Efficient Comparison of Program Slices *

Susan Horwitz and Thomas Reps
Computer Sciences Department, University of Wisconsin−Madison, 1210 W. Dayton St., Madison, WI 53706

Summary. The *slice* of a program with respect to a component *c* is a projection of the program that includes all components that might affect (either directly or transitively) the values of the variables used at *c*. Slices can be extracted particularly easily from a program representation called a program dependence graph, originally introduced as an intermediate program representation for performing optimizing, vectorizing, and parallelizing transformations. This paper presents a linear-time algorithm for determining whether two slices of a program dependence graph are isomorphic.

1. Introduction

The *slice* of a program with respect to a component *c* is a projection of the program that includes all components that might affect (either directly or transitively) the values of the variables used at *c*. For example, Figure 1 shows a program that computes the arithmetic and geometric means of the numbers in the range *M*..*N*, and the slices of this program with respect to three different components. Program slicing was originally defined in [17];[1] [12] gave an efficient algorithm for computing program slices using a representation of programs called the *program dependence graph*.

This paper studies the problem of determining whether two slices are "equal." It is not necessary for the two slices to be exactly identical in the sense of their representations occupying the same storage locations; two slices are considered to be equal if their (labeled) graph representations are *isomorphic*—that is, if there is a 1-to-1 and onto map between the vertex sets of the two graphs that preserves adjacency and labels.

---

* This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602 and CCR-8958530, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, and by grants from IBM, DEC, Xerox, 3M, Eastman Kodak, and the Cray Research Foundation.

[1]Weiser's definition of a slice was more general than the one given here. Weiser allowed a slice to be specified in terms of a component *c* and a set of variables *S*; variables in *S* might or might not be used at *c*.

Original Program	Slice with respect to "amean := sum / num"	Slice with respect to "gmean := prod ** (1 / num)"	Slice with respect to "num := N − M + 1"
program	**program**	**program**	**program**
sum := 0	sum := 0	prod := 1	num := N − M + 1
prod := 1	i := M	i := M	**end**
i := M	**while** i≤N **do**	**while** i≤N **do**	
while i≤N **do**	sum := sum + i	prod := prod * i	
sum := sum + i	i := i + 1	i := i + 1	
prod := prod * i	**od**	**od**	
i := i + 1	num := N − M + 1	num := N − M + 1	
od	amean := sum / num	gmean := prod ** (1 / num)	
num := N − M + 1	**end**	**end**	
amean := sum / num			
gmean := prod ** (1 / num)			
end			

Fig. 1. A program and three of its slices. Note that while the programming language does not include input statements, variables can be used before being assigned to; these variables' values are taken from the initial state.

The work presented here is the first to address the problem of efficiently testing whether two slices are isomorphic, and we present a linear-time algorithm for the problem. Although there is no known polynomial-time graph-isomorphism algorithm for arbitrary, unlabeled graphs [5], there are efficient algorithms for restricted classes of graphs, such as graphs of bounded valence [11]. Slice-isomorphism testing also concerns a restricted class of graphs: the vertices and edges in a program dependence graph are labeled, and the labeling, in conjunction with one of the classes of program-dependence-graph edges (called *def-order edges*—discussed in Section 2), permits the control and flow edges incident on each vertex to be totally ordered. It is this property that allows our slice-isomorphism-testing algorithm to run in time linear in the sum of the sizes of the two slices being tested.

The significance of a slice is that it captures a portion of a program's behavior in the sense that, for any initial state on which the program halts, the original program and the program that corresponds to the slice compute the same sequence of values for each component of the slice [13]. In our case a program component can be (1) an assignment statement, (2) a control predicate, or (3) an expression in an output statement. Because a program component can be reached repeatedly in a program, by "computing the same sequence of values for each element of the slice" we mean the following: (1) for an assignment statement, the same sequence of values is assigned to the target variable; (2) for a control predicate, the same sequence of boolean values is produced; and (3) for an expression in an output statement, the same sequence of values is written out.

Theorem. (*Slicing Theorem* [13]). *Let s be a slice of program P and let Q be a program whose program dependence graph is isomorphic to s. If σ is a state on which P halts, then (1) Q halts on σ, and (2) corresponding components in s and Q compute the same sequence of values.*

An immediate consequence of the Slicing Theorem is that two programs that have a slice in common exhibit the same behavior at all corresponding components of the slice.

Corollary. (*Slicing Corollary*). *Let* s_1 *and* s_2 *be slices of programs* P_1 *and* P_2*, respectively, such that* s_1 *is isomorphic to* s_2*. Then, for any initial state σ on which both* P_1 *and* P_2 *halt, corresponding components in* s_1 *and* s_2 *compute the same sequence of values.*

Consequently, the slice-isomorphism-testing technique described in this paper provides a safe way of testing whether two program components (possibly in different programs) have equivalent execution behaviors: if the two components have isomorphic slices then they are guaranteed to have equivalent execution behaviors.[2]

Being able to test whether two slices are isomorphic has a number of practical uses in program-development environments. For example, the slice-isomorphism test can be employed in the language-based program-differencing algorithm described in [7]. Slice-isomorphism testing is also an important aspect of the problem of integrating different variants of a program's source code [8]. However, the integration algorithm given in [8] sidesteps some potential problems in testing slice isomorphism by relying on a pre-existing mapping between the components of different program variants. This mapping is maintained by a special program editor that must be used to create the program variants from the base program (it places a hidden tag on each program component, which is used to identify which components correspond in different variants). Because of the need for such tags, the integration algorithm from [8] can only support program integration within a closed system; integration cannot be performed on programs developed outside the system using ordinary text editors, such as *vi* and *emacs*. As we discuss in Section 5, the techniques developed in [16] offer a way to integrate programs in the absence of such editor-supplied tags—and hence make it possible to handle programs created using ordinary text editors—by manipulating sets of slices. The need to test whether two slices are isomorphic is fundamental to this approach.[3]

The remainder of the paper is organized as follows. Section 2 defines the program dependence graphs of [8]; these are the dependence graphs used by our algorithms (the dependence graphs of [12] could also be used at the expense of first computing the transitive closure of the graphs' output dependences). Section 2 also describes how to compute slices using dependence graphs. Section 3 presents our linear-time algo-

---

[2]A much different algorithm for testing whether two program components have equivalent execution behaviors is given in [18].

[3]A drawback of the approach is that it entails additional costs for finding the program that corresponds to the set of dependence graphs that result from an integration.

rithm for determining whether two slices are isomorphic. The algorithm is proved correct in Section 4. Section 5 discusses applications and extensions.

Sections 2, 3, and 4 concern the slice-isomorphism problem for programs written in a restricted programming language that includes only assignment statements, conditional statements, while loops, and output statements. (Although the language does not include input statements, our convention is that variables can be used before being defined, in which case their values are taken from the initial state). One of the extensions described in Section 5 adapts our slice-isomorphism-testing algorithm to handle programs that consist of multiple (and possibly mutually recursive) procedures.

