Vrije Universiteit Brussel

FACULTY OF ENGINEERING

Program: Applied Computer Science

Tier splitting using Static Analysis

Thesis submitted in partial fulfilment of the requirements for the Master :
Master of Science in Applied Sciences and Engineering: Applied Computer Science, by

AI Xinran

Academic year: 2014-2015

Promoter: Prof. dr. Coen De Roover

Guidance by: Laure Philips

Jens Nicolay

Abstract

To develop web applications in a traditional three-tier architecture, developers have to know different technologies for each tier and manually combine these unalike technologies. This development pattern becoming even more complex since nowadays web applications are enabled with more interactive features. Tierless programming languages aim to reduce this complexity by enabling the developer to program a web application as a single mono-linguistic application. However, the current approaches to tierless programming requires the developer to learn a novel programming language. In a way, it doesn't reduce the effort of developing a rich internet application. In this regard, we are trying to extend a general-purpose language instead, like JavaScript already applied both to client tier and server tier, to realize tierless web applications. For this purpose, the tier splitting tool named Stip.js, based on program slicing and abstract interpretation theory, implements this idea.

The current version of Stip.js is able to distribute code into client and server tier via program slicing, a decomposition technique that uses a graph-based representation of the program, called the Program Dependency Graph. This graph is built using a state graph resulting from an abstract interpretation of the program. The process of tier splitting a tierless program consists of slicing source code into independent segments and reconstructing this code in client and server tier.

We investigate in a new slicing algorithm, called JipdaSlicer, to examine if the program dependency graph is necessary to acquire a correct slice of a program. After omitting the process of constructing a dependency graph, program slicing is implemented by static analysis of source code via state graph.

In this dissertation, we elaborate on our implementation of JipdaSlicer and set three test cases to validate our solution. By comparing the slicing result from JipdaSlicer with results from Stip.js, we can say JipdaSlicer is able to calculate accurate program slices and is more capable in analyzing the conditional statements than Stip.js.

Keywords: static analysis, program slicing, abstract interpreter

Acknowledgments

I would never have been able to finish my dissertation without the guidance of several people. First of all I would like to thank professor Coen De Roover for promoting this thesis and giving me the opportunity to conduct my apprenticeship at the Software Languages Lab. Moreover, special thanks go to my advisors Laure Philips and Jens Nicolay for all the help they have given me. To start with, they came up with the subject of this dissertation, furthermore they always assisted when needed and without their proofreading and wise words, this document would not have been up to standards as it is now.

I would like to thank my parents a lots for supporting me study abroad. Without their continuous support, encouraging, unconditional love, I would not be here today.

Also thanks my friends in there sharing joy and care. Thanks Brussels is a quite, beautiful city and comfortable to live

Content

Introduction 6

1.1 Problem statement 7

1.2 Contribution 8

1.3 Road map 9

Program Slicing 10

2.1 Static program slicing 11

2.1.1 Introduction 11
2.1.2 Data flow analysis 11
2.1.3 Static slicing on classic PDG 14

2.2 State of the art in program slicing 17

2.2.1 Method for static program slicing 17
2.2.2 Dynamic and conditional program slicing 23

2.3 Applications of program slicing 25

2.4 Conclusion 27

Abstract Interpreter 29

3.1 Background knowledge 30

3.1.1 Syntax and semantics of programs 31
3.1.2 Static semantics of programs 32

3.2 Static analysis based on abstract interpretation 33

3.3 CESK-Machine 37

3.4 Meta-circular evaluator 41

3.5 Conclusion 44

Program Slicing on State Graph 45

4.1 Introduction of STIP.js 45

4.1.1 Jipda framework 46
4.1.2 Constructing PDG on top of state graph 49

4.2 Case study on state transition graph 50

4.2.1 Assumption 50
4.2.2 Conceive outline of interpreter 54

4.3 Implementation design of JipdaSlicer 56

4.4 Conclusion **60**

Validation ***62***

5.1 Evaluation **62**

5.1.1 Methodology 62

5.2.2 Case testing 64

5.2 Conclusion **74**

Conclusion ***76***

6.1 Contribution **77**

6.2 Future work **78**

1 Introduction

Today people can't live without internet. The applications built for the internet provides a lot of functionalities to convenience for people's life. This kind of changes requires web applications to provide a large amount of information and services, and also to be able to handle more interactions with users. In this case, developers have to create so-called rich internet applications, where the client contains program logic as well. Developing such applications in the traditional three-tier architecture increases the complexity of web development, because each tier comes with its own technology stack and requires manually combination with other tier. In contrast with traditional methods, tier splitting of a web application is a rather new concept in the domain of web development. In distributed programming, tier splitting splits a single-tier application into its multi-tier variant [15]. This approach used on web applications tries to blur the distinction between the different tiers of a web application such that the development is more straightforward.

Instead of investing in a novel programming language to realize tier splitting, extending existing technologies such as JavaScript with distributed features to cater for tier splitting seems more beneficial. To enable tierless programming for the JavaScript language, we use static analysis based on abstract interpretation. On top of that, we need to split the actual tiers of the program which can be achieved by program slicing.

1.1 Problem statement

Development of web application in typical three-tier architecture has become complex since the applications require more interactive features. The typical three-tier architecture of web applications includes client, server, and data tier. Figure 1 depicts a web application structure that conforms to this three-tier architecture introduced in [6].

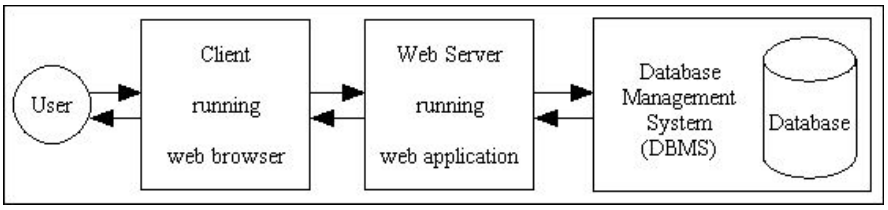


Figure 1 three-tier web application

Normally, the client tier represents the presentation layer such as a browser installed on user's computer, the server tier is used to handle computations and events and the data tier acts as an interface that communicates to the database system. The traditional development method requires the programmer to use the appropriate technologies for each tier and a manual combination of each technology is required. In this case to develop a web application, a combination of HTML, JavaScript and CSS is used for client tier, often together with Ajax and jQuery. The server tier itself is written in another programming language, often depending on which web framework is chosen by the programmer. A data query language like SQL usually is used to develop data tier. This leads to complex glue between each technology of the different tiers.

Tierless programming languages reduce this complexity by developing a web application in only one programming language. In these languages, a preprocessor or the runtime of the language can split the source code into client, server and sometimes a database tier. The communication between the different, distributed tiers is also handled by these languages. However, investment in novel programming languages is not an efficient way to relieve developers from handling the described complexity.

By allowing tierless programming in a general-purpose language the developer can profit from existing tool support for that language. JavaScript is an event-driven language that was originally designed to run on the client tier, but now is seeing increased adoption on the server tier thanks to Node.js. In view of expandability of JavaScript, we would like to enable it as tierless web applications. By using program slicing mechanisms, the tier-splitting tool called STIP.js[1] is able to split a tierless JavaScript program into client and server tier. Program slicing[1, 2] is a mechanism to get a reduced program which obtained from a source program by removing statements based on data flow and control flow analysis. Such a reduced program is a valid program itself and can be executed independently. This mechanism is widely used to analyze and understand the behavior of code[11,12,13].

Currently, the process of generating a tier split by the STIP.js tool consists of two parts. First, a State Transition Graph (STG), also referred to as state graph in this paper, is generated based on abstract interpretation for JavaScript. Concretely, the Jipda[2] framework is used for this analysis. This graph is a description of possible states the program can transition to. Then STIP.js constructs a Program Dependency Graph (PDG) out of the state graph. Such a graph contains data and control dependencies[7]. Data dependencies are used to represent the data flow relationships of a program, whereas control dependency corresponds to the control flow relationships. A PDG is used as the input for program slicing algorithms. From the selected node in the PDG, a slicing algorithm performs a backward traversal on the basis of data and control dependencies, resulting in two subsets of the tierless program: the client and server program. To get the tier split code, the developer only needs to annotate the tierless JavaScript code with @client and @server annotations.

1.2 Contribution

This dissertation introduces JipdaSlicer, a new program slicing algorithm based on the Jipda abstract interpreter framework. When slicing a program, this algorithm works on a state graph directly, instead of first constructing a Program Dependence Graph and using the classical slicing algorithms on this graph. JipdaSlicer accepts a variable pair containing a line number and

[1]https://soft.vub.ac.be/~lphilips/jspdg/stip/stip-web/stip.html

a variable identifier from source JavaScript code, which are together the slicing criterion. This criterion is a representation of a point in the code where a user wants to check which statements have effect on it. Based on this criterion a backward slice is computed; this is the set of instructions in the program that influence the slicing criterion. The current algorithm supports a subset of JavaScript, which we discuss in more detail in Chapter 4. To validate the new solution, we test and compare different cases. These cases are design in different situation to test if JipdaSlicer can compute a correct program slice. As we show, the result of JipdaSlicer is satisfactory.

1.3 Road map

The rest of this document is structured as follows: in Chapter 2, we describe program slicing. We introduce the basic concept of static program slicing in this chapter first and then give a general introduction of slicing on PDG. In the last of this chapter we introduce the development and different uses of program slicing technology. Chapter 3 introduces abstract interpretation. Chapter 4 is about program slicing on the state transition graph. We first introduce the structure of the tier splitting tool STIP.js. We dissect the structure and working principle of STIP.js. Then we give a case analysis to explain how can we apply program slicing techniques on the STG. In the same chapter, we describe our implementation design of JipdaSlicer. Chapter 5 validates the new algorithm, by comparing the result to that of a current program slicing tool for JavaScript. This dissertation ends with a conclusion chapter, including work summary and future work.

[2]https://github.com/jensnicolay/jipda/

2

Program Slicing

In this chapter, we introduce program slicing. The original definition of program slicing, proposed by Weiser [1,2] is mainly used for program debugging and maintenance. Weiser defined a slice of program *S* as a reduced program obtained from a 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 a variety of ways to adjust to different uses 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 criterion 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 slici-ng. This example supports Weiser's theory from two aspects: slice *S* on slicing criteria <*10,{abc}*> is achieved by deleting several statements; looking at the 10th statement, executing the slice S and the program *P* both lead to the same result on variable *ac*.

```
1    var a=1;
2    var b=2;
3    var c=3;
4    var ab=0;
5    if (a>b){
6        ab=a-b;
7        }
8    else ab=a+b;
9    var ac = a + c;
10   var abc = ac + c;
```

```
var a=1;
var b=2;
var c=3;
~~var ab=0;~~
~~if (a>b){~~
~~    ab=a-b;~~
~~    }~~
~~else ab=a+b;~~
var ac = a + c;
var abc = ac + c;
```

Figure 2.1(a) a JavaScript program P; 2.1 (b) get slice S out of slicing criteria <10,{abc}>

2.1.2 Data flow analysis

Weiser put forward some constructive opinions in [1]. Weiser's first slicing theory is established on the graph representation of a program. Each node in a graph represents a statement in program. Weiser defined a flowgraph $G = <N,E,n_o>$ where N is the set of nodes, E is the set of 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 starts. A hammock graph structure $G = <N, E, n_0, n_e>$ where n_e is an exit node where the program terminates. 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 discussing how to slice a program by using the 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

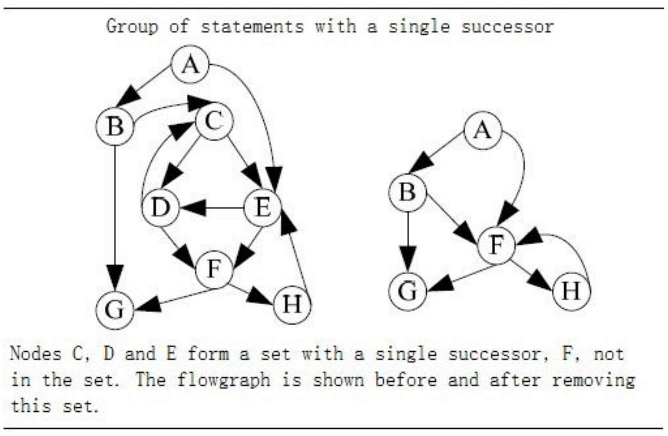

Group of statements with a single successor
A
B
C
D
E
F
G
H
A
B
F
G
H
Nodes C, D and E form a set with a single successor, F, not in the set. The flowgraph is shown before and after removing this set.

Figure 2.2 Deleting statement in a flowgraph [2].

Figure 2.2 is an example of slicing on a flowgraph Weiser gave in [2]. From the figure we see that a flowgraph is an oriented graph with an initial node. An edge (*n*, *m*) in the 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* with length *k*. A node *n* is dominate 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 of the flowgraph, *m* is an inverse dominator of *n*. Deleting statements in a flowgraph to calculate slices must ensure that there is no increase of immediate successors of statements during deletion.

With this concern in mind, Weiser defined statement deletion as: a set of nodes with single successor can be seen as 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.

