# 2

# Program Slicing

In this chapter, we introduce program slicing. The original definition of program slicing is a 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 execute independently and keep same behavior when it's involved in *P*. Program slicing is the process to get slice *S*.

So far, program slicing has been extended in variety method to adjust different use and properties in different applications. [3] is a nice survey about program slicing technology. Static slicing is distinct from dynamic slicing without input assumption. [3] summarized static and dynamic slicing from basic algorithm, procedures, unstructured flow and composite data and gave overview of slicing technology applied in different application areas.

This thesis is stressing 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. In second section, slicing on PDG is explain in first since the previous version of STIP.js is built on PDG. Lastly, we'll discuss the development of program slicing technology.

## 2.1 Static program slicing

### 2.1.1 Introduction

A program slicing is conform to Weiser's theory[1] have 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 aspect: slice *S* on slicing criteria <9,{sum}> is got by deleting several statements; on 9$^{th}$ statement, execute S and P will lead to same result on variable *sum*.

| | | |
|---|---|---|
| 1 | var a=1; | var a=1; |
| 2 | var b=2; | var b=2; |
| 3 | var add = function(x,y){ | var add = function(x,y){ |
| 4 | return x+y; | return x+y; |
| 5 | } | } |
| 6 | var foo = function(z){ | ~~var foo = function(z){~~ |
| 7 | return z+1; | ~~return z+1;~~ |
| 8 | } | ~~}~~ |
| 9 | var sum=add(a,b); | var sum=add(a,b); |
| 10 | var increase =foo(a); | ~~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 construtive 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 $G=<N,E,n_o>$ where $N$ is the nodes, $E$ is edges in set of $N\times N$ indicates the existing path from one node to another, an initial node $n_0$ as single entry where the program start. A hammock graph structure $G=<N, E, n_0, n_e>$ where $n_e$ is an exist node where 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]. In [2], after discuss 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. Introduce of flowgraph

Figure 2.2 is an example of slicing on flowgraph Weiser gave in [2]. From the figure we can know, 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 $n_0$ to $m$. If $m$ is on every path from $n$ to the terminate node $n_e$ on flowgraph, $m$ is a inverse dominator of $n$. Deleting statement on flowgraph to get slices must ensure no increasing of immediate successor of statement during deletion. With this concerning, Weiser defined statement deletion is: a set of node 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.

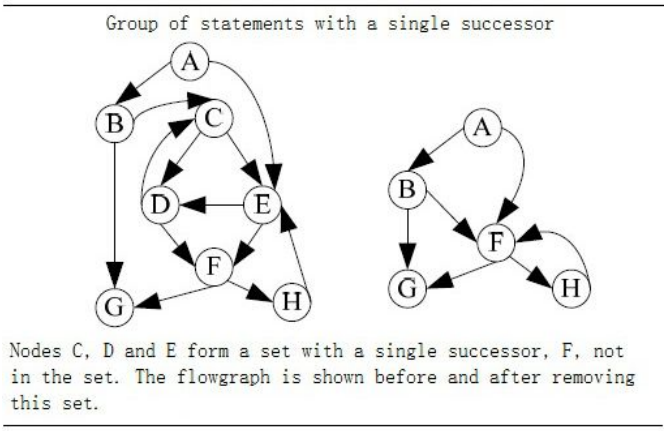


Figure 2.2

2. Dataflow algorithm

Weiser's dataflow algorithm is finding program slices by iteratively calculating the set of the related variables of each node in flowgraph. The calculation steps as follow:
Input: the flowgraph of program *P*, and slice criteria *C*=<*i, V*>
Output: slice *S* program *P* on slice criteria *C*.

1. Calculation the directly relevant variables and directly relevant statement.
   a) 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:
      i. *n* = *i* and *v* is in *V*: *R[0,C](n)=V.*
      *ii.* *n* is an immediate predecessor of a node *m*: $R[0,C](n) = \{v \mid v \in R[0,C](m),\ v \notin DEF(n)\} \cup \{v \mid v \in REF(n),\ DEF(n) \cap R[0,C](m) \neq \phi\}$
   b) directly relevant statements is denoted as:
   $S[0,C] = \{N \mid DEF(n) \cap R[0,C](n) \neq \phi\}$.
2. Iterative calculate indirectly relevant variables and indirectly relevant statements.
   a) indirectly relevant variables set is denoted as $R[k,C](n)(k \geq 0)$, 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:
   $R[k+1,C](n) = R[k,C](n) \cup \{n \mid \exists\, n \in R[k,C](n), b \in INFL(b)\}$
   b) Similarly for indirectly relevant statements S[k,C]:
   $S[k+1,C] = \{n \mid \exists\, n \in R[k,C](n), b \in INFL(b)\} \cup \{n \mid DEF(N) \cap R[i+1,C](n) \neq \phi\}$.
3. Repeat step 2 until the size of set *S* doesn't increase any more, and the statements in *S* consist of slice.

