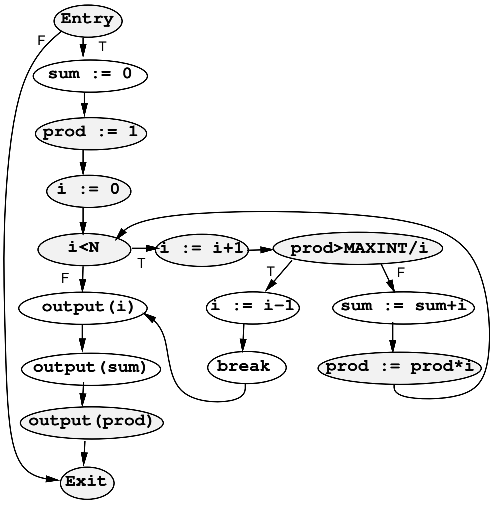
(e) Correct projection

Figure 2. An example program, its control flow graph, its program dependence graph (solid arrows are control dependences, dashed arrows are flow dependences), the (incorrect) projection that would be computed using the Ottensteins' algorithm to slice with respect to **output**(*prod*), and the correct projection.

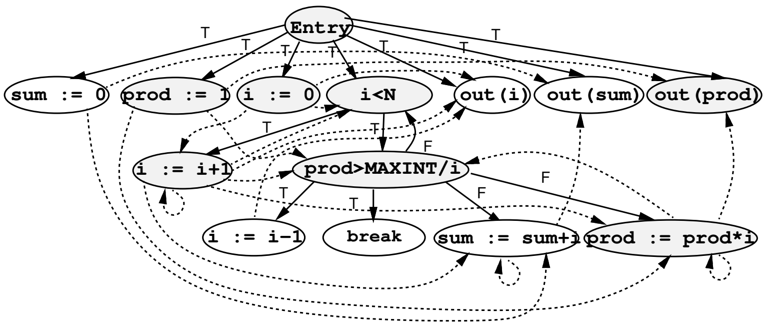
Simply including a vertex for the **break** in the control flow graph, as shown in Figure 3(a), does not solve the problem. The **break** will still be omitted from the slice because in the program dependence graph—shown in Figure 3(b)—there is no path from the **break** vertex to the vertex that represents the statement "**output**(*prod*)"; in fact, the **break** vertex has *no* outgoing edges, so it will not be included in any slice other than the slice with respect to the **break** itself.

The main result of this paper is a slicing algorithm for programs with unstructured control flow, and a proof of the correctness of this algorithm; that is, we show that the program projections produced by the algorithm have the desired semantic property: Both the original program and the projection compute the same values at the point of the slice. The algorithm is in the style of the Ottensteins' algorithm in that it operates on a program dependence graph representation of a program; however, the program dependence graph is based on a control flow graph in which a jump is represented as a pseudo-predicate vertex (that always evaluates to *true*). The jump vertex's *true*-successor is the target of the jump, and its *false*-successor is the vertex that represents the jump statement's continuation (that is, the vertex that would be the jump vertex's successor if it were a "no-op" rather than a jump).

We are able to prove that by using this augmented control flow graph, a projection of the program that has the desired semantic property can be formed. Furthermore, for large classes of programs with complex control flow, we can show that the resulting slices are the smallest slices that can be formed by backwards-closure in the program dependence graph (see Section 4.1).



(a) Control flow graph including **break**



(b) Corresponding program dependence graph

Figure 3. The same set of vertices is identified whether or not a **break** vertex is included in the control flow graph.

From: Proc. of the 1st International Workshop on Automated and Algorithmic Debugging (1993): LNCS Vol 749

The remainder of the paper is organized as follows. Section 2 provides background material, including a discussion of the language under consideration, and the definitions of control flow graph and program dependence graph. Section 3 presents our slicing algorithm and gives an outline of its proof of correctness (a full proof can be found in [3]). Section 4 discusses the issues of *minimal* slices, and of the extensibility of our slicing algorithm. Section 5 summarizes our results.