2. 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. Calculate the directly relevant variables and directly relevant statements.
    a) For node *n* and *m* in flowgraph, if there exists a 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 are denoted as:

$$S[0,C] = \{N \mid DEF(n) \cap R[0,C](n) \neq \phi \}.$$

2. Iteratively 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 control dependencies into account. $INFL(b)$ 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 $S[k,C]$:

$$S[k+1,C] = \{n \mid \exists n \in R[k,C](n), b \in INFL(b)\} \cup \{n \mid DEF(N) \cap R[i+1,C](n) \neq \phi\}.$$

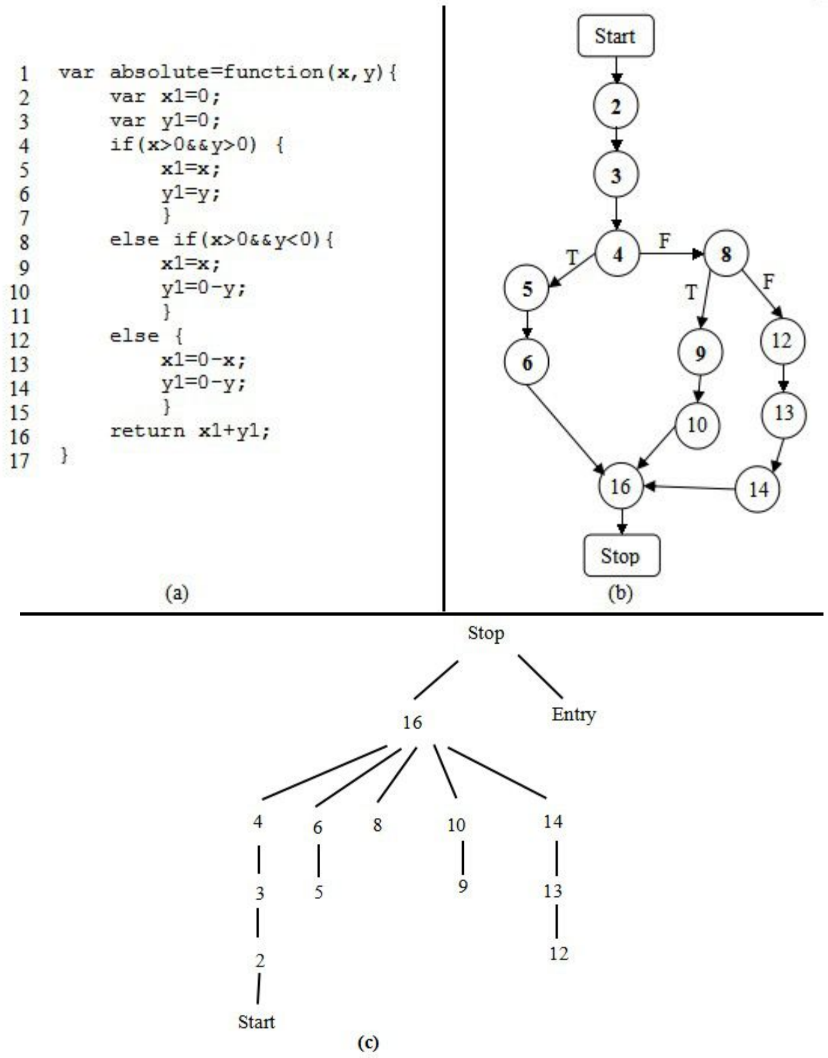
3. Repeat step 2 until the size of set *S* doesn't increase any more, and the statements in *S* consist of the slice.

2.1.3 Static slicing on classic 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. Informally, when a variable is defined or appears in one statement, another statement uses this variable without redefining it, then we say this dependency is a data dependency. If one statement determines whether another statement will be executed, then the latter is control dependent on former. J.Ferrante et al. defined a PDG which can be used for static slicing of single-procedure programs[7]. According to their definition, the control dependency in PDG is constructed on top of a Control Flow Graph (CFG) associated with a Control Dependency Graph (CDG), and a data dependency is identified through a Data Dependency Graph (DDG).

They follow the principle in [35,36] to define and construct CFG and post-dominator tree. A CFG describes the control flow of a program. A CFG is flowgraph augmented with a unique *Start* node represents for entry of the program and unique *Stop* node represents for exist of the program. Each node in CFG has at most two successors and they assume the attributes of outgoing edges get to the two successors is *"T"(true)* and *"F"(false)*. The *post-dominator* is defined as if every path from *n* to *Stop* contains *m*, then we can say *n* is post-dominated by *m*[35]. If a node *X* is control dependent on *Y* in CFG, is must exist a path from *X* to *Y*, and *X* is not post-dominated by *Y*. In other word, if a statement *S'* is dependent on statement *S*, there will be two edges out of *S*, the *"T"* edge will lead to execute on *S'*, *"F"* edge will lead the program jump to another statement. *"F"* edges also ensure there always has an exist for loop in CFG. Figure 2.3 shows an example code with its corresponding CFG and *post-dominator* tree.

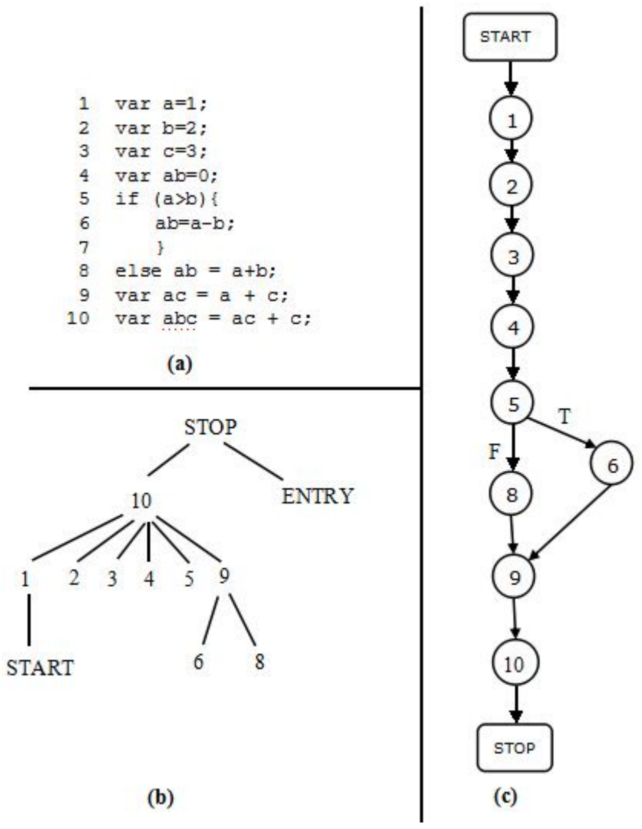
In the figure 2.3, we use the line number to mark each node in the CFG. Whether line 5 and 6 will be executed is control dependent on the predicate in line 4. Another information in CFG to get control dependences is in term of *region node*. This term can be seen as an entry of a block of statements with the same control conditions. In the example above, the prediction in line 4 can be seen a *region node* of statement in line 5 and 6. In addition, each block has an exit node. The control dependences can be identified by examining the edges (*A, B*) label with *"T"* or *"F"* in post-dominator tree. As the definition of control dependency, the examining is aim to skip the post-dominator of *A* which actually is the parent node of *A*. Integrate with two type of region node, we can get the CDG that contains the control dependencies inside a program.



```
1   var absolute=function(x,y){
2       var x1=0;
3       var y1=0;
4       if(x>0&&y>0) {
5           x1=x;
6           y1=y;
7           }
8       else if(x>0&&y<0){
9           x1=x;
10          y1=0-y;
11          }
12      else {
13          x1=0-x;
14          y1=0-y;
15          }
16      return x1+y1;
17  }
```

Figure 2.3 (a)An example code;(b) control flow graph of (a); (c) post-dominator tree.

Data dependent is defined as: 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*. DDG is a set of data dependencies between statements in a program. Their solution to build DDG is derived from [37,38] and perform data flow analysis[35] with additional assumption that every variable is initiated at entry. After constructing PDG by integrating the CDG and the DDG, a slice can be 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 earlier described method. Figure 2.4 shows the analysis of code in figure 2.1(a).



```
1   var a=1;
2   var b=2;
3   var c=3;
4   var ab=0;
5   if (a>b){
6       ab=a-b;
7       }
8   else ab = a+b;
9   var ac = a + c;
10  var abc = ac + c;
```

Figure 2.4 (a)An example code;(b) post-dominator tree; (c)control flow graph of (a).

Figure 2.5 shows the corresponding PDG of figure 2.1(a). The gray node is the slice on slicing criterion *<10,{abc}>*. In the figure, a solid line denotes a control dependency, a dashed line denotes a data dependency. Slicing starts by searching the node that represents line 10, and then traverse backward along dependency edges from it and mark all the visited nodes. In the first traversal, nodes represent for code “var b=2;” and “var ac=a+c;” are visited. In step 2, backward traverse the dependences edges from the node you marked in first traversal. Iterative step 2 until all marked node on the path from slicing node to entry node. The second turn of traversal is through nodes represent for code “var a=1;” and “var c=3;”.

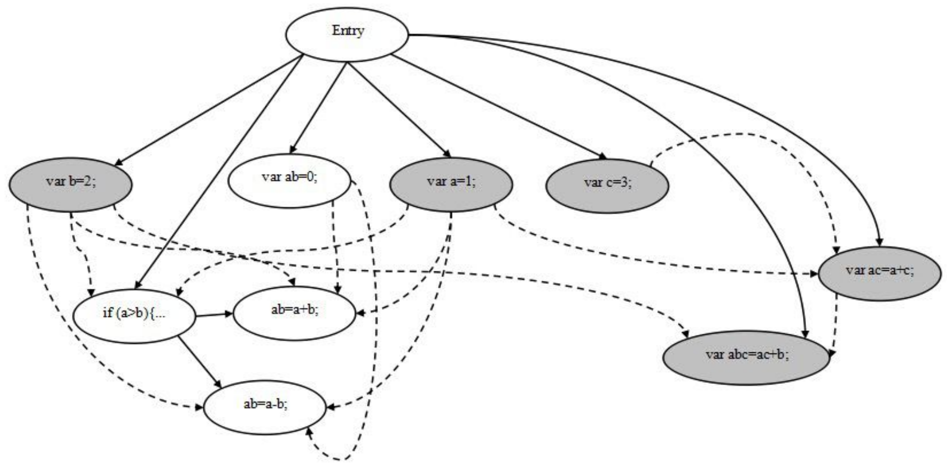


Figure 2.5 program dependency graph of program in figure2.1(a)

2.2 State of the art in program slicing

2.2.1 Method for static program slicing

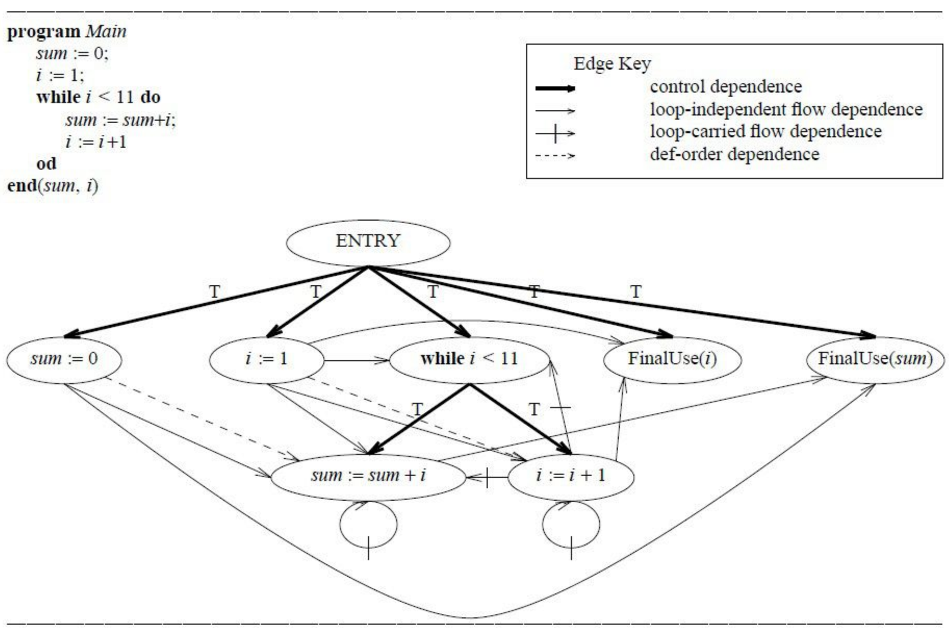
The two approaches above describe the simple case to compute slices of structured, single-procedure programs with scalar variables. The slice we get from a backward static slice, is the statements that influence the slicing criterion. In a similar way, a forward static slice[17] determines slice by computing the set of statements that are influenced by the the slicing criterion, it requires forward tracing on dependences from slicing criterion. A statement that is influenced by slicing criterion means that the value computed in the slicing criterion effects the statement or determines the execution of it. 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* [8]: MOD(P) is a set of variables that may be modified by procedure *P*, and the variables that may be used at *P* denoted as USE(P). The data flow algorithm is extended to identify a call effect with this *summary information*. Assuming a procedure *R* calls *P*, and *Q* is called by *P*, function Up(S) is defined as: a set of slice criteria (n_Q, V_Q) where n_Q is the last statement of *Q* and V_Q is all actual parameters from *P* pass to *Q*; function Down(S) is defined as: a set of (N_R, V_R) such that N_R is statements in *R* that call *P*, and V_R is the set of relevant variables at the first statement of *P* that substituted with formal parameters. The slice is achieved by iteratively calculating the Up(S) and Down(S) sets until no new criteria are generated.

However this approach is inaccurate, because it fails to account for the "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* from procedure *P* and exit *Q* to another, different procedure. To solve this, when a procedure *Q* is called by procedure *P*, not only *P* but also all call sites that call procedure *Q* have to be considered. 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* computed by *P*. They define control dependency between two vertex as: if v_1 is an entry node, v_2 is control dependent on v_1 when v_2 is not nested in loop or conditional; if v_1 presents for predicate of while-loop, then v_2 is a component nested in loop body and edge $v_1\text{->}_c v_2$ is labeled true; if v_2 represents for predicate of conditional statement, v_2 is the component in the conditional branch and edge $v_1\text{->}_c v_2$ is labeled true or false depends on whether v_1 lead to execute it. The data depencency edges occurs in their PDG contains two types: *flow dependences* and *def-order dependences*. *Flow dependences* is the path in standard CFG augmented with *loop carried* or *loop independent*. The edge *loop carried* flow dependence if the changes of value in the vertex within a loop will effect on the predicate or another vertex inside the loop. If the execution of vertex have no effect on the loop, then edge is a loop independent flow dependence. The *def-order* dependences edges represent for reassignment of a variable. Figure 2.6 is an example of the PDG defined in [18]. 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



```
program Main
    sum := 0;
    i := 1;
    while i < 11 do
        sum := sum+i;
        i := i+1
    od
end(sum, i)
```

Figure 2.6 An example program, which sums the integers from 1 to 10 and leaves the result in the variable sum, and its program dependence graph. The boldface arrows represent control dependence edges, solid arrows represent loop independent flow dependence edges, solid arrows with a hash mark represent loop-carried flow dependence edges, and dashed arrows represent def-order dependence edges[18].