The isomorphism-testing algorithms from this paper have been been implemented as part of the Wisconsin Program-Integration System [14, 15]. In particular, the isomorphism-testing algorithm (including an extension to handle variable renaming as described in Section 5) is used as the basis of an implementation of the language-based program-differencing algorithm from [7].

2. The Program Dependence Graph and Program Slicing

Different definitions of program dependence representations have been given, depending on the intended application; however, they are all variations on a theme introduced in [10]. The definition used in this paper is taken from [8].

The *program dependence graph* (or PDG) for a program P, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edges. G_P is defined in terms of an augmented version of the program's control-flow graph. The standard control-flow graph includes a special *Entry* vertex and one vertex for each *if* or *while* predicate, and each assignment and output statement in the program. The control-flow graph is augmented by adding a vertex at the beginning of the control-flow graph for each variable x that may be used before being defined. This vertex, called an *initial definition* vertex, represents an assignment to x from the initial state, and is labeled "$x := InitialState(x)$."

Example. Figure 2(a) shows the augmented control-flow graph for the program of Figure 1.

The vertices of G_P are the same as the vertices in the augmented control-flow graph (an *Entry* vertex and one vertex for each predicate, assignment statement, output statement, and initial definition). The edges of G_P represent *control* and *data* dependences.

A control dependence edge from vertex v to vertex w is denoted by $v \longrightarrow_c w$. The source of a control dependence edge is always either the *Entry* vertex or a predicate vertex; control dependence edges are labeled either **true** or **false**. The intuitive meaning of a control dependence edge from vertex v to vertex w is the following: if the program component represented by vertex v is evaluated during program execution and its value matches the label on the edge, then, assuming that the program terminates normally, the component represented by w will eventually execute; however, if the value does not match the label on the edge, then the component represented by w may never execute. (By definition, the *Entry* vertex always evaluates to **true**.)

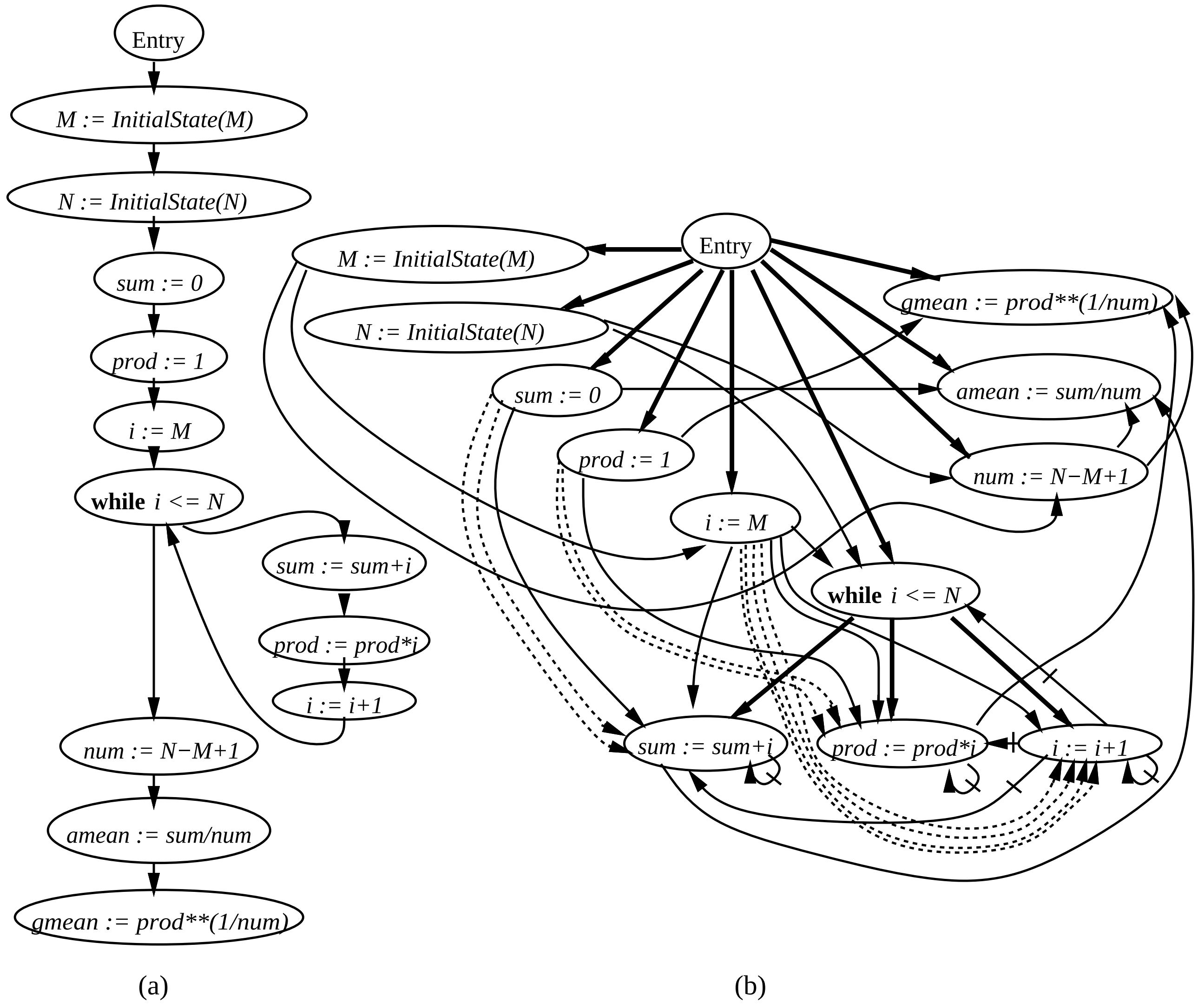


Fig. 2. (a) The augmented control-flow graph for the program of Figure 1; (b) the program's program dependence graph. In the program dependence graph, all edge labels have been omitted; control dependence edges are shown using bold arrows; loop-independent flow dependence edges are shown using arcs; loop-carried flow dependence edges are shown using arcs with a hash mark; def-order dependence edges are shown using dashed arcs.

Algorithms for computing control dependences in languages with unrestricted control flow are given in [3, 4]. For the restricted language under consideration here, control dependence edges reflect the nesting structure of the program (*i.e.*, there is an edge labeled **true** from the vertex that represents a *while* predicate to all vertices that represent statements nested immediately within the loop; there is an edge labeled **true** from the vertex that represents an *if* predicate to all vertices that represent statements nested immediately within the true branch of the *if*, and an edge labeled **false** to all vertices that represent statements nested immediately within the false branch; there is an edge labeled **true** from the *Entry* vertex to all vertices that represent statements that are not nested inside any *while* loop or *if* statement).

Data dependence edges include both *flow* dependence edges and *def-order* dependence edges. A flow dependence edge from vertex v to vertex w is denoted by $v \longrightarrow_f w$. Flow dependence edges represent possible flow of values, *i.e.*, there is a flow dependence edge from vertex v to vertex w if vertex v represents a program component that assigns a value to some variable x, vertex w represents a component that uses the value of variable x, and there is an x-definition clear path from v to w in the augmented control-flow graph. For the purposes of this paper it is convenient to assume that if vertex w uses variable x as more than one operand (*e.g.*, vertex w represents the statement "a := x + x ** 2") then there is more than one flow-dependence edge from v to w, and that each such edge is labeled with the appropriate operand number.