2. BACKGROUND

2.1. The Language Under Consideration

To simplify our presentation and focus on the problem of slicing with arbitrary control flow, we consider a simplified language with the following characteristics: Expressions contain only scalar variables and constants; statements are either assignment statements, jump statements (*e.g.*, **break, halt, goto**), output statements, conditional statements (**if-then** or **if-then-else**), or loops (**while** and **repeat**). It is easy to generalize our techniques to handle languages with *N*-way branch constructs, such as **case** statements, and other looping constructs. The problems of slicing in the presence of multiple procedures, non-scalar variables, and dynamic control flow are orthogonal to the problem discussed here.

2.2. The Control Flow Graph and Its Semantics

In this section we define the control flow graph and its execution semantics. We also discuss the standard translation from a program to its control flow graph. In Section 3.1, we discuss the *augmented* translation that we use as the basis for our slicing algorithm.

A *control flow graph* (CFG) is any directed, rooted graph[2] that satisfies the following conditions. The CFG has three types of vertices: Statement vertices (either assignment statements or output statements), which have one successor; predicate vertices, which have one *true*-successor and one *false*-successor; and an *EXIT* vertex, which has no successors. The root of the CFG is the *ENTRY* vertex, which is a predicate that has the *EXIT* vertex as its *false*-successor. Every vertex is reachable from the *ENTRY* vertex, and the *EXIT* vertex is reachable from every vertex. Edges in the CFG are labeled; the outgoing edges of a predicate vertex are labeled *true* or *false* (as appropriate) and the outgoing edge of a statement vertex is labeled *null*.

We assume that the reader is familiar with the standard control flow translations of the control constructs specified in Section 2.1 [1]. In the standard translation from a program to a CFG, the CFG includes a vertex for every assignment statement, output statement, and predicate in the program. The edges of the CFG represent the flow of control (the *ENTRY* vertex's *true*-successor is the first statement in the program). Jump statements (such as **break** and **goto**) are not represented directly as vertices in the CFG; instead, they are represented indirectly in that they affect the flow of control, and therefore the targets of some CFG edges.

---

[2]A *directed graph G* consists of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$, where $E(G) \subseteq V(G) \times V(G)$. Each edge $(b, c) \in E(G)$ is directed from b to c ($b \rightarrow c$); we say that b is the *source* and c the *target* of the edge.

Example. Figure 2(b) shows the CFG of the program in Figure2(a). □

The operational semantics for the CFG is defined as follows: Execution starts at the *ENTRY* vertex (which always evaluates to *true*), with an initial state σ; at any moment there is a single point of control together with a state mapping variables to values; the execution of each statement or predicate vertex passes control to a single successor. The execution of an assignment statement changes the state. Execution terminates normally if *EXIT* is reached (execution can fail to terminate normally if the program includes an infinite loop or an exception such as division by zero). An execution of CFG G on initial state σ is denoted by $G(\sigma)$.

For an execution $G(\sigma)$, we characterize the *behavior* at a vertex by the sequence of values that arise at that vertex[3]. This is defined as follows: For an assignment statement vertex, the sequence of values assigned to the left-hand-side variable; for an output statement, the sequence of values output; and for a predicate vertex, the sequence of boolean values to which the predicate's boolean expression evaluates. $G(\sigma)(v)$ denotes the sequence of values that arise at vertex v in execution $G(\sigma)$.

The following definition defines what we mean for a vertex in CFG H to replicate the behavior of a vertex in CFG G. Because a CFG includes a vertex for every program component, this definition also makes precise the semantic goal of slicing a program with respect to component c: To create a projection whose behavior at c replicates that of the original program at c. In examining this definition, it is useful to think of CFG H as a slice of CFG G (the vertices in CFG H are a subset of the vertices in CFG G). Because H may not contain vertices present in G, it is impossible to require conditions such as "for all σ, if $H(\sigma)$ terminates normally then $G(\sigma)$ terminates normally". This leads to the asymmetry in the definition.

