2

Program Slicing

In this chapter, we introduce program slicing. The original definition of program slicing is proposed by Weiser [1,2] mainly used for program debugging and maintenance. Weiser defined a slice of program *S* is a reduced program obtain from program *P* by removing statements based on data flow and control flow analysis.

*S* can be executed independently and keeps the same behavior when it’s involved in *P*. Program slicing is the process to obtain the slice *S*.

So far, program slicing has been extended in variety method to adjust different use and properties in different applications. Tip [3] conducted a survey about all these types of program slicing technologies. Static slicing is distinct from dynamic slicing without input assumption. The survey summarizes static and dynamic slicing, starting from the basic algorithm, procedures, unstructured flow and composite data and gives an overview of slicing technologies applied in different application areas.

This thesis focuses on static program slicing. First in this chapter, we introduce the basic notions and character of static program slicing and Weiser’s data flow algorithm. Then we explain slicing on a program dependence graph (PDG). Lastly, we discuss the development of program slicing technology.

2.1 Static program slicing

2.1.1 Introduction

A program slice, conform to Weiser’s theory [1], has the following two properties:

1. A slice S of program P, is obtained from a specific slicing criteria denoted as a pair of value *<i, V>*, where *i* is the line number of statement in *P*, and *V* is set of variables defined or used at *i*.
2. A slice S can be obtained by deleting zero or more statements from program *P*. Meanwhile, *P* and *S* must behave the same with respect to *<i,V>*.

Figure 2.1(a) and 2.1(b) is an example to illustrate Weiser’s program slicing. This example is supporting Weiser’s theory from two aspects: slice *S* on slicing criteria <9,{sum}> is achieved by deleting several statements; looking at the 9th statement, executing the slice S and the program P both lead to the same result on variable *sum*.

1

2

3

4

5

6

7

8

9

10

var a=1;

var b=2;

var add = function(x,y){

return x+y;

}

var foo = function(z){

return z+1;

}

var sum=add(a,b);

var increase =foo(a);

var a=1;

var b=2;

var add = function(x,y){

return x+y;

}

~~var foo = function(z){~~

~~return z+1;~~

~~}~~

var sum=add(a,b);

~~var increase =foo(a);~~

Figure 2.1(a) a JavaScript program P; 2.1 (b) get slice S out of slicing criteria <9,{sum}>

2.1.2 Data flow analysis

Beside, Weiser also put forward some constructive opinions in [1]. Weiser’s first slicing theory establishes on graph representation of program. Each node in a graph represents for a statement in program. Weiser defined a flowgraph *G*=<*N*,*E*,*no*> where *N* is the nodes, *E* is edges in set of *N×N* indicates the existing path from one node to another, an initial node *n0* as single entry where the program start. A hammock graph structure *G=<N, E, n0, ne>* where *ne* is an exist nodewhere the program terminate is extensible definition of flowgraph. Another useful definition is *REF(n)* and *DEF(n)*. *REF(n*) indicates the set of variables whose value are used at statement n, *DEF(n)* indicates the set of variables whose value are changed at statement n. In [1], program slicing is done by flow datatype analysis [5].

Base on the original notions, Weiser proposed a data flow algorithm for program slicing in [2]. After discussinh about how to slice a program by using reader’s intuitive understanding on a flowgraph, Weiser introduced a method to find slices by tracing backwards according to dataflow analysis.

1. Introduction of flowgraph

Figure 2.2 is an example of slicing on flowgraph Weiser gave in [2]. From the figure we see that a flowgraph is an oriented graph with an initial node. The edge (*n*, *m*) in flowgraph indicates the execution progress can be from *n* to *m*, *n* is an immediate predecessor of *m*, and *m* is an immediate successor of *n*. A path of length *k* from *n* to *m* is all possible query on flow graph from node *n* to *m*. A node *n* is dominated of node *m* when *n* is on every path from *n0* to *m*. If *m* is on every path from *n* to the terminate node *ne* on flowgraph, *m* is a inverse dominator of *n*. Deleting statements in a flowgraph to calculate slices must ensure that there is no increase of immediate successor of statement during deletion. With this concerning, Weiser defined statement deletion as: a set of nodes with single successor can be seen a deleted group, for all predecessors of a deleted group, set deleted group’s unique successor as their new successor. The left part of figure 2.2 shows the result of removing statement in deleted group.