Flow dependences are further classified as *loop independent* or *loop carried* [1]. A flow dependence $v \longrightarrow_f w$ is carried by the loop with predicate vertex p, denoted by $v \longrightarrow_{lc(p)} w$, if both v and w are enclosed in the loop with predicate vertex p, and there is an x-definition clear path from v to w in the augmented control-flow graph that includes a backedge to predicate vertex p. Loop-carried flow dependence edges are labeled with their carrying-loop-predicate vertex. A flow dependence $v \longrightarrow_f w$ is loop independent, denoted by $v \longrightarrow_{li} w$, if there is an x-definition clear path from v to w in the augmented control-flow graph that includes *no* backedge. It is possible to have several loop-carried flow edges between two vertices (each labeled with a different loop-predicate vertex); it is also possible to have both a loop-independent flow edge and one or more loop-carried flow edges between two vertices.

For the purposes of this paper it is convenient to assume that, for every vertex w, for every operand i of vertex w, the set of flow edges $v \longrightarrow_f w$ for operand i is stored at w, sorted as follows. The loop-independent edge $v \longrightarrow_{li} w$ (if it exists) is stored first, followed by the loop-carried edges from v to w. The loop-carried edges are ordered by the nesting depth of their carrying loops, from most-deeply to least-deeply nested. Note that if the most-deeply nested carrying loop is at nesting level j and the least-deeply nested carrying loop is at level k, then the loops at all intermediate levels (between j and k) are also carrying loops for this dependence. It is this *density* property of loop-carried flow edges that makes it possible to find a canonical order for the incoming edges of a vertex in time proportional to the number of such edges.

Def-order dependence edges are included in program dependence graphs to ensure that inequivalent programs cannot have isomorphic program dependence graphs [6]. A program dependence graph contains a def-order dependence edge from vertex v to vertex w iff all of the following hold:[4]

---

[4][6] includes a fourth condition, that v and w are in the same branch of any conditional that encloses both of them. However, the same set of programs have isomorphic dependence graphs whether or not this condition is included, and the algorithms of this paper are easier to express given the definition of def-order dependences that omits this condition.

(1) Vertices v and w are both assignment statements that define the same variable.

(2) There exists a vertex u such that there is a flow dependence edge from v to u, and there is a flow dependence edge from w to u.

(3) The program component represented by v occurs before the program component represented by w in a preorder traversal of the program's abstract syntax tree.

A def-order dependence edge from v to w with "witness" u is denoted by $v \longrightarrow_{do(u)} w$. A def-order edge is labeled with its "witness" vertex.

Note that a program dependence graph can be a multigraph (can have more than one edge $v \rightarrow w$). In this case, the edges are distinguished by their types (control, loop-independent flow, loop-carried flow, or def-order) and/or by their labels (the operand number for flow edges, the carrying loop-predicate vertex for loop-carried flow edges, and the "witness" vertex for def-order edges).

Example. Figure 2(b) shows the program dependence graph of the program from Figure 1. All edge labels have been omitted. Note that several pairs of vertices have multiple def-order edges between them. Each such edge would be labeled with a different witness vertex.

2.1. *Computing Program Slices Using the Program Dependence Graph*

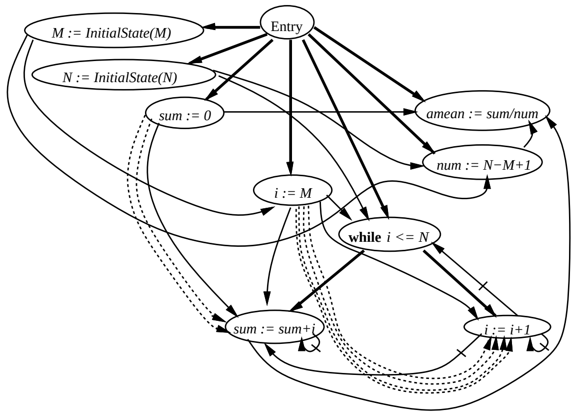
For a vertex x of a program dependence graph G, the *slice* of G with respect to x, denoted by G/x, is a graph containing all vertices on which x has a transitive flow or control dependence (*i.e.*, all vertices that can reach x via flow or control edges): $V(G/x) = \{ w \in V(G) \mid w \longrightarrow^{*}_{c,f} x \}$.

The edges in the graph G/x are essentially those in the subgraph of G induced by the vertices of the slice, with the exception that a def-order edge $v \longrightarrow_{do(u)} w$ is only included if, in addition to v and w, the vertex u that is directly flow dependent on the definitions at v and w is included in the slice. In terms of the three types of edges in a program dependence graph we have:

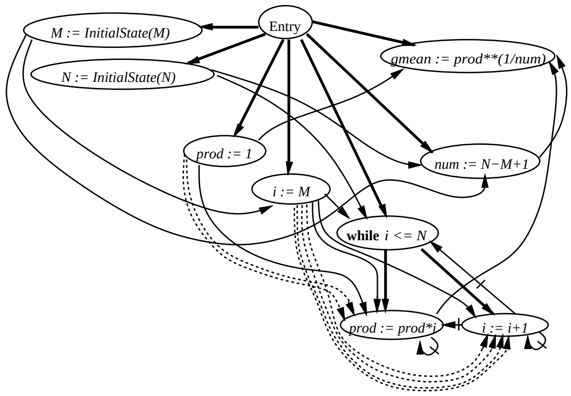
$$\begin{aligned} E(G/x) = \; & \{ (v \longrightarrow_f w) \in E(G) \mid v, w \in V(G/x) \} \\ & \cup \{ (v \longrightarrow_c w) \in E(G) \mid v, w \in V(G/x) \} \\ & \cup \{ (v \longrightarrow_{do(u)} w) \in E(G) \mid u, v, w \in V(G/x) \}. \end{aligned}$$

Example. Figure 3 shows the graphs that correspond to the slices shown in Figure 1.

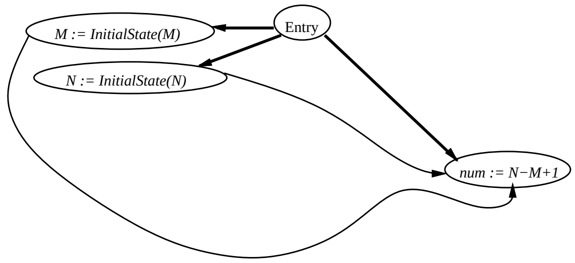
In practice, the slice of graph G with respect to vertex x can be computed by following flow and control edges backwards, starting at x. To permit the efficient computation of slices, each vertex should, therefore, include pointers to its control and flow *predecessors* rather than to its control and flow *successors*. Because a def-order edge is included in a slice only if its "witness" vertex is included in the slice, the presence of a def-order edge should be recorded only at its "witness" vertex (as the pair <*source*, *target*>), rather than at its source or target vertices.