DEFINITION. Vertex v_H of CFG H *replicates the behavior* of vertex v_G of CFG G iff all the following hold:

- For all σ such that $G(\sigma)$ terminates normally, $H(\sigma)$ terminates normally and $G(\sigma)(v_G) = H(\sigma)(v_H)$.
- For all σ such that $G(\sigma)$ does not terminate normally, $G(\sigma)(v_G)$ is a prefix of $H(\sigma)(v_H)$.[4]

2.3. The Program Dependence Graph

The program dependence graph (PDG) is defined in terms of a program's control flow graph. The PDG includes the same set of vertices as the CFG, excluding the *EXIT* vertex. The edges of the PDG represent the *control* and *flow* dependences induced by the CFG as follows[5]:

---

[3]Note that our definition of a vertex's behavior differs from the more standard definition, which characterizes a vertex's behavior as the sequence of *states* that arise at that vertex during an execution (where a state associates a value with *every* variable in the program). The standard definition is not suitable for program slicing, since the semantic goal of slicing is *not* to preserve the values of all variables, but only to preserve the values of the variables used or defined at the point with respect to which the slice is taken.

[4]$H(\sigma)$ may or may not terminate normally in this case.

[5] In addition to control and flow dependences, program dependence graphs usually include either def-order dependences [5] or output and anti-dependences [4]. These additional edges are not needed for program slicing, and so are omitted from the definition given here. We also do not need to distinguish between loop-independent and loop-carried dependences (which are ill-defined for irreducible control flow).

DEFINITION (postdominance). Let v and w be vertices in a CFG. Vertex w *postdominates* vertex v iff $w \neq v$ and w is on every path from v to the *EXIT* vertex. Vertex w *postdominates* the L-branch of predicate vertex v (where L is either *true* or *false*) iff w is the L-successor of v or w postdominates the L-successor of v. While no vertex can postdominate itself, a vertex can postdominate its own L-branch.

DEFINITION (control dependence). Let v and w be vertices in a CFG. Vertex w is directly L-control dependent on v (written $v \longrightarrow_c^L w$) iff w postdominates the L-branch of v and w does not postdominate v. A vertex can be directly control dependent on itself. Intuitively, if $v \longrightarrow_c^L w$, then whenever v executes and evaluates to L, w will eventually execute, barring abnormal termination. Furthermore, if v executes and does not evaluate to L, w might not execute. In this way, v directly controls whether or not w executes. Note that a vertex with only one successor in the CFG can never be the source of a control-dependence edge.

Under the standard definition, there is a *flow dependence* from vertex v to vertex w iff vertex v assigns to variable x, vertex w uses x, and there is a path in the CFG from v to w that does not include an assignment to x (excluding v and w). However, the augmented translation from programs to CFGs that will be introduced in Section 3 causes the CFG to include "dummy" edges; that is, edges that are never traversed in any execution (namely, the *false* edges out of vertices that represent jump statements). Therefore, we use a slightly modified definition of flow dependence:

DEFINITION (flow dependence). Let v and w be vertices in a CFG. There is a *flow dependence* from vertex v to vertex w (written $v \longrightarrow_f w$) iff vertex v assigns to variable x, vertex w uses x, and there is a path from v to w that does not include an assignment to x (excluding v and w) and that does not include any "dummy" edge.

Example. Figure 2(c) shows the PDG of the program in Figure2(a). Control-dependence edges are shown using solid arrows; flow dependence edges are shown using dashed arrows. □

3. SLICING PROGRAMS WITH ARBITRARY CONTROL FLOW

In this section we present our slicing algorithm and sketch a proof that it produces program projections with the desired semantic property: Given program P and component c, our algorithm, $Slice(P,c)$, produces a projection P' of P such that P' replicates the behavior of P at component c.

3.1. The Slicing Algorithm

Our slicing algorithm is similar to the Ottensteins' algorithm in that it uses a program dependence graph (PDG) to identify the program components in the slice. In particular, given a PDG and a vertex v from which to slice, Step 1 of both the Ottensteins' algorithm and our algorithm identifies the subset of the PDG's vertices from which there is a path along control and/or flow dependence edges to vertex v (*i.e.*, Step 1 computes the backwards reflexive transitive closure with respect to v). Step 2 of both algorithms creates a program projection by eliminating components that do not correspond to the vertices identified in Step 1[6].

The important difference between our algorithm and the Ottensteins' is that we use an *augmented* translation from the program to the control flow graph (CFG) from which the PDG is built. The translations for all the structured constructs (*i.e.*, **if-then, if-then-else, while**, and **repeat**) remain the same. However, jump statements are explicitly represented in the CFG as pseudo-predicate vertices that always evaluate to *true*. A jump vertex's *true*-successor is the target of the jump; its *false*-successor is the vertex that represents the jump statement's fall-through or continuation (that is, the vertex that would be the jump vertex's successor if it were a "no-op" rather than a jump). The outgoing false edge of a jump vertex is a "dummy" edge that is never actually traversed in an execution.[7] Representing a jump statement this way causes it to be the source of control dependence edges in the PDG. This in turn allows the jump vertex to be correctly included in the backwards-closure in the PDG.