## 2.2 Slice on PDG.

Many program slicing approaches use PDG[4,7] as a intermediate representation of

program. As we introduced in previous chapter, our tier-splitting tool Stip.js is able to slice on PDG. Similar with flowgraph, nodes in PDG correspond to statements and control predicates of the program, and the edges correspond to data and control dependence between nodes. Data dependence means if there exist a path *k* from node *m* to node *n*, and a variable is defined at m and get used at n without redefine at any other node on path *k*, then we can say *n* has data dependence with *m*. If there exist a path out of *m* that lead to execute *n*, then we can say *n* has control dependence with *m.*

According to the definition proposed by J.Ferrante et al. In [7], PDG is consist of Control Flow Graph(CFG), Control Dependency Graph(CDG), Data Dependency Graph(DDG). CFG describes the control flow of a program; CDG contains the control dependencies inside program, the statement node in CDG represent is for statements in the program, predicate node in CDG is represent for loop or condition; DDG is a set of data dependencies between statements in a program. Their way to build a PDG is firstly build CDG and DDG on top of CFG, then integrate CDG and DDG into PDG. A slice is obtained by backward traverse from a interested node in the graph, visiting all predecessors. In their later work[9], they approve slicing on PDG is more accurate than earlier method.

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

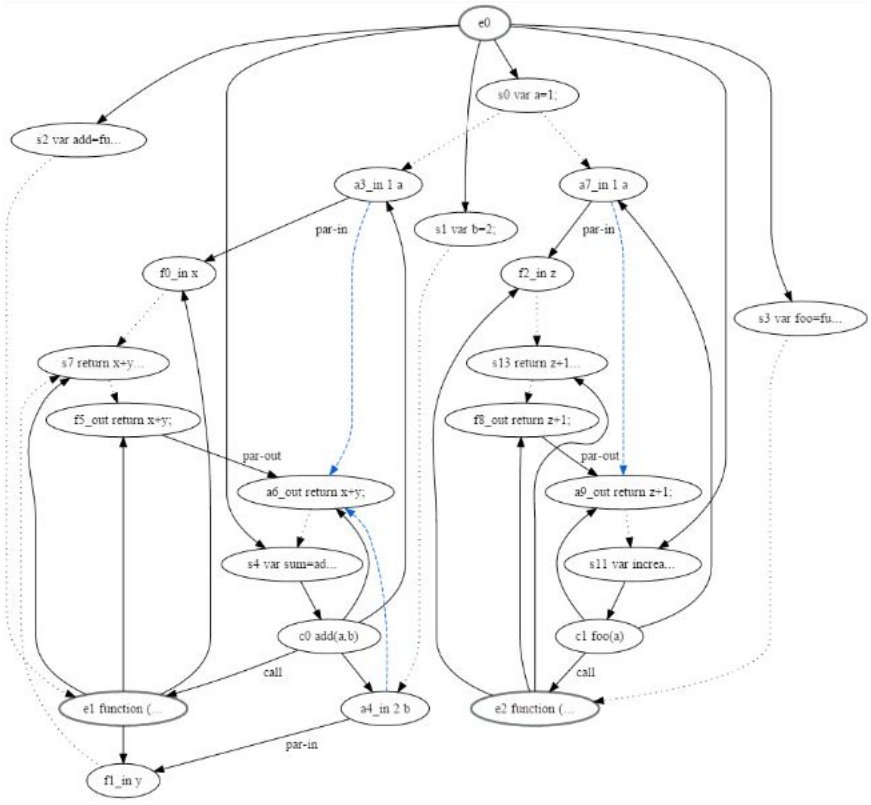


Figure 2.3 program dependency graph of program in figure2.1(a)

## 2.3 Development of program slicing

Slicing techniques are always based on some form of abstract, graph-based representation of the program under scrutiny, from which dependence relations between the entities it manipulates can be identified and extracted.

### 2.3.1 Dynamic and conditional program slicing

### 2.3.2 Backward and forward program slicing

### 2.3.3 Tools for program slicing

//structure of survey: method for static slicing

/*

1. Basic algorithm: dataflow, information flow , dependence graph
2. Procedures
3. Unstructured flow
4. Composite data */

//component identification through program slicing