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 a selected 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 alteration algorithm for inter-procedural static slicing on a SDG. Lakhotia's approach labels vertices in the SDG with a three-valued vertex tag: "$\perp$" marks node has not been visited; "T" marks the visited node; "β" indicate a visited node and some of its reachable node should be visited. These tag permits to traverse each edge at most once to get slices. The node in slice labeled with "T" at beginning and all other node is "$\perp$", and backward slicing algorithm traverse on the node and change the tag value based on operation they configured in algorithm where:

$$\perp \sqcap \beta = \beta \sqcap \perp = \beta,\ x \sqcap \mathsf{T} = \mathsf{T} \sqcap x, \text{ and } x \sqcap x = x.$$

All the nodes labeled β and T in the slice when the algorithm stops. However, it may require to change the value twice when slice program. In a subsequent study, the SDG is also extended into object-oriented programs [20] to identify reference between classes. 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 [17], they define a information-flow relations to compute slices by identifying the information transition statements. In their approach set of variable denoted by *V* and *E* denotes for the set of predict expressions in program. For each statement S, they define three relations between *V* and *E*: $(e,v) \in \lambda s$ if the value of *v* on entry to *S* potentially affects the value computed for *e*; $(e,v) \in \mu s$ if the value computed for *e* potentially affects the value of *v* on exit from *S*; $(e,v) \in \rho_s$ if the value of *v* on entry to *S* may affect the value of *v*' on exit from *S*. They also introduced a notion of *partial statement* which denoted as E_S^v where *v* is any program variable, *S* is any statement. The *partial statement* indicates certain values of expressions in *S* may be used in obtaining the value of *v* on exist from *S*. Slices can be got by replacing the those statements within *S* which doesn't contain some member of E_S^v by an empty statement.

Unstructured control flow

The approach of program slicing introduced above compute the projection for structured control flow. In [23], Ball and Horwitz point out the previous approach[1, 2, 7] is failure to deal with unstructured control flow, such as a unconditional jump statement **break**. Their solution to this problem is construct a PDG on top of an augmented CFG in which a unconditional jump (i.e, **break**, **goto**, **halt**) is represented as a *pseudo-predicate* vertex. The value of *pseudo-predicate* vertex is always set true, and its true-successor is the target of jump; its false-successor indicate the execution of statement is continuation. Similar with Ottenstein's approach, the slicing is graph reachability problem.

Choi and Ferrante [25] represented two methods to deal with **goto** statement. As same as [23], their first method is to build augmented program dependency graph base on augmented CFG. Choi and Ferrante extend augmented CFG with *fall-through* edge. Informally, a *fall-through* edge between two statement S_i and S_j indicates a relationship: if S_i is replaced by **null** statement, the program control flow coming into S_i would be forwarded to S_j. The notion *fall-through* edge is renamed by *lexical successor* in [28]. A *fall-through* edge is used to capture the relevant control dependent when the **goto** statement is deleted. Their testing example shows the augmented slice may contain statements that don't affect the statement for which the slice is computed. After analysis the disadvantage of this method, Choi and Ferrante proposed the second method that computes smaller slice as the executable slice by remove redundant cascaded **goto** statement. In this method, they define a new kind of slice named *CFG slice*. *CFG slice* is extract from CFG that includes nodes from which there is a path to the statement used to compute slice. The executable slice is out of classical PDG and *CFG slice* by replaced with each statement's immediate post-dominator.

Another PDG-based static slicing algorithm for unstructured flow is proposed by Agrawal [28]. Agrawal demonstrated the conventional slicing algorithm (refers to slicing algorithm for classic PDG) is easy to be extended with conditional jump statement in the form of **if** P **then goto** L: if the predicate *P* is in included in the slice because there is other statement control dependent on it, then the associated jump statement *L* must be included in the slice. In the case of unconditional jump statement, Agrawal found the way to determine a statement in a slice by: if and only if the the nearest post-dominator and nearest lexcial successor of statement *J* is different, then *J* must be included in the slice. A statement *S'* is post-dominator of another statement *S*, when every path from *S* to exist contains *S'*[35]. Considering a sequence of statements S_1, S_2, S_3 in original program, if nearest post-dominator of S_2 is different with nearest lexcial successor, it means exclude S_3, S_2 cause control jump to elsewhere. In this case S_2 is unconditional jump statement in original program, omit S_2 from the slice will cause the control always transfer from S_1 to S_3. Agrawal distinguished structured jump statement if its target statement is also its lexical successor. The property of structured jump ensure a single traversal of the post-dominator tree is sufficient to obtain correct slice. Second, if a predicate statement is in slice by conventional slicing algorithm, the jump statement directly dependent on it will be also included in the slice.

Composite variables and pointer

In order to construct flow dependences for pointers, *aliasing* which refers to the phenomenon of two or more variables point to the same memory location has to be determined. The James R. and David proposed [29] algorithm use symbolic execution to construct *aliasing* identical list. They define two type of address: address for static object and address for dynamic object. Address for dynamic object which enable to keep indirect assignment through dynamic address can be treated as assignment to array. Their algorithm consists of two phases. In the fisrt step, extract Pointer State Subgraph (PSS) by deleting the nodes in CFG without pointer value, and also deleting the branch node under the node being deleted, then repeat propagate address from address creation node to all successor on PSS. Second, the point state information we get in first step is used to determine which statements should be included in a slice[34].

Concurrency

Multiple flows, synchronization, communication, non-deterministic selection make it more difficult to identify primary program dependences in concurrent programs. Cheng introduced a graph-theoretical approach[31] for concurrent program slicing based on his previous work. Nondeterministic parallel control-flow net introduced in [31] is used to represent control flows, and can be bundled with information concerning definitions and the uses of variables and communication channels. In addition to data and control dependency, Cheng defines *selection dependency*, *synchronization dependency*, *communication dependency* to construct *Program Dependence Net* (PDN). *Selection dependency* is a kind of control dependency involves non-deterministic selection statement; *synchronization dependency* reflect the synchronous execution between two statement; *communication dependency* corresponds to modification of a variable at a point directly has influence on the value of a variable at another point via interprocess communication. Program slicing is a depth-first or breadth-first graph traversal algorithm on PDN. In final, Cheng also discuss how to alter his algorithm into dynamic slicing and forward slicing.

2.2.2 Dynamic and conditional program slicing

So far, the static program slicing techniques we introduced are independent of the input value of the program. On the contrast, dynamic and conditional program slicing have to take account into input at the slicing criterion. The difference between dynamic and conditional slicing is that the former only considers one specific input when computing slices, but the latter computes slices with a range of possible inputs.

Dynamic program slicing is first proposed by Korel and Laski [21,26]. Their approaches depend on data flow analysis. They redefine the flowgraph of a program P as a tuple $<N, A, en, ex>$ where N is a set of nodes represent for statement, A is a set of binary relationships between nodes in N, en and ex are used to label the unique entry and exit node of the program. The notion of $arc(n,m) \in A$ identifies the control transition from node n to m. They distinguish the path whose execution is invoked by an input feasible path. After the execution of a feasible path derived by a specific input, the tracing history through the path is referred to as program trajectory T. The slicing criterion is defined as a triple (x, I^q, V) where x is the input, V is subset of program variables, and I^q is the q^{th} element of the program trajectory. The dynamic flow they introduce includes three types: Data-Data relations(DD), Test-Control relations(TC) and Identity Relation(IR). DD models the relationship of definition and usage of same variable, and TC is used to determine which statements will be executed upon the predicate in **while** or **if-then-else**. IR refers to duplicate elements in *T*. Furthermore, they subdivide the use of a variable with its last definition as definition-use(DU). The slice is computed by iterative finding directly and indirectly relevant statement. The role of three type of relation ensure traverse on dynamic flow. They formalize the process as follow[21]:

$S^0 = A^0 = LD(q,V) + LT(I^q)$,

$S^{i+1} = S^i + A^{i+1}$,

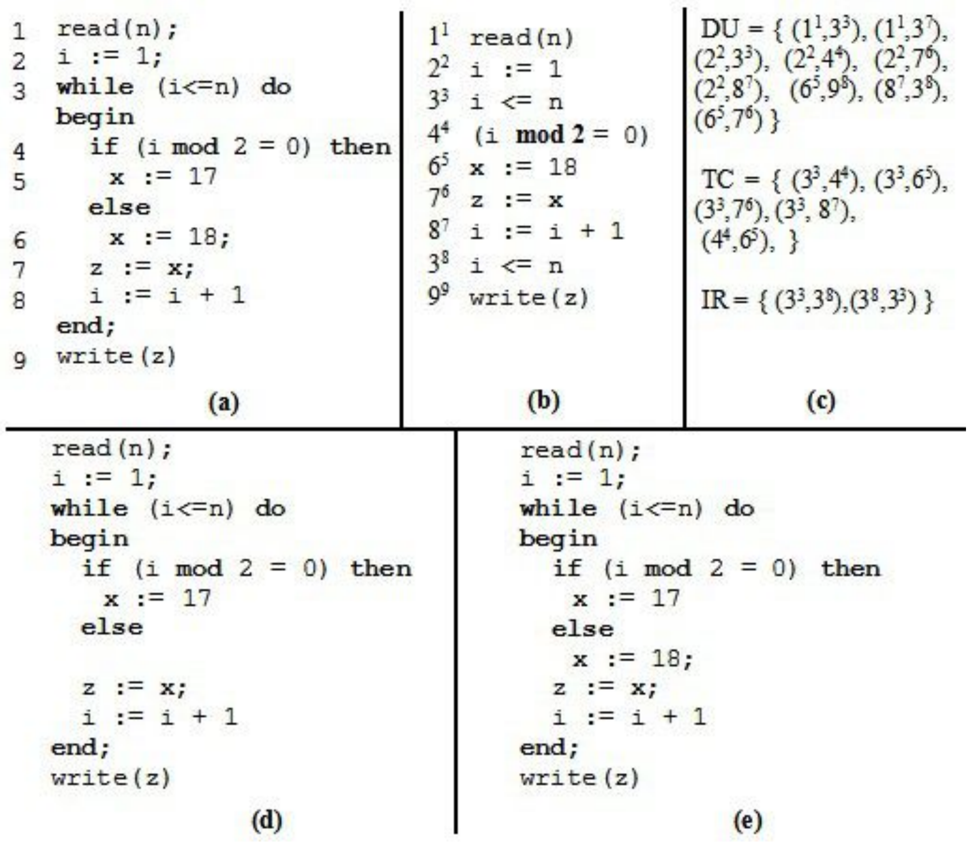
where

$A^{i+1}=\{ X^p \mid X^p \notin S^i, (X^p, Y^t) \in (DD \cup TC \cup IR) \text{ for some } Y^t \in S^i, p < q\}$

In the formalization, X^p is referred to statement *X* at execution position *p*; *LD(q, V)* is the set of last definitions of variables in *V* at execution position *q*, and *LT(I^q)* is the set of test instructions which have control influence on the execution of I^q. The example of this approach is shown in figure 2.5, also the figure gives the comparison of static program slicing.

In the example, dynamic slice on slicing criterion <1, 9^9, z> is computed. In the process of computation, A^0 is $\{7^6\}$ and produced in subsequently $A^1=\{3^3, 2^2, 6^5\}$, $A^2=\{1^1, 3^8, 4^4\}$, A3=$\{8^7, 3^7\}$. From figure 2.5(d) and (e) we can see the different result from dynamic slice and static slice. In static program slicing without assumption of input value, the result slice always contains statement 6, on contrast, dynamic slice get rid of statement 6 because this statement won't be executed when variable *n* equals to 1. In the sake of take account into a specific input, dynamic generate a smaller slice of program. However, Korel and Laski don't give an proof about sufficient of their approach. The trajectory of when input value *n* equals 2 is shown in Figure 2.7 (a). In this time, the dynamic concept follow for this trajectory is :$\{ (6^5, 7^6), (5^{10}, 7^{11}) \} \in DU$, $(7^6, 7^{11}) \in IR$. Statement 6 will be included in slice, however, this statement is not necessary to slice cause by reassignment of variable *z* on statement 5.