Example. Figure 5(a) repeats the program of Figure 2(a) and shows the program's augmented CFG. Figure 5(b) shows the vertices, control edges, and some of the flow edges of the corresponding PDG (flow edges that are not relevant to the backwards-closure with respect to "**output**(*prod*)" are omitted).

Note that in this PDG, the **break** vertex has three outgoing control dependence edges (which are not in the PDG of Figure 2(c)). These edges are consistent with the intuition behind control dependence: Removing the **break** might change the number of times the assignments to *sum* and *prod* as well as the evaluation of the loop predicate were performed (and therefore there are control dependence edges from the **break** vertex to the vertices that represent these three components). However, in a terminating execution, the presence or absence of the **break** has no effect on whether or not statements outside the loop are executed (and therefore there are no control dependence edges from the **break** vertex to a vertex that represents a statement outside the loop).

In Figure 5(b), shading is used to indicate the PDG vertices that are identified by backwards-closure with respect to "**output**(*prod*)". Note that the shaded vertices correspond to the program components that are included in the correct program projection shown in Figure 2(e). □

3.2. Proof of Correctness

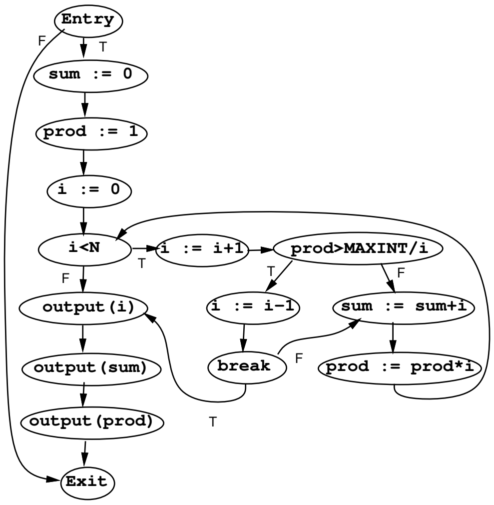
In this section we sketch a proof that our slicing algorithm produces a program projection with the desired semantic property. Because of space limitations, we are unable to give the full proof here. Details of the proof can be found in [3].

---

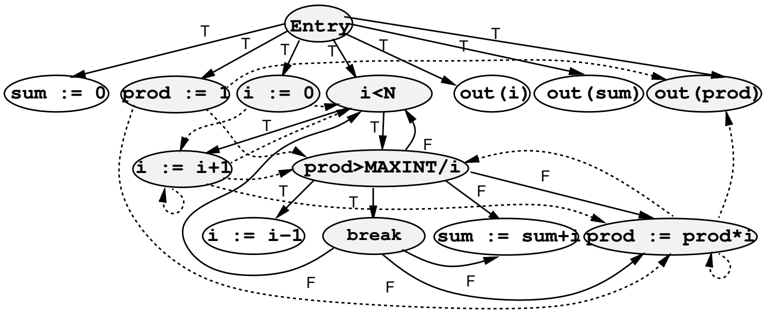
[6]A precise definition of what it means to eliminate a program component is given in [3]. In short, the elimination operation is defined in terms of the program's abstract-syntax tree; every vertex in the program's PDG corresponds to a node in the abstract-syntax tree. Eliminating the component that corresponds to vertex *v* means removing the subtree rooted at the tree node that corresponds to *v*.