Slice with respect to "*amean := sum/num*"



Slice with respect to "*gmean := prod**(1/num)*"



Slice with respect to "*num := N - M* + 1"

Fig. 3. The program dependence graphs that correspond to the slices given in Figure 1. All edge labels have been omitted; control dependence edges are shown using bold arrows; loop-independent flow dependence edges are shown using arcs; loop-carried flow dependence edges are shown using arcs with a hash mark; def-order dependence edges are shown using dashed arcs.

3. A Linear-Time Algorithm for Testing Slice Isomorphism

In this section we define what it means for two slices to be isomorphic, we give an algorithm for testing slice isomorphism, and we show that the algorithm runs in time linear in the sum of the sizes of the two slices.

Definition 3.1. Let s_1 and s_2 be slices with respect to v_1 and v_2, respectively (*i.e.*, $(s_1/v_1) = s_1$ and $(s_2/v_2) = s_2$). Then s_1 and s_2 are *isomorphic with respect to vertices* v_1 *and* v_2 iff all of the following hold:

(1) Slices s_1 and s_2 have the same number of vertices and the same number of edges.

(2) There is a 1-to-1 and onto map M from the vertices of s_1 to the vertices of s_2, such that

  (i) $M(v_1) = v_2$.

  (ii) For all vertices w of s_1, w and $M(w)$ are the same kind of vertex (*i.e.*, entry, assignment-statement, output-statement, if-predicate, while-predicate, or initial-definition).

  (iii) For all vertices w of s_1, w and $M(w)$ have identical abstract syntax trees (*i.e.*, corresponding internal nodes of the two vertices' abstract syntax trees contain the same operator, and corresponding leaf nodes contain the same identifier or the same constant).

(3) For every edge $e = v \to w$ in s_1 there is an edge $e' = M(v) \to M(w)$ in s_2 such that

  (i) The edge type of e (control, loop-independent flow, loop-carried flow, or def-order) is the same as the edge type of e'.

  (ii) If e is a control dependence edge then its true/false label matches the true/false label of e'.

  (iii) If e is a flow dependence edge then its operand-number label matches the operand-number label of e'.

  (iv) If e is a loop-carried flow dependence edge with carrying-loop-predicate label p then the carrying-loop-predicate label of e' is $M(p)$.

  (v) If e is a def-order edge with witness-vertex label u then the witness-vertex label of e' is $M(u)$.

Note that although a program dependence graph can be a multigraph, no two edges between a given pair of vertices can have both the same type and the same label. Thus, a 1-to-1 and onto vertex map M from slice s_1 to slice s_2 induces an edge map E_M from the edges of s_1 to the edges of s_2. Furthermore, if s_1 and s_2 are isomorphic under vertex map M, then E_M is the edge map specified by part (3) of Definition 3.1.

3.1. *An Algorithm for Testing Slice Isomorphism*

Given slices s_1 and s_2 with respect to vertices v_1 and v_2, respectively, our algorithm for testing slice isomorphism, TestIsomorphism(s_1, s_2, v_1, v_2), shown in Figure 4, performs three steps.

Step 1: Preprocess slices s_1 and s_2

Preprocess slices s_1 and s_2, creating graphs G_1 and G_2, as follows:

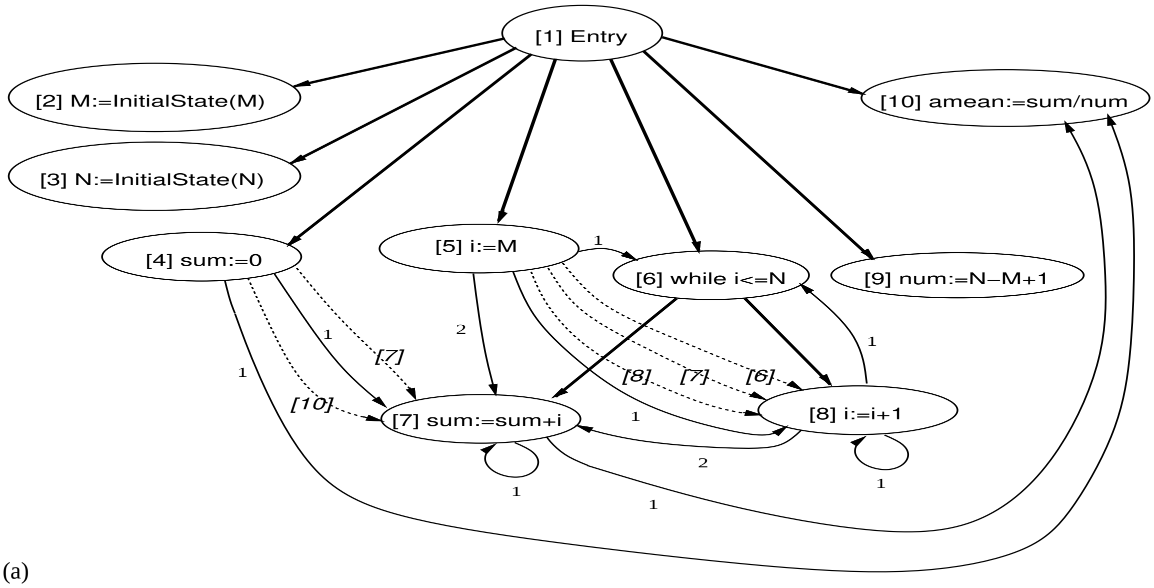
(a) For every flow dependence edge $e = v \to u$, label e with the number of def-order edges with witness label u for which v is the target. This number is used in Step 2 below to order u's incoming edges; thus, this number is called e's *ordering* number.

(b) Remove all def-order edges from the two slices. (Figure 5 shows a slice before and after preprocessing.

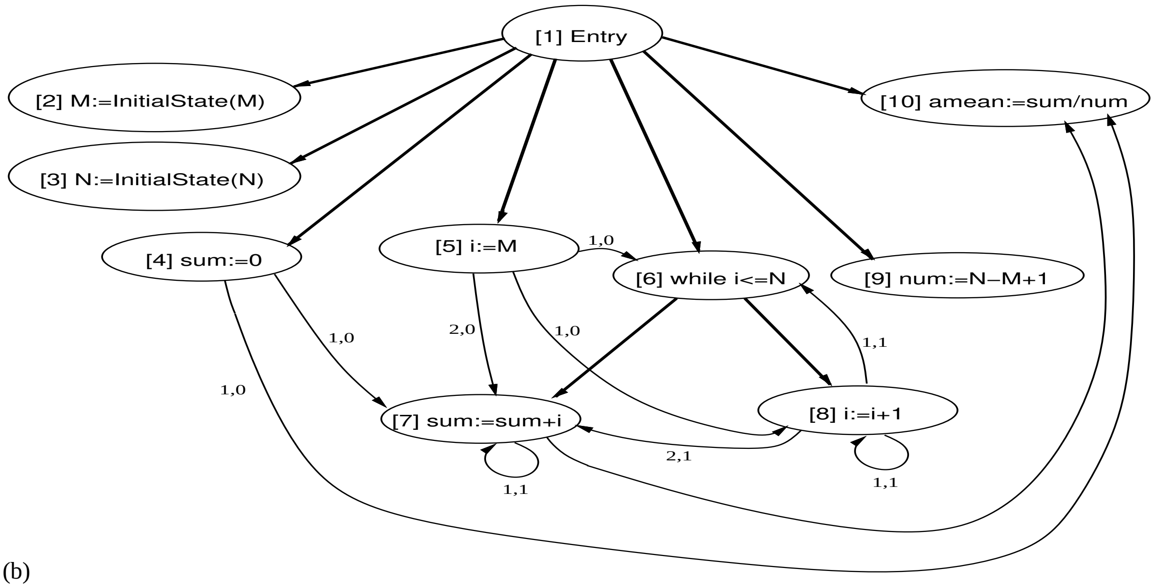
In Figure 5(b), flow edges are labeled with (operand-number, ordering-number) pairs. For example, in Figure 5(b) the edge [8] → [7] has operand-number label 2—because variable i is defined at vertex [8] and is