Based on Korel and Laski's method, Agrawal and Horgan discussed dynamic program slicing on top of Dynamic Dependency Graph(DDG)[32] where DDG is an extension of PDG that represents for execution history with a specific input. The slice is obtained from traversing all vertices that can be reached from the criterion. Figure 2.8 gives the trajectory for input *n* = 2,corresponding PDG and DDG of the program of in figure 2.7 (a). Gray nodes in figure 2.8 (c) is the slice result on < 2, 9^{14}, z>.



```
1  read(n);
2  i := 1;
3  while (i<=n) do
   begin
4    if (i mod 2 = 0) then
5      x := 17
     else
6      x := 18;
7    z := x;
8    i := i + 1
   end;
9  write(z)
```

(a)

```
1^1 read(n)
2^2 i := 1
3^3 i <= n
4^4 (i mod 2 = 0)
6^5 x := 18
7^6 z := x
8^7 i := i + 1
3^8 i <= n
9^9 write(z)
```

(b)

$DU = \{ (1^1,3^3), (1^1,3^7), (2^2,3^3), (2^2,4^4), (2^2,7^6), (2^2,8^7), (6^5,9^9), (8^7,3^8), (6^5,7^6) \}$

$TC = \{ (3^3,4^4), (3^3,6^5), (3^3,7^6), (3^3, 8^7), (4^4,6^5), \}$

$IR = \{ (3^3,3^8),(3^8,3^3) \}$

(c)

```
read(n);
i := 1;
while (i<=n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else

  z := x;
  i := i + 1
end;
write(z)
```

(d)

```
read(n);
i := 1;
while (i<=n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else
    x := 18;
  z := x;
  i := i + 1
end;
write(z)
```

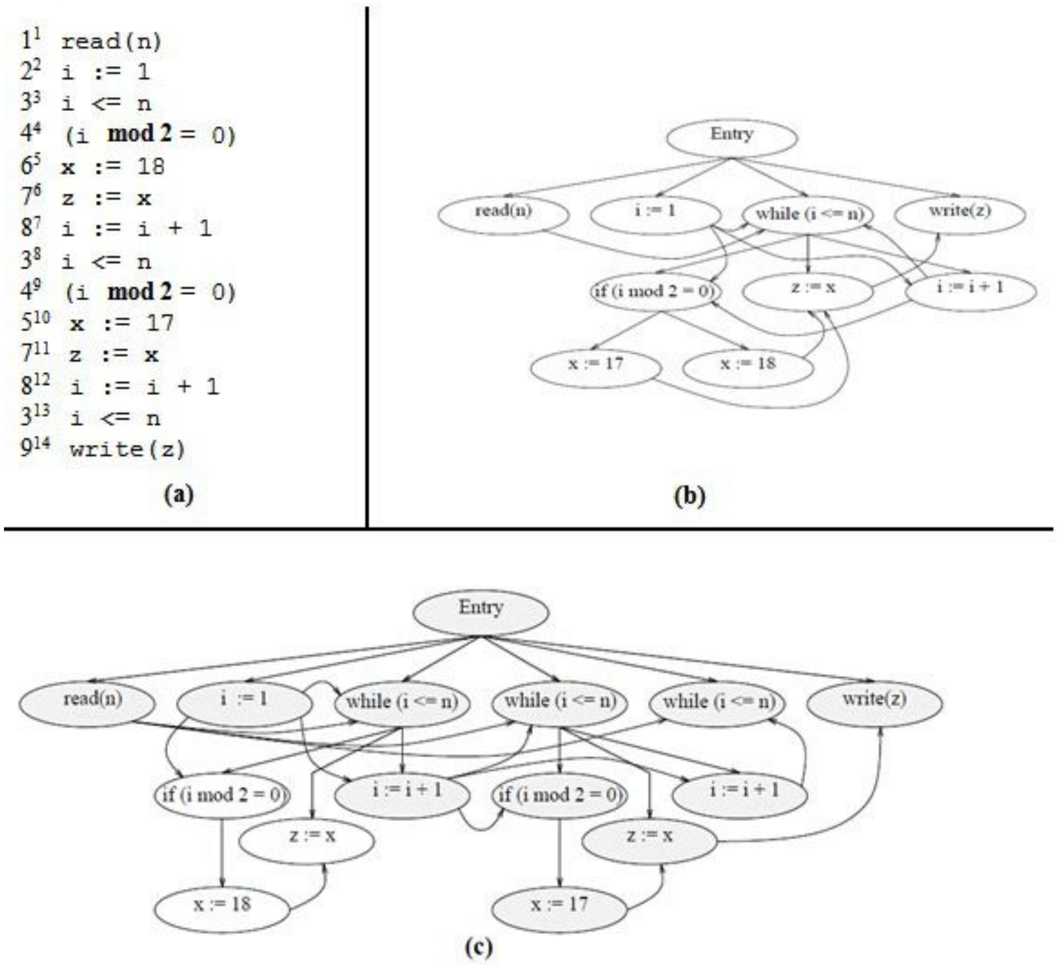
(e)

Figure 2.7 (a) Example code; (b) Trajectory of (a) for input n =1; (c) Dynamic flow concept for this trajectory; (d) Dynamic slice for slicing criterion <1, 9^9, z>; (e) Static slice for slicing criterion <9,z>.

ConSIT is a conditional program slicing tool introduce by Danicic et al.[24]. The slicing criterion is a tuple <*V, n, π*>, where *V* is a set of variables, *n* is the position and *π* is some condition. Their algorithm consists of three phase. The first phase is symbolically execute to propagate state information to each statement in program. The second phase is a theorem proving phase to determine which statement should be eliminate from slice under the input condition. The last phase is carry out traditional static program slicing on conditional program extract from second phase. As they state in conclusion, their approach on conditional is limited on single-procedural program.

2.3 Applications of program slicing

The original purpose of program slicing is for debugging[1, 12]. In the case that the source program produces an erroneous value, backward



1^{1} `read(n)`
2^{2} `i := 1`
3^{3} `i <= n`
4^{4} `(i mod 2 = 0)`
6^{5} `x := 18`
7^{6} `z := x`
8^{7} `i := i + 1`
3^{8} `i <= n`
4^{9} `(i mod 2 = 0)`
5^{10} `x := 17`
7^{11} `z := x`
8^{12} `i := i + 1`
3^{13} `i <= n`
9^{14} `write(z)`

Figure 2.8 (a) Trajectory of the program of figure 2.5 (a) for input n =2; (b) PDG of the program of figure 2.5 (a); (c) DDG of the program of figure 2.5 (a) [3].

slicing enables the backward tracing from the erroneous position to get all statements that have effect on this position. The statement cause erroneous value or some unexpected statement or omissions is likely to occur in the slice. A forward slice can also be used in debugging for two purposes. First, it can query the statements influenced by the slicing criterion; second, investigating what kind of effect on the program's behavior it has when a change on a specific point in the program is made. When slicing with inserting execution, the behavior carried out by statements in the dynamic slice are all dependent on insertion. Dynamic program slicing enables programmers to check if the program exhibits abnormal behavior in the slice.

Later on, the use of program slicing is also extended to other applications.

In software testing, program slices reduce the range of testing. When applying program slicing on data flow testing, each *def-use* pair out of the dependency graph in the slice would have influence on the output value of the program[39]. By comparing the output obtained from the slice with the expected value, programmers can check if the program returns the correct value for the output variable. Program slicing is also useful for regression testing[40, 41]. Both backward and forward slices can be used to determine test portion of program depends on the change. Also, the portion can be used to compare the equivalent behaviors between components of new version and old version of program[25].

Gallagher and Lyle propose applying program slicing for software maintenance[43]. A decomposed slice includes all computations of a variable and also keeps the same behavior as the original program. Thus, the decomposed slices can be seen as independent components of the program and are highly reusable. These components are useful to analyse whether a change in one component influences the behavior of other components. In a similar way, Wang et al.[10] discuss functional independent analysis of web security in the use of program slicing. When merging all slices of each component, if the intersection of two different components is empty, then the error or insecurity result from one component will not effect another one; otherwise, we can't be sure. Program slicing is also useful in program understanding and re-engineering by identifying possible coherent components[11, 44].

Weiser[45] also introduced how program slicing can be used in parallelization. His spliced algorithm requires the set of slices should cover the behavior of original program, the slices can be out of any executable static slice algorithm. Several slices are executed in parallel, the I/O behavior of slices can be spliced together in consistent with original program is preserved.

Beyond traditional program slicing which operate on source code of program, Zhao applies slicing technique on software architecture[14] in assistant of architectural understanding, maintenance and reuse.

2.4 Conclusion

In this chapter, we first introduced Weiser's static program slicing theory: the basic concept and dataflow algorithm. The slices is obtained by deleting statement from original program in such a way that the original behavior is preserved. Intuitionally, static slicing is backward analysis on control flows depends on a interested point in original program to get slices. Then, we introduced the concept of using a PDG to describe data dependences and control dependences between statements of a program and give an example to illustrate static slicing on basic PDG by backward traversal. In the second section, we summarize several different program slicing approaches in two categories: static program slicing, dynamic and conditional program slicing. At the beginning of this section, we also give a short introduction of forward static slice which only different with backward static slice from the direction of tracing dependences. In last of the chapter, we introduce the applications of program slicing in debugging, testing, software maintenance, program analysis and understanding, parallelization, and software architecture.

3

Abstract Interpreter

D. Schmidt[57] said all the world can be understand as an abstract interpretation. Abstraction refers to a process that extracts essence from complicated things; or a process from concreteness to conception. Such as when given the set of words “apples, bananas, pears, grapes, peaches”, we can infer their common feature is fruit, in other word, we use abstraction to describe their property. Another example of abstraction is: “a student is identified by enroll number in their school and is able to sign in a lecture with their enroll number”. On top of this, we can build an abstract interpretation that if the student’s enroll number doesn’t occur in a sign-in list of a lecture, then we can conclude that this student doesn’t participate this lecture. In this case, the abstract interpretation is a prediction of attendance based on abstraction relationship between student and enroll number.

In the domain of computer science, abstraction is a crucial concept that has widespread usage. In Unix, all devices is abstracted to concept of file, the interaction between multiple programs is abstracted as a pipe. Android abstracts out a mobile application as four components: Activity, Intent, Service, Provider. A computer program can be seen as a computation in a domain of abstract concepts. An interpreter is also a computer program that directly executes[49] the behavior of code, normally provided as a list of instructions, by parsing the code and translating it into some form of intermediate representation.

The possible behavior of computation in programs is referred to as semantics. Compare with the lecture enrollment example we mentioned above, an abstract interpreter of program can be informally defined as an approximation of the program's semantics according to its language property. This method is mainly used in program optimization and transformation. By designing new features of an abstract interpreter, we can therefore extend a programming language to deal with new computation pattern. The design of STIP.js is based on the idea of using abstract interpretation for performing static analysis. More concretely, we develop an abstract interpreter for JavaScript based on a CESK machine in the Jipda framework. The role of this abstract interpreter is to generate an intermediate structure of a program to build a state graph. Also, the translation from state graph to PDG is based on the idea of abstract interpreter. The translation is achieved by assessing each node in state graph through an evaluator.

The formal specification of abstract interpretation was first proposed by Cousot P. and Cousot R. in 1977 [46]. In the first section of this chapter, we introduce the Cousot's model of abstract interpretation: syntax and semantics of programs. Then we introduce the notion of static analysis based upon abstraction interpretation. Finally, we introduce the implementation of two kinds of interpreters: one is a machine-based interpreter named CESK machine[53], another one is meta-circular evaluator[54].

3.1 Background knowledge

Syntax[50] describes the rules of a programming language symbolically, while semantics focus on relations between groups of syntax to describe computational behavior. Cousot P. and Cousot R. use the notion of syntax and semantics to describe simple flowchart language[46]. At beginning of [46], Cousot P. and Cousot R. give an intuitive example to explain the concept of abstract interpretation using signs of numbers. Arithmetic computation like -1515*7 can been abstracted as a sign computation, so that (-) * (+) $\Rightarrow$ (-). Even this abstract interpretation leads to imprecision. The rule of calculation on signs, used to predict that the result of the example expression is a negative number is enough to reflect on knowledge about sign operations in a program. Then they elaborate their mathematical syntax and semantics model of program to build an abstract interpretation. Their method, concerned with the underlying program structure, is a reflection of actual execution of a program. We give an overview of their method in this section.

3.1.1 Syntax and semantics of programs

Similar with the graph representation we introduced in chapter 2. Cousots[46] uses a set of "Nodes" and *directed graph*, they call it finite flowchart, to define syntax of a program. For each node in a set there are successors and predecessors. They use *n-pred(n)* define the set of predecessors and use *n-succesor(n)* define the set of succesors of a node. Notation *Arcs* is definition of edges by a pair of node. It denoted as *Arcs <n, m>* where *m* $\in$ *n-successor(n) and n* $\in$ *predecessor(m)*. The node set contains five subsets, the definition for each of node *n* as below:

Entry node: the node has no predecessor *(n-pred(n) =* ϕ *)* and one successor *(n-succ(n) = 1)*.

Assignment node: the node has one predecessor *(n-pred(n) = 1)* and one successor *(n-succ(n) = 1)*. This node represents for the expression *expr* in right hand-side assign its value to identifier *id* in left hand-side.

Test node: the node has one predecessor *(n-pred(n) = 1)* and two successor*(n-succ(n) = 2)*. One is *true* successor *(n-succ-t(n))* another is *false* successor *(n-succ-f(n))*.

Junction node: the node has more than one predecessor *(n-pred(n) > 1)* and one successor *(n-succ(n) = 1)*.

Exist node: the node has one predecessor *(n-pred(n) = 1)* and no successor (*n-succ(n) =* ϕ).

Based on node sets, their *directed graph* is denoted as *<Node, Arcs>* consist of flowing functions:

origin, end: $\forall$ a $\in$ Arcs, a = <origin(a), end(a)>

a-succ: <n, m> where m $\in$ n-succ(n)

a-pred: <m,n> where m $\in$ n-pred(n)

a-succ-t: <n, n-succ-t(n)>

a-succ-f: <n, n-succ-f(n)>

Their basic semantics model is based on complete *lattice,* which is proposed in [51]. A complete *lattice* S^0 is obtained from a domain *S* that has a unique supreme and a unique infimum for all elements *x* inside it. It is denoted as:

$$S^0 = \{\perp_S, \top_S\} \text{ where } \exists x \in S,\ \perp_S < x < \top_S.$$

In Cousot′s model, the complete *lattice* of values is the lattice of *Boolean* addition with other primitive type. For example, the supreme of *Boolean* is *true* and infimum is *false*, supreme of natural number is infinite and infimum is zero. They define the notion of *environment* is binding of identifiers to their values. The set of "State" is a joint of $Arcs^0$ and *environment*. In this case, a state is define as *<cs(s), env(s)>* where *env(s)* is related information in current *environment* and *cs(s)* indicates possible transition. A conditional function *cond(b,* e_1*,* e_2*)* is denotes for if *b* then e_1, else e_2. Review the *directed graph* we introduced above, the possible transition of state of their basic model consist of:

Assignments →

<a-succ-(n), evaluate expression with binding identifier>

Tests →

cond(predicate, <a-succ-t(n), env(s)> , <a-pred-f(n), env(s)>)

Junctions → <a-succ(n), env(s)>

Exists → s

3.1.2 Static semantics of programs

The above theory models program execution as a graph of states. On top of this, Cousot P. and Cousot R. define two notations, *context* and *context-vector*, which also respect to the conclusion in [52] about static properties of a program that each program point is concerned a set of associated states. Primarily, a sequence states has a initial state. *Context* is defined as the environments involved in a sequence state transition from initial state associated to point *q*. *Context-vector* is a context of each points in program to summarize possible execution of program. Their definition of abstract interpreter for static semantics of program is a tuple:

$$I = < \text{Contexts}, \bigcup, \subseteq, \text{Env}, \phi, \text{n-context}>$$

where *Env* is supreme and ϕ is infimum of *Contexts*. $\subseteq$ denotes relation between *a*, *b* in *Contexts* where, $\exists\, a, b \in$ *Contexts*, we have $a \subseteq b \Leftrightarrow a \bigcup b = b$.