[7] It is important to note that representing jump statements this way in the CFG does not change the semantics of the CFG as defined in Section 2.2. In particular, since a jump is treated as a predicate that always evaluates to *true*, and since the jump vertex's *true*-successor is the target of the jump, it is clear that for every vertex *v* in the standard CFG *G* and every initial state σ, the behavior at *v* when *G* is executed on σ is the same as the behavior at the corresponding vertex when the augmented CFG is executed on σ.

begin
  sum := 0
  prod := 1
  i := 0
  while *i* < *N* **do**
    i := *i*+1
    if *prod* > *MAXINT*/*i* **then**
      i := *i*−1
      break
    fi
    sum := *sum*+*i*
    prod := *prod**i*
  od
  output(*i*)
  output(*sum*)
  output(*prod*)
end



(a) Example program and its augmented CFG



(b) Corresponding PDG

Figure 5. The example program from Figure 2(a), its augmented CFG, and the corresponding PDG. Shading is used to indicate the PDG vertices identified by backwards-closure with respect to "**output**(*prod*)". Flow edges that are not relevant to the backwards-closure with respect to "**output**(*prod*)" are omitted.

A semantics-preserving transformation on CFGs

The first step of the proof is to show that eliminating the vertices not identified by backwards-closure in the PDG with respect to the slicing vertex (Step 1 of the algorithm) is a semantics-preserving transformation on CFGs.[8] In particular, we show that every vertex in the resulting CFG replicates the behavior of the corresponding vertex in the original CFG. This part of the proof does not rely at all on the augmented translation. That is, the results here are for arbitrary control flow graphs, irrespective of the program from which they were derived.

---

[8] To eliminate a vertex x from CFG G: For every vertex a such that there is an edge $a \rightarrow^{L1} x$ and for every vertex b such that there is an edge $x \rightarrow^{L2} b$, remove edges $a \rightarrow^{L1} x$ and $x \rightarrow^{L2} b$; add edge $a \rightarrow^{L1} b$. Remove vertex x.

Example. Figure 6 repeats the (augmented) CFG of Figure 5(a) and shows the CFG that results from eliminating the vertices not identified by backwards-closure with respect to "**output**(*prod*)". □

The proof that eliminating the vertices not identified by backwards-closure in the PDG is a semantics-preserving transformation relies on the following definitions and lemmas:

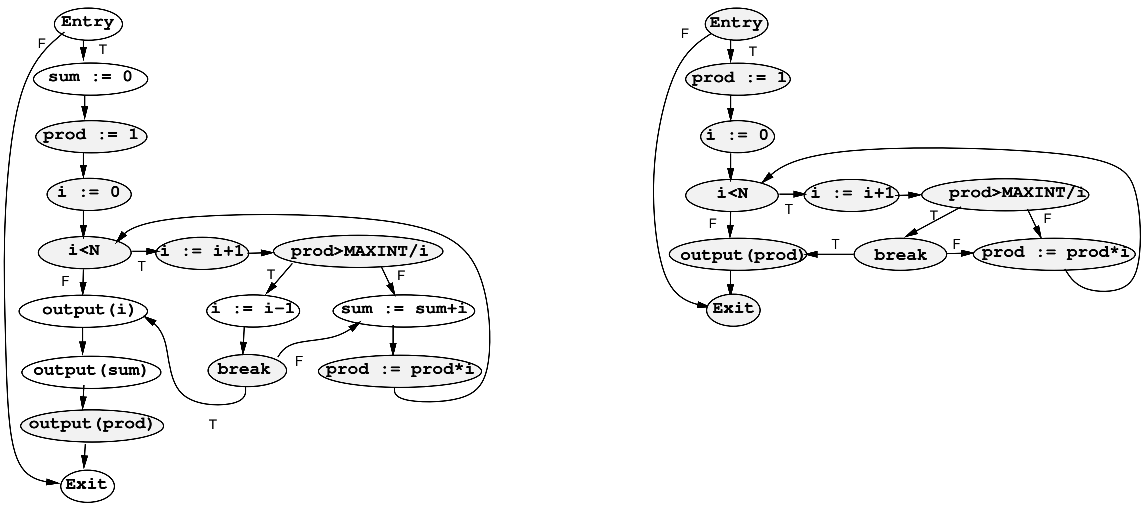
DEFINITION (path-projection). CFG H is a *path-projection* of CFG G iff all of the following hold:

(1) The vertices of H are a subset of the vertices of G.
(2) For every path in G (a sequence of [vertex, edge-label] pairs), if the vertices that are not in H are eliminated along with their outgoing edge labels, then the resulting sequence is a path in H.
(3) For every path *PTH* in H, there is a path in G whose projection is *PTH* (as in (2)).