function TestIsomorphism(s_1, s_2: slices, v_1, v_2: the sources of s_1 and s_2, respectively): **returns** a boolean
declare
  G_1, G_2: preprocessed slices
  DFS: a map from vertices of G_1 to vertices of G_2
begin
  G_1 := Preprocess(s_1)
  G_2 := Preprocess(s_2)
  DepthFirstSearch(G_1, v_1)
  DepthFirstSearch(G_2, v_2)
  $DFS := \lambda v\,.$ (the $w \in G_2$ such that $depthFirstNumber(w) = depthFirstNumber(v)$)
  return(FinalCheck(G_1, G_2, v_1, v_2, *DFS*))
end

Fig. 4. Linear-time slice-isomorphism test.



(a)



(b)

Fig. 5. The first slice from Figure 3 before and after preprocessing. All control edges are shown (unlabeled). In Figure 5(a), all def-order edges are shown with their witness labels. Only flow edges that contribute to the presence of a def-order edge are shown: in Figure 5(a) these edges are shown with their operand-number labels; in Figure 5(b) these edges are shown with their operand-number labels and their ordering-number labels.

used as the second operand at vertex [7]—and has ordering-number label 1—because in Figure 5(a) vertex [8] is the target of one def-order edge with witness label [7], namely the edge $[5] \longrightarrow_{do([7])} [8]$.)

Step 2: Number the graphs' vertices using depth-first search

Assign numbers to the vertices of G_1 and G_2 using depth-first search: start with v_1 or v_2 (depending on which graph is being processed), and search backwards along flow and control dependence edges; assign a number to a vertex when that vertex is first visited. The depth-first search is made deterministic by using edge type, carrying-loop-predicate vertex, operand number, and the ordering numbers assigned in the preprocessing step to decide the order in which to visit a vertex v's predecessors. In particular, v's incoming edges are ordered as follows:

(1) Vertex v's incoming control dependence edge is first in the ordering.

(2) For each operand i, in order,

For each ordering number j, in order,

First, append the loop-independent flow dependence edge with operand-label i and ordering number j (if it exists).

Then append all loop-carried flow dependence edges with operand-label i and ordering number j, ordered by loop nesting level (from most-deeply to least-deeply nested).

The depth-first search is also used to count the number of vertices and edges in each graph. (Code for this step is given in Figure 6.)

Step 3: Check whether the depth-first numbering is an isomorphism map

Let *DFS* be a map from the vertices of G_1 to the vertices of G_2, defined as follows. For every vertex v of G_1, let $DFS(v)$ be the vertex of G_2 that has the same depth-first search number as vertex v. Let E_{DFS} be the edge map induced by *DFS*. Check whether the maps *DFS* and E_{DFS} are isomorphism maps between G_1 and G_2. In addition, check whether, for every flow-dependence edge e of graph G_1, the ordering number of e is the same as the ordering number of edge $E_{DFS}(e)$. (Code for this step is given in Figure 7.) If both of these conditions are satisfied, then the algorithm returns **true**; under all other conditions, the algorithm returns **false**.

3.2. *Algorithm TestIsomorphism Runs in Linear Time*

By considering each step of the algorithm in turn, we can show that the algorithm runs in time proportional to the sum of the sizes of the two slices.

(1) The preprocessing step computes ordering numbers (which involves examining each flow dependence edge and each def-order edge once) and removes def-order edges. Thus, an upper bound for the time required for this step is O(number of edges).

(2) Depth-first search requires time O(number of vertices + number of edges). The ordering of a vertex's incoming edges can be performed by function OrderInEdges in time proportional to the number of edges because of the density property of loop-carried flow edges discussed in Section 2.

```
global vertexNumber, edgeNumber: integer

procedure DepthFirstSearch(G: a preprocessed slice, v: a vertex of G; the source of the slice)
begin
   for every vertex x in G do depthFirstNumber(x) := 0 od
   vertexNumber := 0; edgeNumber := 0
   Visit(G, v)
   numberOfVertices(G) := vertexNumber; numberOfEdges(G) := edgeNumber
end

procedure Visit(G: a preprocessed slice, v: a vertex of G)
declare
   inEdges: an ordered list of the edges incident on v
begin
   vertexNumber := vertexNumber + 1
   depthFirstNumber(v) := vertexNumber
   inEdges := OrderInEdges(G, v)
   for each edge e in inEdges (in order) do
      edgeNumber := edgeNumber + 1
      if depthFirstNumber(source(e)) = 0 then Visit(G, source(e)) fi
   od
end

function OrderInEdges(G: a preprocessed slice, v: a vertex of G): returns an ordered list of the edges incident on v
declare
   inEdges: an ordered list of edges
   e: an edge of G incident on v
begin
   if v is the Entry vertex then return(emptyList) fi /* the Entry vertex has no incoming edges */
   inEdges := incomingControlEdge(v)
   for i := 1 to numberOfOperands(v) do
      for j := 1 to numberOfFlowPredecessors(v, i) do
         if ∃ an incoming loop-independent flow edge e for operand i with ordering number j – 1
            then inEdges := inEdges || e
         fi
         for k := mostDeeplyNested(v, i, j – 1) to leastDeeplyNested(v, i, j – 1) do
            e := the incoming loop-carried flow edge for operand i with ordering number j – 1 and labeled with
                  the predicate of the enclosing loop at nesting level k
            inEdges := inEdges || e
         od
      od
   od
   return(inEdges)
end
```

Fig. 6. DepthFirstSearch is applied to the two preprocessed slices G_1 and G_2. It assigns a depth-first search number to each vertex and counts the number of vertices and edges in G_1 and in G_2.

```
function FinalCheck(G1, G2: preprocessed slices,
                    v1, v2: the vertices with respect to which the slices were taken,
                    DFS: a map from the vertices of G1 to the vertices of G2
                   ): returns a boolean
declare
   E_DFS: a map from the edges of G1 to the edges of G2
   w: a vertex of G1
   e: an edge of G1
begin
   if numberOfVertices(G1) ≠ numberOfVertices(G2) then return(false) fi
   if numberOfEdges(G1) ≠ numberOfEdges(G2) then return(false) fi
   if DFS(v1) ≠ v2 then return(false) fi
   for all vertices w in G1 do
      if vertexKind(w) ≠ vertexKind(DFS(w)) then return(false) fi
      if w and DFS(w) do not have identical abstract syntax trees then return(false) fi
   od
   E_DFS := the edge map induced by DFS
   for every edge e in G1 do
      if E_DFS(e) is undefined then return(false) fi
      if ordering-number label (e) ≠ ordering-number label (E_DFS(e)) then return(false) fi
   od
   return(true)
end
```

Fig. 7. FinalCheck tests whether the vertex map *DFS* and the edge map E_{DFS} define an isomorphism between graphs G_1 and G_2, and whether the ordering numbers of the edges of G_1 and G_2 correspond under map E_{DFS}.