3.2 Static analysis based on abstract interpretation

The model we introduced above is the analysis of all possible executions for a point in the program. On step further, there is a method to implement abstract interpretation by using abstract semantics to simulate the computation traces of concrete semantics. Concrete semantics is sequence of statements which describe the meanings of program. These statements are in a particular order that carry out an execution with an unevaluated identifier, also referred to as symbolic execution of a program. Based on symbolic execution, static analysis is the most common way to make most precise abstract interpretation.

First of all, we have to introduce the notion of a Galois connection. A Galois connection is build on top of a value abstraction. A graphic illustration of an example value abstraction is in figure 3.1.

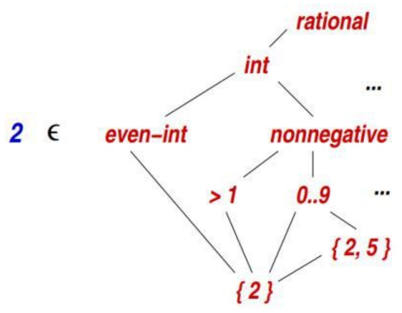


Figure 3.1 Value abstractions[57]

The right hand side lists all the properties that are abstractions of concrete value number 2. We can see the properties connected with upward lines became general and loss precision. This attribute of abstract is denoted by $\sqsubseteq$. For every value abstraction, we also have a supreme $\top$, and infimum $\bot$. The infimum of abstraction is *{2}*, and supreme is *rational* in above example.

A function γ is the mapping from each abstract value to the set of concrete values it represents. A function α maps each set of concrete values to the abstract value that best describes it. Following figures show an example of functions γ and α . $\gamma \circ \alpha$ as a set of mapping relations between concrete and abstract values in a numerical domain.

From figure 3.2(b) we can see that using an abstract value to describe a set of concrete values will result in a loss of accuracy. However α by definition makes the description as precise as it possibly can be. The Galois connection formalizes the situation, and the graphical explanatory of it is in figure 3.3.

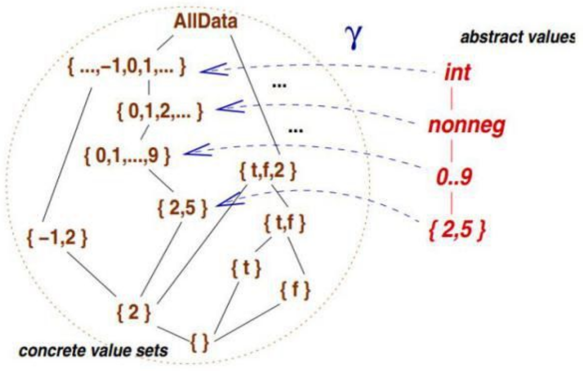


Figure 3.2 (a) Function γ [57]

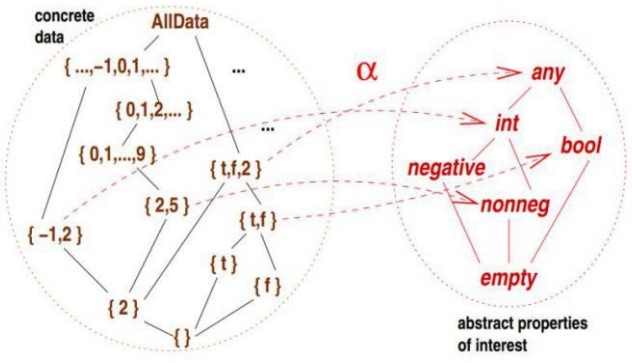


Figure 3.2 (b) Function α [57]

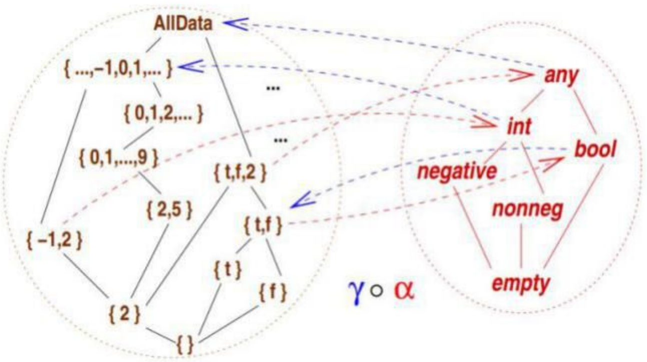


Figure 3.2 (c) $\gamma \circ \alpha$ [57]

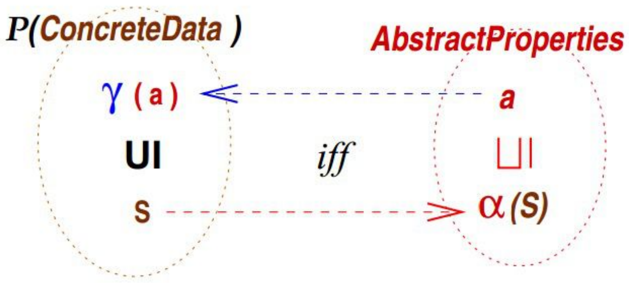


Figure 3.3 Formalization of Galois connection[57]

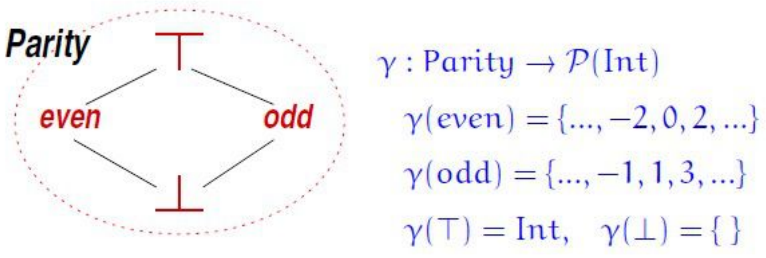
The definition is, for all S ∈ P(ConcreteData), a ∈ AbstractProperties,

$$S \in \gamma(a) \text{ iff } \alpha(s) \sqsubseteq a$$

When γ and α monotone, this is equivalent to

$$S \subseteq \gamma \circ \alpha(S) \text{ and } \gamma \circ \alpha(a) \sqsubseteq a .$$

Using a Galois connection we can interpret concrete semantics with abstract values, as shown in the example below[57] :



$$\gamma : \text{Parity} \rightarrow \mathcal{P}(\text{Int})$$
$$\gamma(\text{even}) = \{\ldots, -2, 0, 2, \ldots\}$$
$$\gamma(\text{odd}) = \{\ldots, -1, 1, 3, \ldots\}$$
$$\gamma(\top) = \text{Int}, \quad \gamma(\bot) = \{\,\}$$

Figure 3.4 Example of abstract interpretation based on Galois connection[57]

This is an abstract interpreter that judges the parity of an integer. The abstraction value set contains two value to indicate the category of an integer: one is *even*, another is *odd*. The supreme of this value abstraction is whole set of integers, and the infimum is an empty set. Function γ is mapping the category value in the set of abstraction to its concrete value set. Function α in this example is defined as P(Int), to predicates an concrete number weather is an odd or an even basic the rule of $\{ \beta(v) \mid v \in S\}$, where $\beta(2n)$ = even and $\beta(2n+1)$ = odd.

Based on the abstract interpretation of semantics in above example, we can predict the computation traces of concrete semantics. For example, consider the following code:

```
S0 | while( x div 2 == 0) {
S1 |         x = x/2;
   | }
S2 | x=x+1;
S3 | return x;
```

In this program fragment we know that if the value of *x* in S_0 is even, the program will go into the body of loop, otherwise it will jump to S_2. Assuming the operation of program is denoted by a pair of a program point and the abstract value of variable, we can get two trace trees from basic on underlying abstract interpretation semantics as shown in figure 3.5:

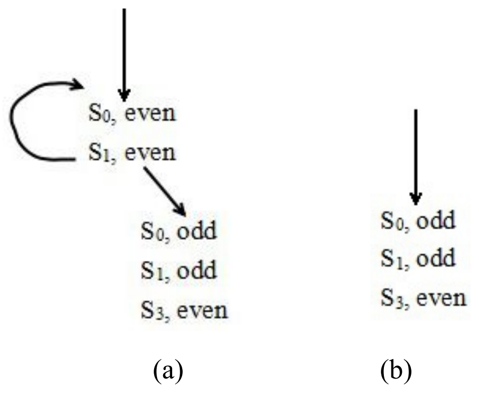


Figure 3.5 (a)Trace trees if input value is even; (b)Trace tree if input value is odd.

To ensure every transition in concrete semantics is simulated by one in the abstract semantics, we have to follow this principle: if a concrete variable *c* at S_i can transit to *c'* at S_j, then we have to ensure there is a corresponding transition in abstract interpreter from a within S_i to a' within S_j, where S_j, c' $\in \{S_j, c' \mid c' \in \gamma(a')\}$.

So far, we know a static analysis of a program is a sound, finite, and approximate calculation of the program's execution. Additional, static analysis is an important method to deal with halting problem. This kind of problem is generated by undecidable termination by a given input. In the parity example, because of loss precise on abstraction, we can't say we get termination for all even values, even if only 0 will cause non-termination. In other word, static analysis attempts to predicate program behavior as precise as possible.

Beside above solution of static program analysis based on abstract interpretation, in the following section of this chapter, we'll introduce two types of abstract interpreters that can be used as a framework to perform static analysis.

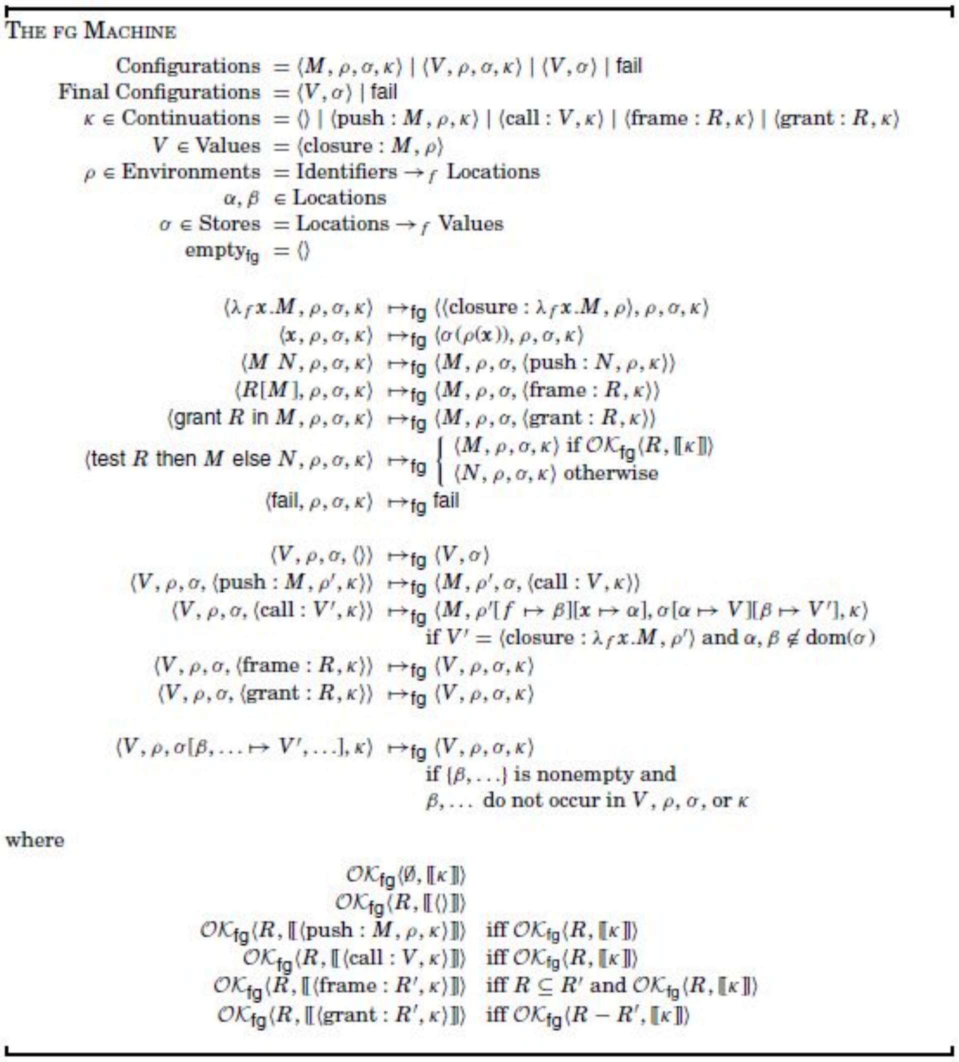
3.3 CESK-Machine

Mtthias Felleisen[55] introduced CESK machine to translate λ semantics which they also call it *frame-generating* or *fg*. It includes four component: a (C)ontrol component, a (E)nvironment, a (S)tore and a (K)ontinuation. The control component can be seen as instructions based on the programming language being reduced. In functional programming language, this component can be expressions. Environment and store can be a map or a first-class function. Environment is meaning the machine will assign an address in storage to each variables. The store is used to map from address to value. The continuation component K is the stack of program, it represents for instruction's control context.