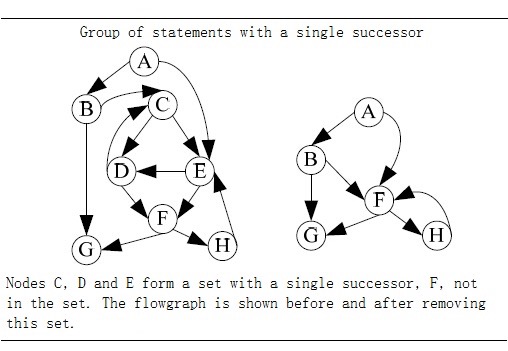


Figure 2.2 **TODO** : give a title + reference to paper where you copied it from.

1. Dataflow algorithm

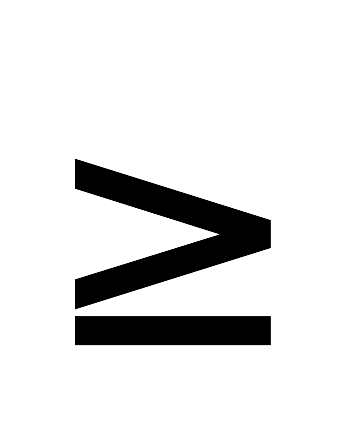
Weiser’s dataflow algorithm finds program slices by iteratively calculating the set of the related variables of each node in the flowgraph. The calculation steps are as follows:

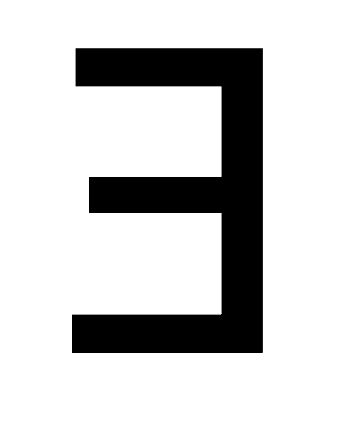
Input: the flowgraph of program *P*, and slice criteria *C=<i, V>*

Output: slice *S* of program *P* on slice criteria *C*.

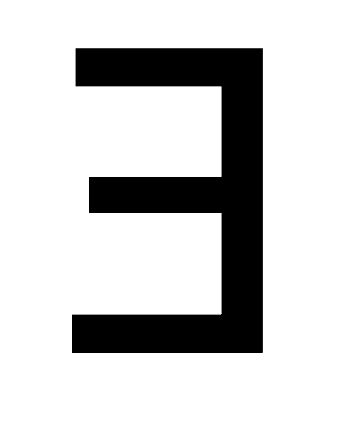
1. Calculation the directly relevant variables and directly relevant statement.
   1. For node *n* and *m* in flowgraph, if there exist path from *n* to *m*, then the set of directly relevant variables *R[0,C](n)* is denoted as: all variables v such that either:
      1. *n = i* and *v* is in *V*: *R[0,C](n)=V.*
      2. *n* is an immediate predecessor of a node *m*: *R[0,C](n) = {v | v∈ R[0,C](m), v∉DEF(n)}{v | v∈ REF(n), DEF(n)R[0,C](m) ≠ ϕ}*
   2. directly relevant statements is denoted as:

*S[0,C] = {N | DEF(n)R[0,C](n)≠ ϕ }*.

1. Iteratively calculate indirectly relevant variables and indirectly relevant statements.
   1. indirectly relevant variables set is denoted as *R[k,C](n)(k0)*, when calculate *R[k,C](n)*, we have to take account into control dependency. INFL(b) is a set used to represent for the statements influence on statement b, then:

*R[k+1,C](n) = R[k,C](n){n|n∈R[k,C](n),b∈INFL(b)}*

* 1. Similarly for indirectly relevant statements S[k,C]:

*S[k+1,C] =* *{n|n∈R[k,C](n),b∈INFL(b)}{n| DEF(N)R[i+1,C](n)≠ ϕ}.*

1. Repeat step 2 until the size of set *S* doesn’t increase any more, and the statements in *S* consist of slice.

2.1.3 Static slicing on PDG.

Many program slicing approaches use a so-called program dependence graph (PDG)[4,7] as an intermediate representation of a program. Similar with a flowgraph, nodes in a PDG correspond to statements and control predicates of the program, and the edges correspond to data and control dependencies between nodes. Data dependent means there exists a path *k* from node *m* to node *n*, and a variable is defined at m and gets used at n without redefining it at any other node on path *k*, then we can say *n* is data dependent on *m*. If there exist a path out of *m* that lead to execute *n*, then we can say *n* is control dependent with *m.*

***TODO*** *: introduce true/false control dependencies.*

According to the definition proposed by J.Ferrante et al. in [7], a PDG consists of a Control Flow Graph (CFG), a Control Dependency Graph (CDG) and a Data Dependency Graph (DDG). A CFG describes the control flow of a program; a CDG contains the control dependencies inside a program, the statement nodes in a CDG represent statements in the program, predicate nodes in a CDG represent iterlation loops or conditions. A DDG is a set of data dependencies between statements in a program. Their way to build a PDG is to first build a CDG and a DDG on top of a CFG, then integrate the CDG and the DDG into the PDG.