(3) FinalCheck tests whether G_1 and G_2 are isomorphic and have corresponding ordering numbers under maps *DFS* and E_{DFS}. It examines each pair of vertices v and $DFS(v)$ exactly once, and also examines each pair of edges e and $E_{DFS}(e)$ once to check ordering numbers. This requires time O(number of vertices + number of edges) assuming that $DFS(v)$ and $E_{DFS}(e)$ can be determined in constant time, and that abstract syntax trees can be compared in constant time.

4. Correctness of the Isomorphism-Testing Algorithm

Theorem 4.1. *Function TestIsomorphism(s_1, s_2, v_1, v_2) returns* **true** *if and only if s_1 and s_2 are isomorphic with respect to v_1 and v_2.*

This theorem follows immediately from the following two lemmas:

Lemma 4.2. *Slices s_1 and s_2 are isomorphic under vertex map M iff the graphs G_1 = preprocess(s_1) and G_2 = preprocess(s_2) are isomorphic under M, and E_M respects ordering numbers.*

Lemma 4.3. *If G_1 = preprocess(s_1) and G_2 = preprocess(s_2) are isomorphic under some vertex map M, and E_M respects ordering numbers, then G_1 and G_2 are isomorphic under vertex map DFS and E_{DFS} respects ordering numbers.*

The proof of Lemma 4.2 relies on the following proposition, which we prove first before turning to the proofs of Lemmas 4.2 and 4.3.

Proposition 4.4. *For all vertices v, w, and u of slice s, such that $v \longrightarrow_{do(u)} w$, (i.e., v and w are both flow predecessors of vertex u for the same operand i) the ordering number of the edge $v \rightarrow u$ in preprocess(s) is less than the ordering number of the edge $w \rightarrow u$ in preprocess(s).*

Proof. Let S be the subgraph of s that includes u, its flow predecessors for operand i, and the flow edges and def-order edges between these vertices. It is clear that if the proposition holds for S and preprocess(S), then it holds for s and preprocess(s). We will show that the proposition holds for S and preprocess(S) using induction: we start with graph S', a subgraph of S and show that the proposition holds for S'; then we show that each time the size of S' is increased by adding a vertex from S (and its incoming and outgoing edges) the proposition still holds.

Base case. Subgraph S' is the graph induced on S by the three vertices u, v, and w. In S', the only def-order edge is from v to w; thus, in preprocess(S'), the ordering number of edge $v \rightarrow u$ is 0 and the ordering number of edge $w \rightarrow u$ is 1, so the proposition holds.

Induction Step. We assume that the proposition holds for subgraph S'_n with n vertices, such that $1 \leq n <$ number-of-vertices(S). We create graph S'_{n+1} by first adding to S'_n a vertex x from S (and not in S'_n), and then adding to S'_{n+1} all edges of S that have both source and target in S'_{n+1}. We show that the proposition still holds for S'_{n+1}.

Let the values *old_v* and *old_w* be defined as follows:

$$old\_v \triangleq \text{ordering-number}(v \rightarrow u) \text{ in preprocess}(S'_n)$$
$$old\_w \triangleq \text{ordering-number}(w \rightarrow u) \text{ in preprocess}(S'_n)$$

By the induction hypothesis, $old\_v < old\_w$. We must show that in preprocess(S'_{n+1}) ordering-number($v \rightarrow u$) < ordering-number($w \rightarrow u$).

By the definition of def-order edges, there must be an edge between v and vertex x, as well as between w and vertex x in S, and therefore in S'_{n+1}. If the def-order edge between v and x is directed from v to x, then in preprocess(S'_{n+1}), ordering-number($v \rightarrow u$) = old_v. Because $old\_v < old\_w \leq$ ordering-number($w \rightarrow u$), we have that ordering-number($v \rightarrow u$) < ordering-number($w \rightarrow u$), as required. On the other hand, if the def-order edge between v and x is directed from x to v, then in preprocess(S'_{n+1}), ordering-number($v \rightarrow u$) = $old\_v + 1$. However, in this case, there must also be an edge from x to w in S'_{n+1}, because otherwise there would be a def-order-edge cycle. (Such a cycle cannot occur because the direction of a def-order edge is determined by the relative order of its source and target in a pre-order traversal of the program's abstract-syntax tree.) Thus, in preprocess(S'_{n+1}) we have $old\_v + 1 < old\_w + 1 =$ ordering-number($w \rightarrow u$), and so ordering-number($v \rightarrow u$) < ordering-number($w \rightarrow u$), as required. □

Proof of Lemma 4.2, $\Rightarrow$ case: We must show that if slices s_1 and s_2 are isomorphic (under some map M) then the graphs G_1 = preprocess(s_1) and G_2 = preprocess(s_2) are isomorphic under M, and E_M respects ordering numbers. Suppose this is not true. If $s_1 \approx s_2$ under map M, then obviously $G_1 \approx G_2$ under map M as well. Therefore, if the lemma fails, it must be because E_M does not respect the ordering numbers of G_1 and G_2; *i.e.*, there must be an edge $e = v \rightarrow u$ in G_1, such that ordering-number(e) $\neq$ ordering-number($E_M(e)$). This means that the number of def-order edges with witness label u incident on v in s_1 differs from the number of def-order edges with witness label $M(u)$ incident on vertex $M(v)$ in slice s_2. However, in this case, M is not an isomorphism map between s_1 and s_2 as assumed (since there must be some def-order edge with witness label u and target v in s_1 that has no corresponding edge in s_2, or *vice versa*). Contradiction.