DEFINITION (flow/path-projection). CFG H is a *flow/path-projection* of CFG G iff both of the following hold:

(1) H is a path-projection of G.
(2) For every vertex $w \in H$, if G induces the flow dependence $v \longrightarrow_f w$, then vertex v is also in H.

Example. Figure 7 shows four CFGs. Both H and J are path-projections of G; however, K is not. This is because G includes the path ((Entry, T)($x > 0$, F)($y := 0$, *null*)(**output**(y), *null*), (**output**(x), *null*)(Exit)), but the path ((Entry, T)(Exit)) is not in K. H is also a flow/path-projection of G, but J is not. This is because vertex "**output**(y)" is in J, graph G induces a flow dependence from "$y := 1$" to "**output**(y)", but vertex "$y := 1$" is not in J. □



(a) Original (augmented) CFG of the example program. Shading indicates the vertices identified by backwards-closure with respect to "**output**(*prod*)".

(b) After eliminating unshaded vertices.

Figure 6. Eliminating vertices not in the backwards-closure preserves CFG semantics.

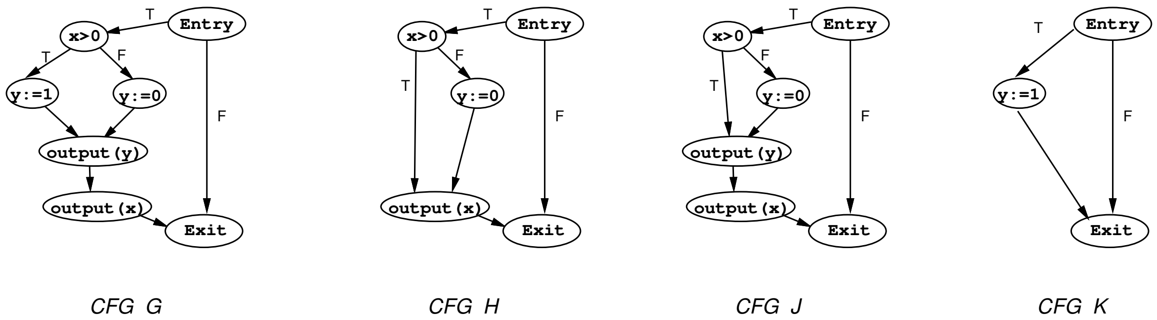


Figure 7. *H* and *J* are path-projections of *G*; *K* is not. *H* is also a flow/path-projection of *G*; *J* is not.

LEMMA (flow/path-projections preserve CFG semantics). If CFG *H* is a flow/path-projection of CFG *G*, then the behavior of every vertex in *H* replicates the behavior of the corresponding vertex in *G*.

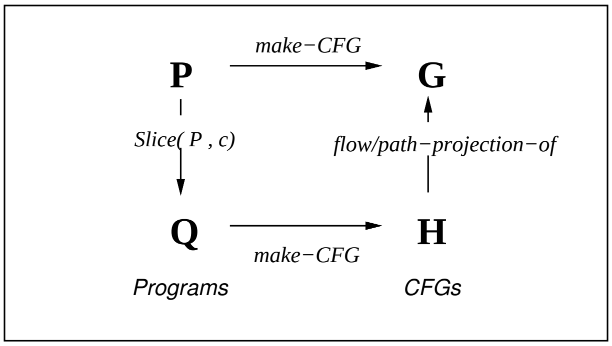
The following lemma shows that backwards-closure in the PDG finds the set of vertices necessary *and* sufficient to form a CFG that is a flow/path-projection of the original CFG.

LEMMA (backwards closure in PDG finds flow/path-projection). Given: CFG *G*, its PDG *D*, and vertex *v*. Eliminating the vertices from *G* that are not in the backwards closure of *D* with respect to *v* yields a CFG that is the minimal flow/path-projection of *G* that contains *v*.

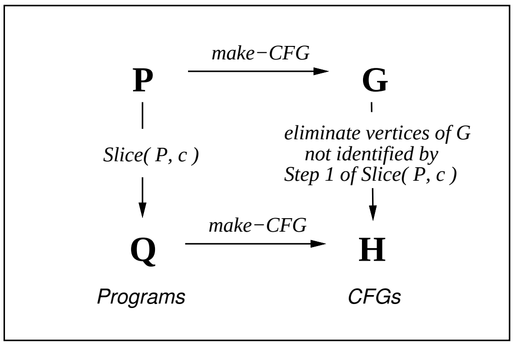
A semantics-preserving transformation on programs

Recall that the goal of program slicing is to produce a projection of a given *program*, not to produce a projection of a given CFG. As illustrated by the example of Figure 2, under the standard control flow translation, creating a program projection by eliminating components that do not correspond to the vertices identified by backwards-closure does *not* result in a projection with the desired semantic property.