A slice is obtained by performing a backward traverse from an interested node in the graph, visiting all predecessors. In their later work[9], they state that slicing on a PDG is more accurate than the described, earlier method.

The below picture shows corresponding PDG of Figure 2.1(a).

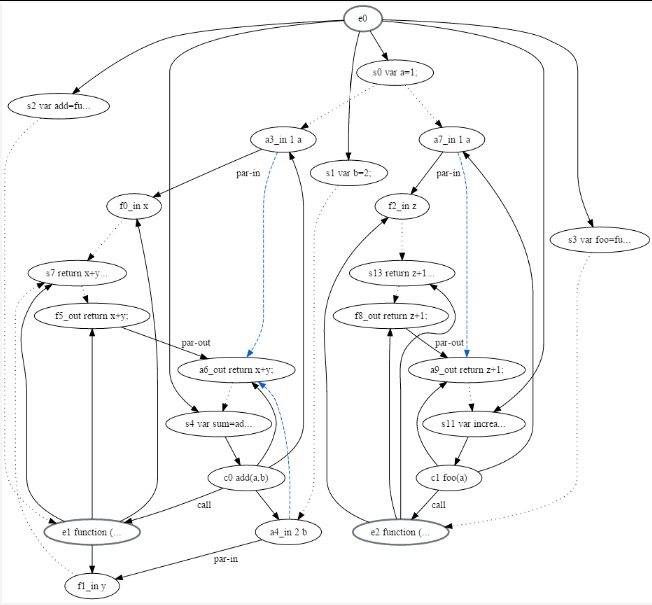


Figure 2.3 program dependency graph of program in figure2.1(a)

2.1.4 Other approach of static program slicing

The above approaches describe the simple case to compute slices of structured, single-procedure programs with scalar variables. In addition to this, program slicing also applies for inter-procedural program, unstructured control flow, composite variables and pointer, and concurrent program. The solutions to these problem will be discussed below.

***Procedures***

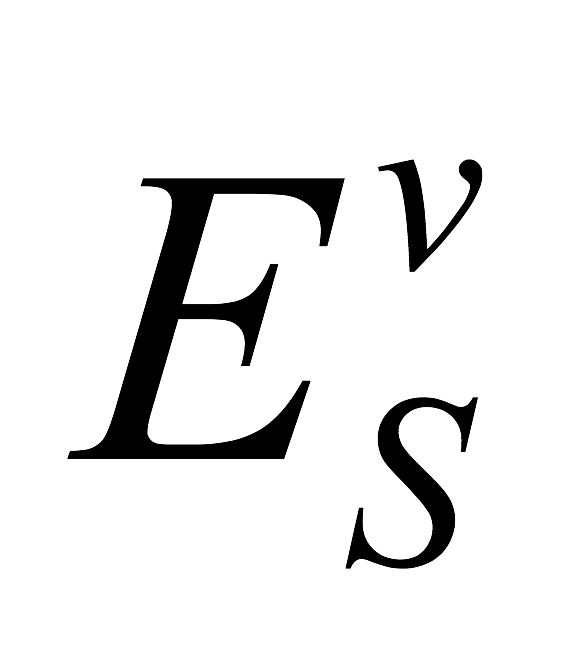
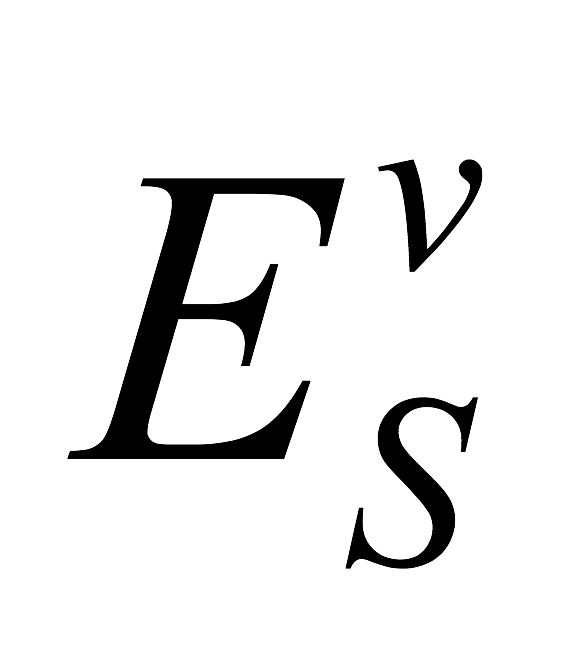
The main idea to solve inter-procedural static slicing is to construct a call-return structure to depict the interaction between procedures. Weiser’s approach [2,16] is able to approximate inter-procedural static slicing by taking account into *summary information* byusing previously techniques [8] and iteratively generates new slicing criteria. **TODO**: what are these “previously techniques”? You mean previously introduced techniques? (you described them in previous sections)? If not, describe them in short.