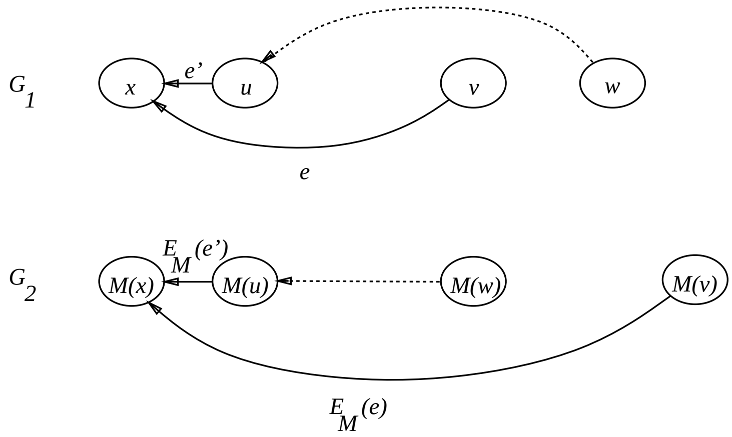
Proof of Lemma 4.2, $\Leftarrow$ case: We must show that if the graphs G_1 and G_2 are isomorphic under vertex map M, and E_M respects ordering numbers, then the slices s_1 and s_2 are isomorphic under M. Suppose $G_1 \approx G_2$ under map M and E_M respects ordering numbers, but not $s_1 \approx s_2$ under M. In this case, there must be vertices v and w in s_1 such that $v \longrightarrow_{do(u)} w$, but there is no edge $M(v) \longrightarrow_{do(M(u))} M(w)$ in s_2. The edge $v \longrightarrow_{do(u)} w$ means that both v and w are flow predecessors for the same operand of vertex u. Since M is an isomorphism map for G_1 and G_2, and since G_1 and G_2 include all of the flow edges in s_1 and s_2, it must be that $M(v)$ and $M(w)$ are also flow predecessors for the same operand of vertex $M(u)$; thus, there must be a def-order edge with witness $M(u)$, either from $M(v)$ to $M(w)$ or vice versa. Because we have assumed that there is no such edge from $M(v)$ to $M(w)$, it must be that the edge runs from $M(w)$ to $M(v)$. However, in this case, by Proposition 4.4, ordering-number($v \rightarrow u$) < ordering-number($w \rightarrow u$), but ordering-number($M(v) \rightarrow M(u)$) > ordering-number($M(w) \rightarrow M(u)$), so E_M does not respect the ordering numbers of G_1 and G_2 as assumed. Contradiction. □

Proof of Lemma 4.3. We prove Lemma 4.3 by proving a stronger property: if there exists a vertex map M such that G_1 and G_2 are isomorphic under M, and E_M respects ordering numbers, then the vertex map *DFS* is identical to map M. The proof is by contradiction.

Assume that the lemma is wrong. Consider the vertices of G_1 ordered by their depth-first-search numbers, and let vertex v be the first vertex in this sequence such that $M(v) \neq DFS(v)$. Note that $M(v)$ must have a higher depth-first-search number (in G_2) than $DFS(v)$, otherwise v would not be the first vertex on which M and *DFS* disagree. (Consider the vertex v' of G_1 whose depth-first-search number in G_1 equals the depth-first-search number of $M(v)$ in G_2; v' would occur earlier in the sequence than v and $M(v') \neq DFS(v')$.)

Note, in addition, that v cannot be the vertex v_1 with respect to which the slice was taken; by definition, $M(v_1) = v_2$, and both of these vertices have depth-first-search number = 1. Because v is not the first vertex in the depth-first-search number sequence, v must have at least one control or flow successor that precedes it in the depth-first-search number sequence. Let x be the control or flow successor of v that occurs latest in the sequence but ahead of v. $M(v)$ must also have a control or flow successor $M(x)$ with a lower depth-first-search number than $M(v)$, and it must be that $DFS(x) = M(x)$; otherwise x, not v, is the first vertex on which M and DFS disagree. Thus, x and $M(x)$ are visited on corresponding calls to procedure Visit of Figure 6.

Let w be the vertex of G_1 such that $M(w) = DFS(v)$. Note that w must have a higher depth-first-search number (in G_1) than v, otherwise v would not be the first vertex on which M and DFS disagree. The relationships between v, x, w, $M(v)$, $M(x)$, and $M(w)$ (and one additional vertex that will be introduced shortly) are depicted below (in the two graphs, left-to-right ordering of vertices indicates low-to-high depth-first-search numbering):



Note that in graph G_2 $M(v)$ is a direct predecessor of $M(x)$ and that $M(v)$ occurs later in the depth-first-search number sequence than $M(w)$. Therefore, by the properties of depth-first search, $M(w)$ must be a transitive predecessor of $M(x)$ (*i.e.*, there must be a path from $M(w)$ to $M(x)$ in G_2). By the assumption that G_1 and G_2 are isomorphic under map M, there must be a corresponding path from w to x in G_1.

Now consider the sequence of calls made on Visit during the depth-first search of G_2. Because $M(w)$ precedes $M(v)$ in depth-first search order in G_2 and $M(v)$ is a direct predecessor of $M(x)$, there must have been a call "Visit(G_2, $M(u)$)" (which traverses some edge $E_M(e')$ and eventually leads to the visit on $M(w)$) prior to the call "Visit(G_2, $M(v)$)," which traverses edge $E_M(e)$.

Because the calls on predecessors are made according to the canonical ordering of a vertex's incoming edges determined by function OrderInEdges, in the depth-first search of G_1 there must have been a call "Visit(G_1, u)," which traverses edge e', prior to the call "Visit(G_1, v)," which traverses edge e. However, because Visit performs a depth-first search, all transitive predecessors of u that have not been previously visited are visited during the call "Visit(G_1, u)." In particular, vertex w will (eventually) be visited for the first time as a result of this call. Note that vertex v cannot be a transitive predecessor of u because, by assumption, x is the control or flow predecessor of v that occurs *latest* in the depth-first-search number sequence of G_1 (but still ahead of v). Therefore, w must *precede* v in the depth-first-search number sequence of G_1, which contradicts our previous deduction that w must occur later than v in the sequence. □

5. Applications and Extensions

As discussed in the Introduction, the work on slice-isomorphism testing reported in this paper was motivated by our previous work on an algebraic framework for manipulating programs [16] and on identifying the textual and semantic differences between two versions of a program [7]. In this section we discuss several extensions to the slice-isomorphism testing algorithm presented in Section 3 that make it particularly useful in those two contexts.

5.1. Algebraic Program Manipulation

One of the results reported in [16] is the definition of a lattice-theoretic framework for representing and manipulating programs. In this lattice, a program is represented as a set of slices; programs can be combined using meet and join, as well as a kind of difference operation. These operations involve taking the intersection, union, and set-difference of the sets of slices that represent the programs. These set operations require testing two set elements—two slices—for equality (for example, if slice s is in set S_1, then it is in $S_1 \cap S_2$ iff there is a slice s' in S_2 such that $s = s'$). As discussed in the Introduction, two slices are considered to be *equal* if they are *isomorphic*. Thus, an efficient slice-isomorphism test is essential to providing efficient slice-set operations.