The second part of the proof of correctness of our algorithm involves showing that under the augmented translation, eliminating program components that do not correspond to the vertices identified by backwards-closure *is* a semantics-preserving transformation on programs (Step 2 of the algorithm). To prove this, we have shown that the relationships pictured below hold (see [3] for details).



That is, given a program *P* and a component *c*, we show that the program *Q* that results from applying our slicing algorithm to *P* has a CFG *H* that is a flow/path-projection of *P*'s CFG *G*. By the results of the previous section, this means that the vertices of *H* (in particular, the vertex that corresponds to *c*) replicate the behaviors of the corresponding vertices of *G*. This part of the proof relies on several properties relating the abstract syntax of the language to the augmented control flow graph. Rather than argue directly that *H* is a flow/path-projection of *G*, we have shown that *H* can be obtained from *G* by eliminating all vertices not identified by backwards-closure (*i.e.*, we have shown that the diagram shown below commutes). It then follows from the results of the previous section that *H* is a flow/path-projection of *G*.



4. OTHER CONCERNS

This section address some interesting issues regarding slicing programs with arbitrary control flow.

4.1. Issues of minimality

A slicing algorithm identifies a program projection that behaves identically to the original program at some point of interest. As has been noted before, the usefulness of a slicing algorithm is inversely proportional to the size of the slices it produces. While it is an undecidable problem to find slices of minimal size, it would be possible to employ common compiler optimizations to further reduce the size of slices. For example, copy propagation could be used to prune away copy chains from a slice, as shown below (of course, some renaming may need to be done also):

x := x+1;	x := x+1;
y := x;	
z := y;	z := *x*;

We believe that smaller slices are useful, up to a point. In this paper, we have shown that our slicing algorithm based on the program dependence graph produces programs whose CFGs are flow/path-projections of the original program's CFG. That is, they preserve paths of the original program (modulo projection) and the flow of values between components. While these properties are useful for proving the semantic results, they also are intuitively appealing. A slice that does not preserve the paths in a program or the flow of values amongst its components may compute the same result as the original program, but does so in a different way than the programmer originally intended. Flow/path-projections retain the *structure* of the computation as well as its result.

We also have shown that backwards-closure in the PDG identifies the minimal set of vertices needed to form a flow/path-projection of a CFG (that includes a given vertex). In particular, control dependence identifies the vertices that must necessarily be included in a slice in order to form a path-projection. That is, if CFG H is a path-projection of CFG G, w is in H and $v \longrightarrow_c w$ is in G's PDG, then v must also be in H.

There are cases where a flow/path-projection that is minimal under the augmented translation is not minimal under the standard translation. In the example below, program Y is the projection that results from slicing program X with respect to A (using the augmented translation). However, under the standard translation, the control flow graphs of programs Y and Z are identical, but program Z is clearly smaller than program Y.

X	*Y*	*Z*
if P **then**	**if** P **then**	**if** P **then**
goto L;	**goto** L;	
M: A	M: A	A
fi	**fi**	**fi**
goto N;	**goto** N;	
L: B	L:	
goto M;	**goto** M;	
N: C	N:	

It is possible to show that for certain languages with limited unstructured control flow, our algorithm (using the augmented translation) produces programs that are minimal flow/path-projections with respect to the standard control flow translation. One example of such a language is one in which **goto**s may only pass control to the continuation of an enclosing control construct, be it a loop or conditional. Furthermore, as we explain in Section 4.3, if we drop the requirement that the resulting program be a projection of the original program then it is easy to construct programs that are minimal flow/path-projections with respect to the standard control flow translation.