Their purpose to develop such kind of machine interpreter is to translate a source language, which defines source component, to a target language which augment source language with *frame expression* to specify the permission of component Their model based on λ -calculus language and allowing explicit name λ -calculus. The intersection of appropriate permissions *R* in caller and callee λ -expression used to ensure the translation is reliable. They also define an *Eval* function which is a combination of each permission of component in source to determine the meaning of source program. In *fg* machine, the role of *Eval* function is abstract context transition, denoted as $\mapsto_{fg}$. The following figure illustrate the structure of *fg* machine.

In the first part, is their definition of principle of evaluation sub-expression and extending corresponding continuation. Symbol *M* denotes for target language. The basic call-by-value λ -calculus is attached with *test expression* to check the branch of enabled permission, *grant expression* enables privilege, and *fail expression* direct halt to security failure.

In figure, the second group in definition of *fg* machine is a complementary of part one to determine the transition. The evaluated program, *R[M],* is target language adds a framing expression to source language. $\mathcal{OK}$ *is* predicate applied on current context and desired permissions. The last part of their machine is garbage collection in storage. The *Eval* function takes charge of validating target in a valid machine configuration, and also replaces variables by their bound value in environment.



THE FG MACHINE

$$\begin{aligned}
\text{Configurations} &= \langle M,\rho,\sigma,\kappa\rangle \mid \langle V,\rho,\sigma,\kappa\rangle \mid \langle V,\sigma\rangle \mid \mathsf{fail}\\
\text{Final Configurations} &= \langle V,\sigma\rangle \mid \mathsf{fail}\\
\kappa \in \text{Continuations} &= \langle\rangle \mid \langle \text{push} : M,\rho,\kappa\rangle \mid \langle \text{call} : V,\kappa\rangle \mid \langle \text{frame} : R,\kappa\rangle \mid \langle \text{grant} : R,\kappa\rangle\\
V \in \text{Values} &= \langle \text{closure} : M,\rho\rangle\\
\rho \in \text{Environments} &= \text{Identifiers} \rightarrow_f \text{Locations}\\
\alpha,\beta &\in \text{Locations}\\
\sigma \in \text{Stores} &= \text{Locations} \rightarrow_f \text{Values}\\
\text{empty}_{\mathsf{fg}} &= \langle\rangle
\end{aligned}$$

$$\begin{aligned}
\langle \lambda_f x.M,\rho,\sigma,\kappa\rangle &\mapsto_{\mathsf{fg}} \langle\langle \text{closure} : \lambda_f x.M,\rho\rangle,\rho,\sigma,\kappa\rangle\\
\langle x,\rho,\sigma,\kappa\rangle &\mapsto_{\mathsf{fg}} \langle \sigma(\rho(x)),\rho,\sigma,\kappa\rangle\\
\langle M\ N,\rho,\sigma,\kappa\rangle &\mapsto_{\mathsf{fg}} \langle M,\rho,\sigma,\langle \text{push} : N,\rho,\kappa\rangle\rangle\\
\langle R[M],\rho,\sigma,\kappa\rangle &\mapsto_{\mathsf{fg}} \langle M,\rho,\sigma,\langle \text{frame} : R,\kappa\rangle\rangle\\
\langle \mathsf{grant}\ R\ \mathsf{in}\ M,\rho,\sigma,\kappa\rangle &\mapsto_{\mathsf{fg}} \langle M,\rho,\sigma,\langle \text{grant} : R,\kappa\rangle\rangle\\
\langle \mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N,\rho,\sigma,\kappa\rangle &\mapsto_{\mathsf{fg}} \begin{cases} \langle M,\rho,\sigma,\kappa\rangle \text{ if } \mathcal{OK}_{\mathsf{fg}}\langle R,[\![\kappa]\!]\rangle\\ \langle N,\rho,\sigma,\kappa\rangle \text{ otherwise}\end{cases}\\
\langle \mathsf{fail},\rho,\sigma,\kappa\rangle &\mapsto_{\mathsf{fg}} \mathsf{fail}
\end{aligned}$$

$$\begin{aligned}
\langle V,\rho,\sigma,\langle\rangle\rangle &\mapsto_{\mathsf{fg}} \langle V,\sigma\rangle\\
\langle V,\rho,\sigma,\langle \text{push} : M,\rho',\kappa\rangle\rangle &\mapsto_{\mathsf{fg}} \langle M,\rho',\sigma,\langle \text{call} : V,\kappa\rangle\rangle\\
\langle V,\rho,\sigma,\langle \text{call} : V',\kappa\rangle\rangle &\mapsto_{\mathsf{fg}} \langle M,\rho'[f \mapsto \beta][x \mapsto \alpha],\sigma[\alpha \mapsto V][\beta \mapsto V'],\kappa\rangle\\
&\qquad \text{if } V' = \langle \text{closure} : \lambda_f x.M,\rho'\rangle \text{ and } \alpha,\beta \notin \text{dom}(\sigma)\\
\langle V,\rho,\sigma,\langle \text{frame} : R,\kappa\rangle\rangle &\mapsto_{\mathsf{fg}} \langle V,\rho,\sigma,\kappa\rangle\\
\langle V,\rho,\sigma,\langle \text{grant} : R,\kappa\rangle\rangle &\mapsto_{\mathsf{fg}} \langle V,\rho,\sigma,\kappa\rangle
\end{aligned}$$

$$\begin{aligned}
\langle V,\rho,\sigma[\beta,\ldots \mapsto V',\ldots],\kappa\rangle &\mapsto_{\mathsf{fg}} \langle V,\rho,\sigma,\kappa\rangle\\
&\qquad \text{if } \{\beta,\ldots\} \text{ is nonempty and}\\
&\qquad \beta,\ldots \text{ do not occur in } V,\rho,\sigma, \text{ or } \kappa
\end{aligned}$$

where

$$\begin{aligned}
&\mathcal{OK}_{\mathsf{fg}}\langle \emptyset,[\![\kappa]\!]\rangle\\
&\mathcal{OK}_{\mathsf{fg}}\langle R,[\![\langle\rangle]\!]\rangle\\
&\mathcal{OK}_{\mathsf{fg}}\langle R,[\![\langle \text{push} : M,\rho,\kappa\rangle]\!]\rangle && \text{iff } \mathcal{OK}_{\mathsf{fg}}\langle R,[\![\kappa]\!]\rangle\\
&\mathcal{OK}_{\mathsf{fg}}\langle R,[\![\langle \text{call} : V,\kappa\rangle]\!]\rangle && \text{iff } \mathcal{OK}_{\mathsf{fg}}\langle R,[\![\kappa]\!]\rangle\\
&\mathcal{OK}_{\mathsf{fg}}\langle R,[\![\langle \text{frame} : R',\kappa\rangle]\!]\rangle && \text{iff } R \subseteq R' \text{ and } \mathcal{OK}_{\mathsf{fg}}\langle R,[\![\kappa]\!]\rangle\\
&\mathcal{OK}_{\mathsf{fg}}\langle R,[\![\langle \text{grant} : R',\kappa\rangle]\!]\rangle && \text{iff } \mathcal{OK}_{\mathsf{fg}}\langle R - R',[\![\kappa]\!]\rangle
\end{aligned}$$

Figure 3.6 Definition of *fg* machine[55].

[53] gives an overview of implementation of a simple CESK machine for A-Normalized lambda calculus. They review the machine-based interpreter of program is model source program by group statements with its execution within an environment. The environment is a immutable hash maps that mapping between variables to values. In general, a machine-based interpreter should contains, a set of program used to be abstracted, a set of machine states, an *injection* function used to initial the state of program and a *step* function used to transit state. They assuming the terminate of state transition is deterministic, and for undecidable termination we have to set a function to generate multiple potential successor states. There are three kinds of expression used to evaluated: atomic expression (*aexp*), complex expression (*cexp*) and expression (*exp)*. Atomic expression is the expression which always has terminate; a complex can contains error, side effect, or defer execution such as conditional statement; expression can be atomic, complex or let-bound.

The component of their CEK-machine is followed Felleisen's design. Control string is expression. The environment maps variable to its address and a store maps address to values. The trace of variable query in CESK-machine is first through some environment and then through the store to get value. Their machine includes five type of value: void, booleans, integers, closures and first-class continuation. The set of value is defined as:

$$val \in \text{Value} ::= \mathbf{void} \mid \mathbf{z} \mid \mathbf{\#t} \mid \mathbf{\#f} \mid \mathbf{clo}(\text{lam}, \rho) \mid \mathbf{cont}(\text{k})$$

where **z** is integer, **#t** and **#f** are boolean.

By the assumption that a continuation execution is always triggered by the result it awaiting for, continuation can deal with sub-expression in *let-bound* expression by keeping resume execution. For instance, given a *let-bound* expression (*let* ([*v exp*]) *body*), the machine will first evaluated *exp* and bind the computation result to *v*, then holding the resume to *body*. A special initial continuation triggered by result of program is called halt. Finally, their continuation includes address information, that is the scope of variable.

They elaborate their implementation from two-stage. The first is evaluate on *aexp* with auxiliary semantic function: $\mathcal{A}$: AExp $\times$ Env $\times$ Store $\rightarrow$ Value. The evaluate result of integers and booleans is themselves: $\mathcal{A}(\text{z}, \rho, \sigma) = \mathbf{z}$; $\mathcal{A}(\mathbf{\#t}, \rho, \sigma) = \#\text{t}$; $\mathcal{A}(\mathbf{\#f}, \rho, \sigma) = \#\text{f}$. Lambda becomes to closures: $\mathcal{A}(\text{lam}, \rho, \sigma) = \text{col}(\text{lam}, \rho)$. By use function O(*prim*) to define mapping from primitive operation to its corresponding operation, primitive expression are evaluated recursively:

$$\mathcal{A}((prim\ aexp_1,\ \ldots\ aexp_n\ ), \rho,\ \sigma\ ) =$$
$$\mathrm{O}(prim)\langle \mathrm{A}\,(\mathrm{prim\,aexp_1}, \rho, \sigma), \ldots, \mathrm{A}\,(\mathrm{prim\,aexp_n}, \rho, \sigma)\rangle\ .$$

In second stage, they use *step* function to evaluate each expression type. When evaluation of procedure, the return statement of each procedure is an *atomic* expression and return the result to the continuation is waiting for its result. The procedure call is evaluated the expression of invocation first and the expressions in arguments, and then evaluate the body of procedure with current arguments. The first class continuation as call-with-current-continuation (call/cc) in functional program, is evaluated the current continuation as first-class procedure, and returns the value to the continuation of the call/cc application. The next step of conditional expression is evaluation by predicate expression. Overwrite is done by searching the address and modify the value of it. Recursive is valuated by self-reference which is the extending of environment. The *step* function is defined as:

$$step((letrec([v_1\ aexp_1]\ldots[v_n\ aexp_n])\ body), \rho, \sigma, \kappa) = (body, \rho', \sigma', \kappa),$$

where: $a_1, \ldots, a_n$ are fresh addresses in σ

$$\rho' = \rho[v_i \mapsto a_i]$$
$$value_i = \mathrm{A}(aexp_i, \rho', \sigma)$$
$$\sigma' = \sigma[a_i \mapsto value_i].$$

The two auxiliary function, one is *applyproc* is used to apply a procedure to value, another one is *applykont* is used to apply a continuation to a return value. The definition of two function are:

$$applyproc(\ \mathbf{clo}\ ((\lambda\ (\mathrm{v_1 \ldots v_n})\ \mathrm{body}), \rho)\ \langle \mathrm{value_1}, \ldots \mathrm{value_n} \rangle, \sigma, \kappa) =$$

$$(\mathrm{body}, \rho', \sigma', \kappa),\ \text{where}$$

$\mathrm{a_1}, \ldots, \mathrm{a_n}$ are fresh addresses in σ

$$\rho' = \rho[v_i \mapsto a_i]$$
$$\sigma' = \sigma[a_i \mapsto value_i].$$

$$applykont(\ \mathbf{letk}\ (v, e, \rho, \kappa), value, \sigma) = (e, \rho[v \mapsto a], \sigma[\ a \mapsto value], \kappa),$$

where $a \notin dom(\sigma)$ is a fresh address.

3.4 Meta-circular evaluator

A meta-circular evaluator is a special case of a self-interpreter [53] fit for the programming language whose program structure is similar to its syntax. In a meta-circular evaluator, the evaluator is a program to determine the meaning of expressions written in the same language with its source program. The chapter 4 of [55] is a tutorial of how to extend languages feature by building an evaluator in LISP and its dialect. Their meta-circular implementation abides by *eval-apply cycle*, as shown in figure 3.7.

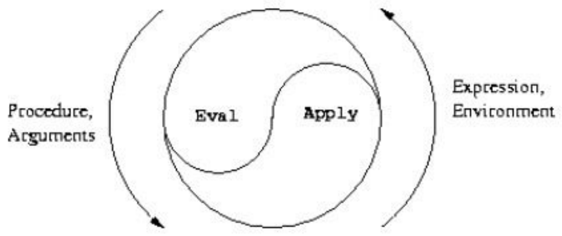


Figure 3.7 The *eval-apply cycle* exposes the essence of a computer language[55].

The essence of *eval-apply cycle* is that procedure is reduced to new expressions to be applied arguments within new environments, recursive evaluation of sub-expression in combination, until we get symbols. The core of evaluation consist of two procedure: *eval* and *apply*. The process of evaluation is interplay between *eval* and *apply*.