If a sequence of operations is to be performed on a collection of programs, it may be desirable to preprocess the programs so that slice isomorphism can be determined in *constant* time for any pair of slices. In other words, the preprocessing partitions the programs' slices into equivalence classes; two components are in the same class iff their slices are isomorphic. A naive partitioning technique based on slice-isomorphism testing would compare all pairs of slices; in the worst case this would require time $O(n^3)$, where n is the size of the programs' dependence graphs. A more efficient technique for performing such a partitioning would exploit the fact that a vertex's incident control and flow edges can be totally ordered, which permits an entire slice to be linearized in a canonical fashion. Given this insight, partitioning can be performed in time proportional to the sum of the sizes of all the slices in the programs, which is $O(n^2)$ in the worst case.

The key to this more efficient partitioning is to group isomorphic slices into equivalence classes, assigning each class a unique representative. Each vertex of the programs' graphs in turn is associated with the representative for its slice's isomorphism class; thus, two vertices have isomorphic slices iff they are associated with the same representative. The partitioning is performed as follows:

(1) A dictionary of linearized slices is maintained. Associated with each different slice is the unique representative for that equivalence class.

(2) For each program dependence graph G and each vertex v of G, the canonical linearization of slice G / v is computed. The linearized slice is looked up in the dictionary; if the slice is in the dictionary, the unique representative for that equivalence class is associated with vertex v; if the slice is not in the dictionary, it is inserted along with a new unique representative.

Assuming that a lookup can be performed in time proportional to the size of the slice (*e.g.*, using hashing) the total time for the partitioning is proportional to the sum of the sizes of the programs' slices.

5.2. Testing Component Equivalence in Procedures

A tool that identifies both the textual and semantic differences between two versions of a program is of obvious utility in a program-development environment. The design of such a tool is proposed in [7]. The tool makes use of an auxiliary algorithm that partitions the programs' components into equivalence classes. In [7], the suggested auxiliary algorithm is the one defined in [18]; however, the partitioning technique based on slice-isomorphism testing discussed above can also be used for this purpose. While Yang's algorithm is more efficient than the technique based on slice-isomorphism testing, the latter has the advantage of being extendible to handle programs with procedures, as described below (it might also be possible to extend Yang's technique to handle procedures, however no such extension is currently known).

One way to identify procedure components with equivalent execution behaviors is to compare the components' "backward-2" slices, where a backward-2 slice is obtained using the second pass of the interprocedural slicing algorithm defined in [9]. (Backward-2 slices are computed using a *system dependence graph*, a graph representation for programs with procedures. A backward-2 slice taken with respect to a component c in procedure P includes all components of P or of procedures called (directly or transitively) from P that might affect the values of the variables used at c. The backward-2 slice does *not* include components of procedures that call P that might affect the values of the variables used at c.)

Because system dependence graphs include some kinds of vertices and edges that are not in program dependence graphs, the slice-isomorphism testing algorithm of Section 3 does not immediately apply to backward-2 slices. However, it is straightforward to extend that algorithm to handle backward-2 slices because for every vertex in a backward-2 slice the vertex's incoming edges can be canonically ordered. Consequently, it is possible to determine whether two backward-2 slices are isomorphic in linear time, and it is possible to partition the components of a procedure into equivalence classes in time proportional to the sum of the sizes of the procedure's backward-2 slices.

As shown in [2] two procedure components with isomorphic backward-2 slices have equivalent behaviors according to the following definition: two procedure components have equivalent execution behaviors iff they produce the same sequence of values when their respective procedures are called with the same argument values. (Note that two such procedure components may produce different sequences of values when their respective *programs* are executed, because the programs may include different sequences of calls and/or may make those calls with different argument values.)

5.3. Isomorphism Under Variable Renaming

One advantage of Yang's technique for identifying program components with equivalent execution behaviors as compared to testing whether the components have isomorphic slices is that the former technique can find equivalences in the presence of variable renaming. For example, consider the two pairs of programs shown in Figure 8. In each pair, the two programs' final statements clearly have equivalent execution behaviors (the same value is assigned to both left-hand-side variables); however, the statements do not have isomorphic slices because of the differences in variable names.

Minor changes to the definition of slice isomorphism and to the algorithm for slice-isomorphism testing can be made so that in each of the pairs shown in Figure 8, the two programs *are* considered to be isomorphic. Rather than requiring that corresponding vertices have identical abstract syntax trees, corresponding vertices are required to have abstract syntax trees that are identical up to variable renaming. That is, for each vertex v in slice s_1, there must exist a one-to-one and onto map from the variable names used in v to the variable names used in the corresponding vertex of slice s_2. Note that each vertex can have a different map; for example, in Figure 8(b), the map for the third statement maps x to a, while the map for the final statement maps x to b.

Acknowledgements. We would like to thank Genevieve Rosay, Sam Bates, and Thomas Bricker for their roles in implementing the isomorphism-testing algorithms.

$x := 0$	$a := 0$	$x := 0$	$a := 0$
$y := 1$	$b := 1$	$y := 1$	$b := 1$
$z := x + y$	$c := a + b$	$z := x - y$	$c := a - b$
		$x := z + 1$	$b := c + 1$
(a)		(b)	

Fig. 8. In 8(a) and (b), the slices with respect to the two programs' final statements are not isomorphic; however, in both cases the two statements have equivalent execution behaviors.

REFERENCES

1. Allen, J.R., "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. dissertation, Dept. of Math. Sciences, Rice Univ., Houston, TX (April 1983).

2. Binkley, D., "Multi-procedure program integration," Ph.D. dissertation (in preparation), Computer Sciences Department, University of Wisconsin, Madison, WI (1991).

3. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K., "An efficient method of computing static single assignment form," pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).

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. Hoffmann, C.M., *Group-Theoretic Algorithms and Graph Isomorphism, Lecture Notes in Computer Science,* Vol. 136, Springer-Verlag, New York, NY (1982).

6. 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), ACM, New York, NY (1988).

7. Horwitz, S., "Identifying the semantic and textual differences between two versions of a program," *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 20-22, 1990), *ACM SIGPLAN Notices* **25**(6) pp. 234-245 (June 1989).

8. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

9. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).

10. Kuck, D.J., Muraoka, Y., and Chen, S.C., "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up," *IEEE Trans. on Computers* **C-21**(12) pp. 1293-1310 (December 1972).

11. Luks, E., "Isomorphism of bounded valence can be tested in polynomial time," pp. 42-49 in *Proceedings of the Twenty-First IEEE Symposium on Foundations of Computer Science* (Syracuse, NY, October 1980), IEEE Computer Society, Washington, DC (1980).

12. 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, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

13. Reps, T. and Yang, W., "The semantics of program slicing," TR-777, Computer Sciences Department, University of Wisconsin, Madison, WI (June 1988).

14. Reps, T., "Demonstration of a prototype tool for program integration," TR-819, Computer Sciences Department, University of Wisconsin, Madison, WI (January 1989).

15. Reps, T., “The Wisconsin program-integration system reference manual,” Unpublished document, Computer Sciences Department, University of Wisconsin, Madison, WI (April 1990).

16. Reps, T., “Algebraic properties of program integration,” To appear in *Science of Computer Programming*, ().

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

18. Yang, W., Horwitz, S., and Reps, T., “Detecting program components with equivalent behaviors,” TR-840, Computer Sciences Department, University of Wisconsin, Madison, WI (April 1989).