4.2. Other control constructs

The language considered in this paper has arbitrary control flow, due to the inclusion of the **goto** statement. It also has looping and conditional constructs found in many languages. However, the question naturally arises: do the results of this paper extend to other control constructs, such as **for** loops and **switch** statements? As we have shown, the program dependence graph can be used to form flow/path-projections of completely arbitrary control flow graphs. However, to ensure that the program projection operation works correctly (see the commutative squares at the end of Section 3.2), control constructs must satisfy a few simple properties.

A looping construct must generate a vertex v in the control flow graph such that: (1) v passes control to the continuation of the loop construct (*i.e.*, there is a loop exit); (2) every vertex generated by the (abstract syntax) subtrees enclosed by the looping construct is reachable from v. A **for** loop meets these requirements. However, a construct such as **loop-forever** does not. Fortunately, it is usually possible to translate a construct so that by the addition of dummy vertices and edges, it meets the requirements. For example, a **loop-forever** construct can be treated as a **while** loop where the predicate is *true*. This results in a dummy vertex with an outgoing *false* edge that is not executable.

A selection construct must generate a vertex v such that: (1) every vertex generated by subtrees enclosed by the selection construct is reachable from v; (2) every subtree immediately enclosed by the selection construct passes control to the continuation of the selection construct; The translations of selection constructs such as **if-then**, **if-then-else**, and **switch** meet these requirements.

4.3. Alternative methods for slicing

We have defined the slice of a program to be a projection of that program. That is, the program slice must be formed by eliminating statements from the original program. Because one of the major applications of slicing is debugging, this is a natural restriction. Presenting the programmer with a slice that does not resemble the original program is clearly unsatisfactory.

It is certainly possible to construct programs that meet the semantic goal of slicing but are not program projections. For example, given a program P with standard CFG G, one could construct the minimal flow/path-projection of G with respect to some vertex (using backwards-closure in the PDG to identify the required vertices) and then synthesize a program from that CFG using a structuring algorithm such as Baker's [2]. However, in a language with unstructured control flow, there can be many programs with the same CFG. The program that results from such an approach may not be a projection of the original program, even though it meets the semantic goal (because its CFG is a flow/path-projection of the original program's CFG).

5. CONCLUSIONS

This paper has addressed the problem of slicing programs with arbitrary control flow. Previous slicing algorithms do not always form semantically correct program projections when applied to such programs. This is due to the fact that the algorithms do not detect when an unconditional jump such as a **break** is required in a projection. Our work solves this problem by using a program dependence graph defined using an augmented control flow graph that represents jumps as pseudo-predicates.

REFERENCES

1. Aho, A., Sethi, R., and Ullman, J., *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, MA (1986).
2. Baker, B., “An algorithm for structuring flowgraphs,” *JACM* **24**(1) pp. 98-120 (January 1977).
3. Ball, T. and Horwitz, S., “Slicing programs with arbitrary control flow,” Technical Report 1128, Department of Computer Sciences, University of Wisconsin—Madison (December 1992).
4. Ferrante, J., Ottenstein, K., and Warren, J., “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems* **9**(3) pp. 319-349 (July 1987).
5. Horwitz, S., Prins, J., and Reps, T., “On the adequacy of program dependence graphs for representing programs,” pp. 146-157 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), (1988).
6. Horwitz, S., Prins, J., and Reps, T., “Integrating non-interfering versions of programs,” *ACM Transactions on Programming Languages and Systems* **11**(3) pp. 345-387 (July 1989).
7. Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,” *ACM Transactions on Programming Languages and Systems* **12**(1) pp. 26-60 (January 1990).
8. Horwitz, S., “Identifying the semantic and textual differences between two versions of a program,” *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 1990), pp. 234-246 (June 1990).
9. Korel, B., “PELAS—Program error-locating assistant system,” *IEEE Transactions on Software Engineering* **SE-14**(9) pp. 1253-1260 (September 1988).
10. Ottenstein, K.J. and Ottenstein, L.M., “The program dependence graph in a software development environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (Pittsburgh, PA, April 23-25, 1984), ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).
11. Reps, T. and Yang, W., “The semantics of program slicing and program integration,” pp. 360-374 in *Proceedings of the Colloquium on Current Issues in Programming Languages,* (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science* Vol. *352*, Springer-Verlag, New York, NY (March 1989).
12. Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).

From: Proc. of the 1st International Workshop on Automated and Algorithmic Debugging (1993): LNCS Vol 749