Eval procedure receiving an expression and an environment arguments is implemented as a case analysis using `cond`. The evaluation is divided into three test case. First one is primitive expression that the evaluator will return itself. This is a combination between value and environment. Then the author define several cases of special form which consists of quoted expression, assignment, `if` expression, `lambda` expression, `begin` expression, case analysis (`cond`). Quoted expression is returned with the expression that was quoted in *eval*; an assignment is recursively compute new value to be associated with the variable; `if` expression is evaluated according to the predicates; a `begin` expression is evaluated in the order of sequence of expressions; a `cond` is processed as `if`. The last one is a combination that must recursively evaluate the operator part and the operands in procedure application. The definition of *eval* in [55] is:

```
(define (eval exp env)
 (cond ((self-evaluating? exp) exp)
      ((variable? exp) (lookup-variable-value exp env))
      ((quoted? exp) (text-of-quotation exp))
      ((assignment? exp) (eval-assignment exp env))
      ((definition? exp) (eval-definition exp env))
      ((if? exp) (eval-if exp env))
      ((lambda? exp)
        (make-procedure (lambda-parameters exp)
                        (lambda-body exp)
                        env))
      ((begin? exp)
       (eval-sequence (begin-actions exp) env))
       ((cond? exp) (eval (cond->if exp) env))
      ((application? exp)
       (apply (eval (operator exp) env)
              (list-of-values (operands exp) env)))
      (else
        (error "Unknown expression type -- EVAL"
      exp))))
```

In the definition, `list-of-values` is used to return the value of each operand in expression; `eval-if` evaluates on corresponding consequence of predicates. The `eval-sequence` takes a sequence of expressions withn an environment to evaluated in order and then returns the value of final expression. `Eval-assignment` and `eval-definition` defines in similar way to handles assignment to variable, both of them call `eval` to look up the value to be assigned; the former set resulting value of variables; the later set definition. The list of these procedure definition[55] is:

```
(define (list-of-values exps env)
   (if (no-operands? exps)
     '()
       (cons (eval (first-operand exps) env)
       (list-of-values (rest-operands exps) env))))


(define (eval-if exp env)
   (if (true? (eval (if-predicate exp) env))
        (eval (if-consequent exp) env)
        (eval (if-alternative exp) env)))
```

```
(define (eval-sequence exps env)
   (cond ((last-exp? exps) (eval (first-exp exps)
                                 env))
         (else (eval (first-exp exps) env)
         (eval-sequence (rest-exps exps) env))))


(define (eval-assignment exp env)
   (set-variable-value! (assignment-variable exp)
                        (eval (assignment-value exp)
                                    env)
                         env)
   'ok)


(define (eval-definition exp env)
   (define-variable! (definition-variable exp)
                     (eval (definition-value exp) env)
                    env)
  'ok)
```

Apply also takes two arguments, a procedure and a list of argument that will be used in this procedure. The procedure is divide into two types, one is a primitive procedure, another one are compound procedures that require to be evaluated its body by extension of environment of procedure execution with a binding of argument and parameter of procedure. The definition of *apply*[55] is:

```
(define (apply procedure arguments)
      (cond ((primitive-procedure? procedure)
            (apply-primitive-procedure procedure
                               arguments))
      ((compound-procedure? procedure)
       (eval-sequence
            (procedure-body procedure)
            (extend-environment
              (procedure-parameters procedure)
              arguments
              (procedure-environment procedure))))
      (else
      (error
          "Unknown procedure type -- APPLY"
procedure))))
```

3.5 Conclusion

In this chapter, we started with a real-world example explaining the notion of abstract interpretation, and briefly introduce how it is used in computer science. By computation on the abstracted concept, the abstract interpreter of program is also a program in the use of process code for analysis and reuse. Then we stated the static semantic model proposed by Cousot to build an abstract interpretation of a program. This model contains two parts. The first part abstracts the syntax and semantics of a program by finite flowchart, and integrate the semantic model with a lattice to approximate the behavior. Then the possible behavior of program is summarized by computing the state graph during execution. In the second part of this chapter, we introduced static analysis, which is most widely used pattern in setting up abstract interpretation. In the same section, we mainly introduced Galois connection, which is a common formalism to establish a connection between concrete data and abstract data. In the final part of this chapter, we discussed essential design issues of two types of abstract interpreters that can be used as a framework to perform static analysis. One is a machine based interpreter proposed by Felleisen. Here, we mainly introduced the idea of C, E, S, K components. Another one is meta-circular evaluator, and we gave an overview of essential process to evaluate a language by *eval-apply* cycle. Both of these two styles of interpreters are interpreted by the same language that they are interpreting themselves. In view of these two styles of interpreter, we can further design an abstract interpretation in different applications. As we mentioned in the beginning, our tier splitting tool is erected on CESK-machine in meta-circular interpreter style and we'll give more details about our design in next the chapter. Also, we will elaborate our on implementation of a new slicing algorithm JipdaSlicer.

4

Program Slicing on State Graph

By building a PDG on top of the Jipda framework, the current version of STIP.js is able to calculate slices on the PDG. The slices can be used to generate distributed code to complete the tier splitting. Our program slices are based on Weiser's static slicing theory. This chapter will introduce the new algorithm, JipdaSlicer, to get JavaScript slices from a state graph directly. JipdaSlicer is a new program slicing tool on top of the original version of STIP.js. It can slice JavaScript code on the state graph directly, without building a PDG. In the first section of this chapter, we explain how the original STIP.js tool completes the tier splitting via two phases. The first phase generates a state graph by Jipda framework, the second phase is backward slicing on the PDG constructed out of the state graph to generate distributed code. After reviewing the structure of the current tool, we give some case analysis on the state graph in the Jipda framework to illustrate how we can find data dependencies and control dependencies on the state graph. By observing the state graph, we set three assumptions as essential guidelines of the implementation design. Finally, we introduce our implementation design of JipdaSlicer, which permits to extract slices from a state graph without building any intermediate representation.

4.1 Introduction of STIP.js

In the original version of STIP.js, tier splitting consists of two phases. The first phase is converting source code into a state graph by the Jipda framework. In the second phase, it constructs a PDG on top of the state graph, then completes tier splitting by static program slicing. In following we give more details about how STIP.js works.

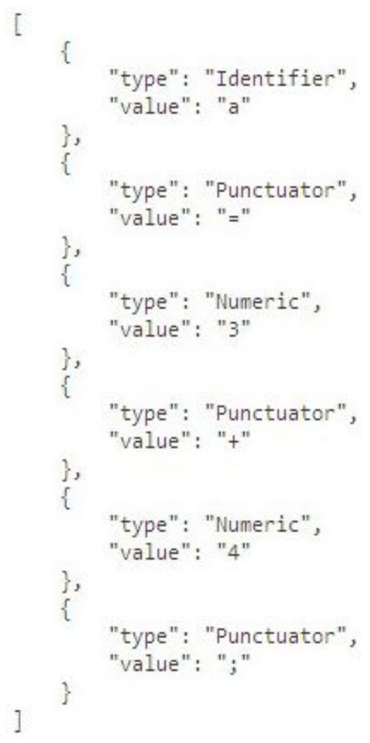
4.1.1 Jipda framework

The analysis done by the Jipda framework on the source code results in a state graph. A state graph represents a program through a sequence structure graph to indicate finite representation of behavior of the program. The first step is to parse the code, achieved by the Esprima.js tool[47]. Esprima.js is a parser of JavaScript used to convert source code into abstract syntax tree[48]. Such a syntax tree is a tree structure representation of program code which is context-free. A syntax tree describes composition and structure of grammatical features of the programming language and implicit unnecessary information such as parentheses. Beside, Esprima.js also results in *tokens* of source code. *Tokens* are lexical information of program code. In general, the structure of a *token* is a pair of "type" and "value" to identify the lexical attribute of each element. Here, we give an example to explain the structure of sensible syntax tree generated by Esprima.js. If the input code is:

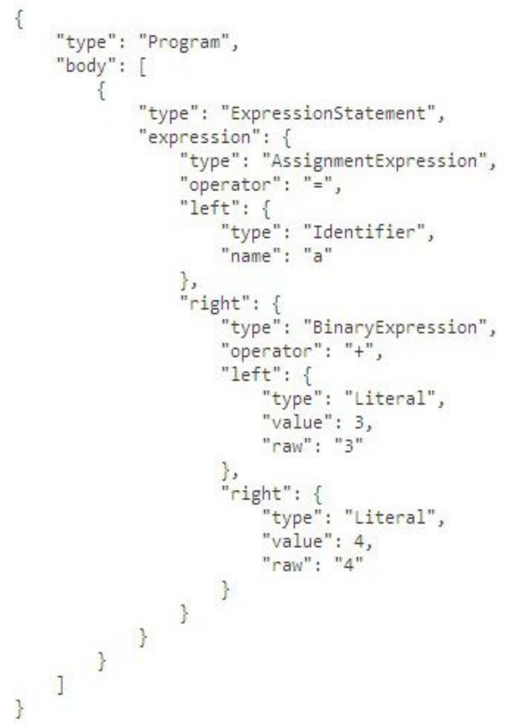
```
a = 3+4;
```

Intuitionally, this is an assignment of arithmetic expression to operand *a*. Figure 4.1 shows parse analysis result of this code out of Esprima.js. In (a) is the list of *tokens* of this assignment. On left of expression, variable a is lexing as an *Identifier* with value *a*; on the right of expression, 3 and 4 is a *Numeric* type with integrate value. All the operators is classified as *Punctuator*. Figure 4.1(b) gives syntax information of the code in JavaScript.

After we get the parsed code, our tool invokes the abstract interpreter which implements a static analysis based on CESK-machine mechanism. Looking back to the definition we introduced in section 3.3, our CESK-machine contains four components. The control string in this interpreter is expressions. When the interpreter is launching, the CESK-machine installs global pointers and references. This component is used to map a variable to its address. The launching also leads to initiate storage, this component maps addresses to its value. The continuation component suffix with *Kont* is the program stack to track computation and operation information of the program. For each *Kont,* it contains two basic variables: *node* and *benva*.



```
[
    {
        "type": "Identifier",
        "value": "a"
    },
    {
        "type": "Punctuator",
        "value": "="
    },
    {
        "type": "Numeric",
        "value": "3"
    },
    {
        "type": "Punctuator",
        "value": "+"
    },
    {
        "type": "Numeric",
        "value": "4"
    },
    {
        "type": "Punctuator",
        "value": ";"
    }
]
```



```
{
    "type": "Program",
    "body": [
        {
            "type": "ExpressionStatement",
            "expression": {
                "type": "AssignmentExpression",
                "operator": "=",
                "left": {
                    "type": "Identifier",
                    "name": "a"
                },
                "right": {
                    "type": "BinaryExpression",
                    "operator": "+",
                    "left": {
                        "type": "Literal",
                        "value": 3,
                        "raw": "3"
                    },
                    "right": {
                        "type": "Literal",
                        "value": 4,
                        "raw": "4"
                    }
                }
            }
        }
    ]
}
```

(a) (b)

Figure 4.1 (a) tokens of assignment a=3+4;; (b) syntax tree structure of assignment a=3+4;.

Node is the syntax and *benva* is the address. This component is also used to associate the scope information of variables and a continuation of the current continuation. The resume of an execution in this interpreter is kept by *body stack (BodyKont)*. The functions suffix with *State* is associated to calculate state transitions. Then apply static analysis on CESK-machine to get *halt*, *push* and *pop* edges with address information in the state graph to reflect behavior of the program.

```
var a=1;
var b=a+1;
```

Figure 4.2 Example code

Figure 4.3 shows the analysis result derived by the above code. From the figure we can see that the node in the state graph is either an element of the syntax tree or a node containing one or more statements. The digit in parentheses on the edges is the edges' number. Each edge also contains stack information of the abstract interpreter. This consists of a *tag* of a

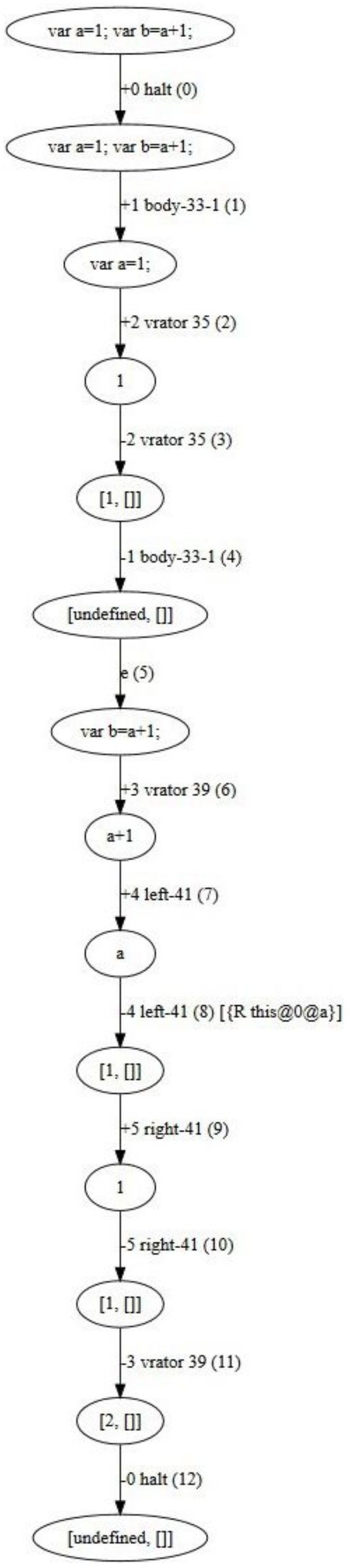


Figure 4.3 Corresponding state graph of code in figure 4.2.

node. This *tag* can be seen as an *id* of each *block*. The *block* in a state graph is the node between corresponding positive and negative edges. This node may be a function declaration or an operand in code. A number of *blocks* compose more complex semantics, such as variable declaration, function call. For example, statement in first line is identified with two *blocks*: node between edge +1 and -1 identify the value of variable, then edge +2 and -2 identify variable declaration statement. For some edges, it also contains reference and scope information such as the 8th edge. 8th edge tells us, this node refers to an identifier whose value is *a*, and the scope of *a* is point to global variable. The edges labeled with positive digit denote for the operation of *push stack*; on contrast, negative digit denotes for operation of *pop stack*. In our CESK-machine, *push stack* indicates the start of evaluation of an identifier or a statement; *pop stack* indicates the evaluation is terminated. Each corresponding positive and negative digit pair is used to identify semantics of the program.

4.1.2 Constructing PDG on top of state graph

The PDG in the tier splitting tool is based on a classic PDG augmented with summary edges. The summary edges are added between input actual parameter to output actual parameter if a path between the corresponding formal input to formal output exists. The constructing process is impleme-nted by the notion of *an evaluator* we introduced in section 3.4. By using the *evaluato*r to estimate the type of nodes in the state graph, the abstract interpreter will result in corresponding adding operation of node as well as the connection with its predecessors and successors in PDG. The type of node in PDG is identified by corresponding positive and negative digit pair of edge in state graph.