However this approach is inaccurate, because it fails to account for “calling context” problem as Horwitz, Reps, and Binkley pointed out in [18]. The shortcoming of Weiser’s approach is that it is infeasible in the case of entering a procedure *Q* fromprocedure *P* and exit *Q* to another, different procedure. To solve this, when a procedure *Q* is called by procedure *P*, all call sites that call procedure *Q* have to be considered, not only *P*. Horwitz et, al. had their PDG based on [7] with three other categories of vertices: a distinguished vertex called *entry vertex*, *initial definition of x* vertex represents initial state of variable *x* in program *P*, *final use of x* vertex represents final value of *x* compute by *P*. **TODO**: give a small example + PDG of that program. Point out the newly introduced edges and nodes.

Then they construct the *System Dependency Graph (SDG)* on top of their PDG. In the first step, they model procedure calls and parameter passing with five new kinds of vertices and three new kind of edges. A call site in a SDG is represented by a *call-site* vertex; on the calling side, *actual-in* and *actual-out* vertices are control dependent on the *call-site* vertex and copy the value of the actual parameter to temporary variables/ F*ormal-in* and *formal-out* vertices correspond to formal parameters of a procedure and are control dependent on this procedure’s entry vertex. A *call* edge start from *call-site* vertex to corresponding procedure entry vertex; a *parameter-in* edge is binding corresponding *actual-in* and *formal-in* vertex; similar, a *parameter-out* edge is binding corresponding *actual-out* and *formal-out* vertex. So the SDG includes two parts, a PDG for the main program, and a procedure dependency graph for each procedure. The second step to construct the SDG is to build transitive dependence edges. This dependent relationship due to the value changing of variables after the calling of a procedure. *Subordinate characteristic graphs* are used to compute transitive dependencies. This graph depicts the transitive flows between the procedure’s input and output, and also, the call and return relations between the call site and procedure. To perform an inter-procedural slicing, their algorithm includes two phases that can be derived by starting vertex *s*: first, identify the vertices which can reach *s* without descending into procedure calls, it’s either in *P* itself or in the procedure that calls on *P*; second, identify the vertices that can reach s from procedures called by *P* or from procedures called by procedures that call *P*. In addition, they also use a data flow analysis when building the procedure dependency graph to identify modified variables and referenced variables. This results in a more accurate slice.

Later, Lakhotia [22] presents an improved algorithm for inter-procedural static slicing on a SDG. Lakhotia’s approach labels vertices in the SDG with a three-valued vertex tag. These tag permits to traverse each edge at most once to get slices. **TODO**: what are the three values in this tag and why are they needed/better?

In a subsequent study, the SDG is extended into an object-oriented programs [20]. Donglin and Mary discuss a SDG [19] based on existing approach[20, 21] that can distinguish data members for different instances of the same class.

Another approach is also suitable for inter-procedural static slicing proposed by Bergerretti and Carre [16], they define a information-flow relations to identify the information transition statements that can be used to compute slices. In their approach set of variable denoted by *V* and *E* denote the set of predict expressions in program. For each statement *S*, they define three relations between V and E: *(e,v)λs* if for the value of *v* on entry to *S* potentially affects the value computed for *e*; *(e,v)μs* if the value computed for *e* potentially affects the value of *v* on exit from *S*; *(e,v)ρs* if the value of *v* on entry to *S* may affect the value of *v*’ on exit from *S*. They also introduced a notion partial statement denoted as  which indicates those value may be used in obtaining the value of *v* on exist from *S*. Slices can be got by replacing the those statements doesn’t contain some member of  by an empty statement.

***Unstructured control flow***

***Composite variables and pointer***

**Concurrency**

2.2 Development of program slicing

2.2.1 Dynamic and conditional program slicing

2.2.2 Backward and forward program slicing

2.3.3 Program slicing in different applications