For example, in figure 4.3, the edge labeled *+2* and *-2* is stamped with string *vrator*, this string indicates the node between these edges is in *variable declaration stack (VariableDeclarationKont)*. The corresponding edges in *body stack* identify the first variable declaration statement in code and outgoing edge to second line. The *evaluator* in STIP.js handles this information from state graph and makes a statement node for a variable declaration in the PDG. The data dependence in figure 4.3, is identified with scope information in 8th edge. Following two figures show the PDG out of STIP.js. To get slices, the program slicing algorithm is backward traversal as we discussed in section 2.2. The following two figures show the PDG we get from Stip.js.

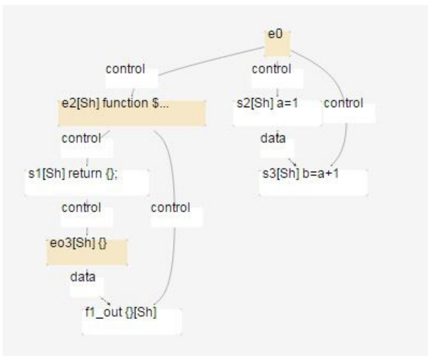


Figure 4.4 Corresponding PDG of code in figure 4.2.

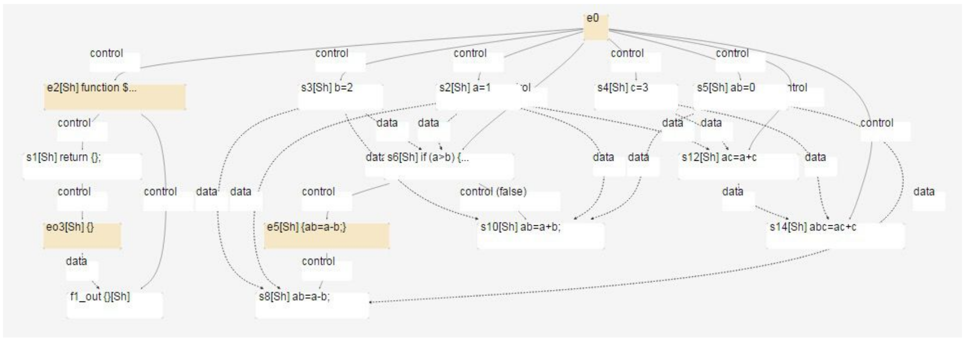


Figure 4.5 Corresponding PDG of code in figure 2.1(a).

4.2 Case study on state transition graph

Beside semantic links between nodes, the predecessors and successors of each node are linked depending on data and control dependences when we construct the dependence graph. So when we find the slicing node that corresponds to the slicing criterion, we can simply get all nodes related to it by performing a backward traversal from this node to the entry node. Since the state graph is also a graph representation of the program code, slicing via Jipda also needs to locate nodes related to the slicing criterion by traversal on the graph.

4.2.1 Assumption

By observing figure 4.3 and 4.4, we can set the state graph apart from the

PDG by two points. The first one is, a node in a PDG represents one corresponding statement, however, a node in a STG represent identifiers in the state graph. The second one, the edges in a PDG identify the type of relationship between each node. Looking at figure 4.3, edges store semantic information linked with the interpreter in the state graph. A statement is identified by a sequence of edges in the state graph. Based on the characteristics of the state graph we observed in figure 4.3 and 4.4, we make an assumption that by point out an interested line to slice on program, if this line does not contain an identifier then this line isn't data dependent on other code; otherwise, locating the value of identifier from top to this line. In this case, the first step to drive slicing algorithm is calculate the range of interested line to get all corresponding nodes in state graph. Then, from the start of the range, we examine each node to confirm data dependences. In order to illustrate our assumption, we make an observation of instances below.

```
1 var a=1;
2 var b=2;
3 var c=a+1;
4 var d=b-1;
```

The corresponding state graph of this experimental code from Jipda framework is shown in figure 4.6. We know variable *a* directly influences variable *c*; and variable *b* is a direct influence on variable *d*. This code can be divided into two slices. The first slice can be obtained by slicing on criterion < *3, {c}* > is:

```
1 var a=1;
2 var c=a+1;
```

The second slice result from slicing criterion < *4, {d}* > is:

```
1 var b=2;
2 var d=b-1;
```

Checking the first slice on the state graph with regard to our assumption, we locate the range of the 3th statement in the source code between edge 9 to 16 in figure 4.6. While going through this range, we receive identifier reference at edge number 12. Then looking up this identifier from the top to 9, the variable declaration matching with this identifier is between edge 1 to edge 4. Therefore, we can add these nodes to the slice. And return to range of 3th statement to go on querying. When reaching the end of the range, we accomplish program slicing. We can get the second slice by same method of query on state graph.

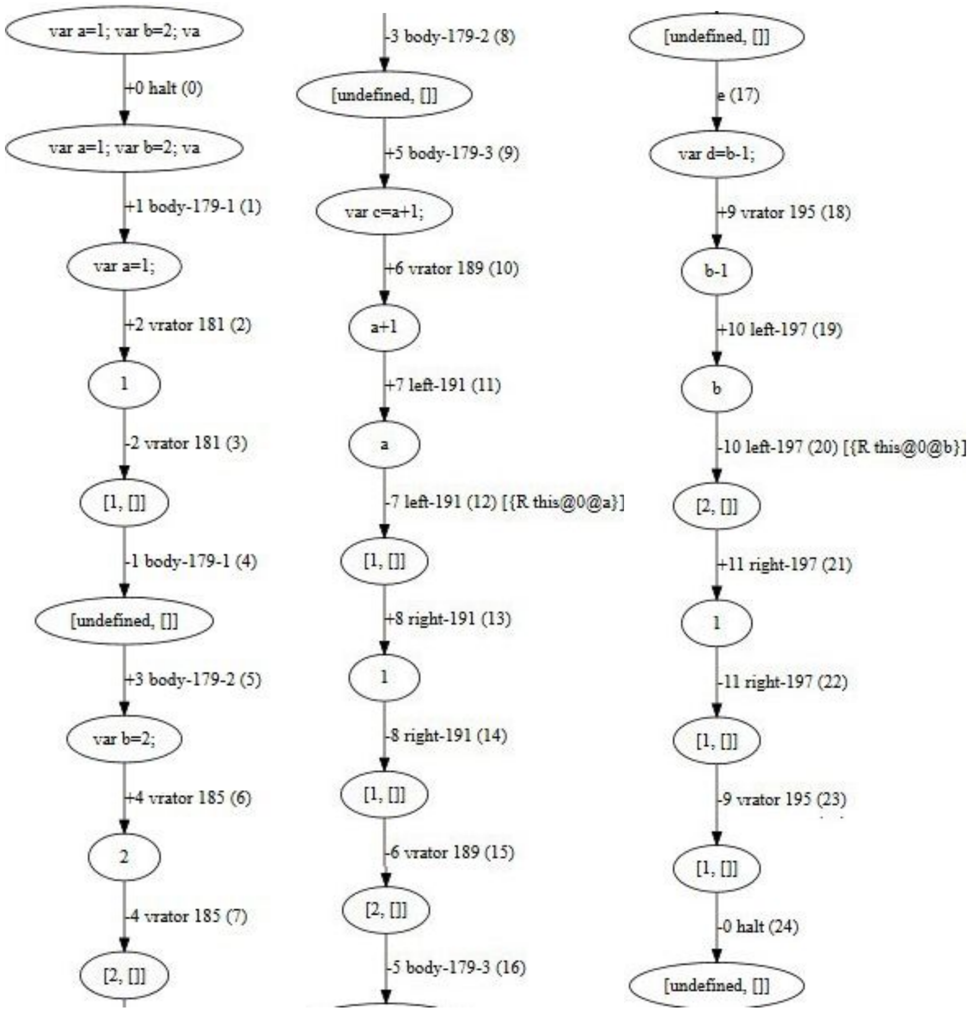


Figure 4.6 State graph of experimental code

We use the same method of observation to find possible solution to get control dependency. The experimental code is in below:

```
1  var a=1;
2  var b=3;
3  var c=2;
4  var add=function(x,y){
5      return x+y;
6      }
7  var sum=add(a,b);
```

In this experimental code, slicing on line 7th invokes the function add with actual parameter pass in. By querying the state graph in figure 4.7, the range of statement 7th is between edge 18 to 35. As the identifier information labeled on edge number 20. We know when interpreter building state graph, function referencing is recognized in the same way as variable referencing, by comparing identifier with function name, we can determine which function is called. Also the actual parameters *a* and *b*, are identified with their declaration statement. On the state graph, we can see the graph listing all states of how a program walk through into function *add* after a statement pass actual parameter in that function. The final state of function returns computation result is assigned to the variable sum. The details of the state transition is not needed for the slice. We only need to involve the part of the function declaration in sliced code. To go on querying to the end edge of this statement, we receive identifier *a* and *b* subsequently, to each identifier, query from top again. The function declaration and the variable binding with actual parameters is directly influenced on slicing criterion.

Over the sample code, we assuming variable *sum* would have effect on some statements in follow-up code, such as an assignment come after the code, like:

```
var d = sum;
```

This time, we have to take into account into statements that are indirectly influenced by the slicing criterion. While querying on the corresponding node to this statement on state graph, we will get an identifier point to declaration of *sum*. After find out the start edge of sum declaration statement, we further query the corresponding end edge of it. During the query in this turn, we will first receive function identifier to search corresponding function declaration. Then we need to search data dependency on identifier *a* and *b*. When we meet the end edge of statement in second query, the process to find relative code which effect on *sum* in sliced line is finished. We can draw conclusions based on the above, the process of dependences collection is iteratively executed to find indirect dependences.

For conditional statements, Jipda is able to analyse state transitions on **if-else** statements. By checking on corresponding state graph of **if-else** statement, we can know each predict expression has a boolean state relies on its computation result. The followed state transits into its clause only when the state's value of predict is true. Our hypothesis to handle conditional statement is: if a statement in the clause of false predicate, then we can say in static analysis, this code is never used. Therefore, we can replace the code in that clause with null value. If line of slicing criterion is involved in clause of **if-else** statement, we say the statement in predicate is directly influence on slicing criterion; otherwise, is indirectly influence.

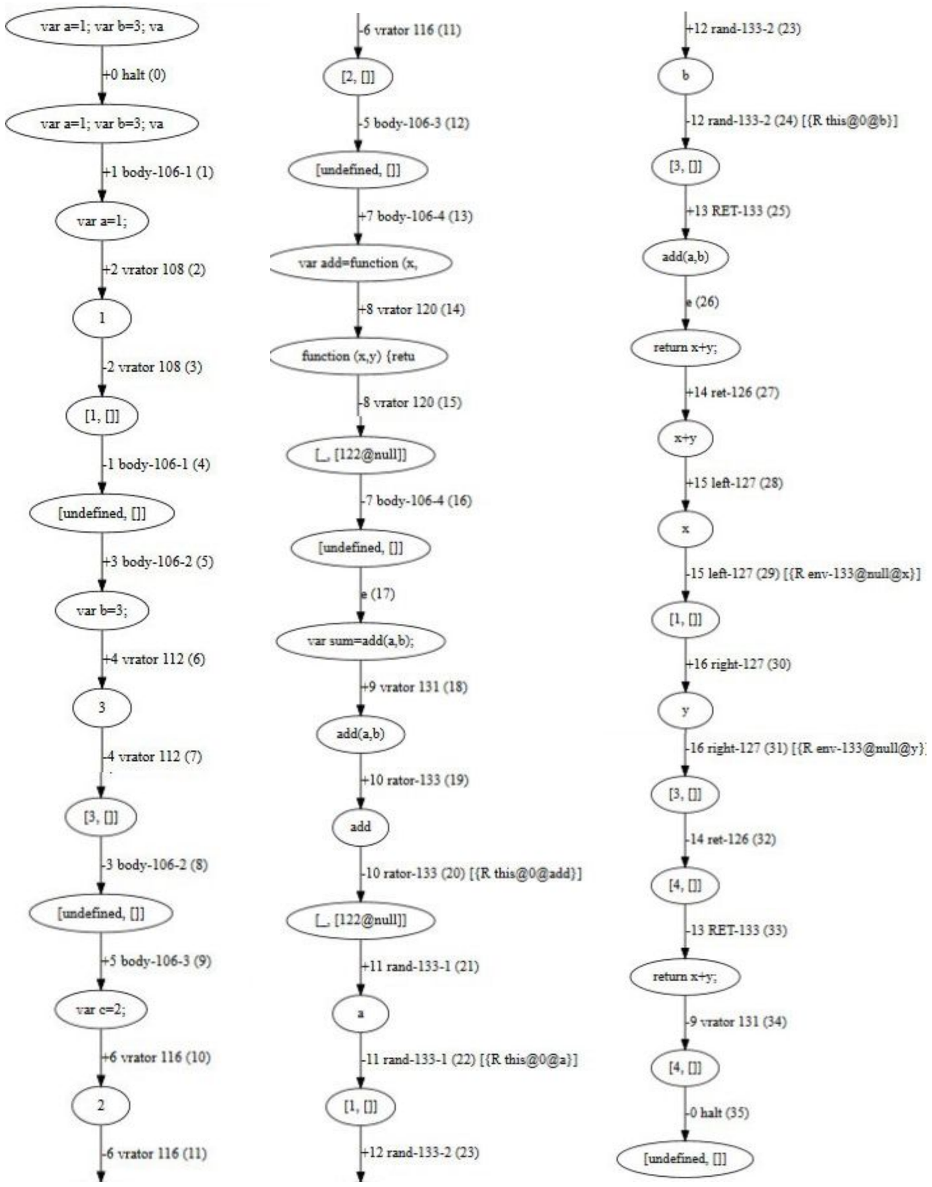


Figure 4.7 Corresponding state graph of experimental code.

4.2.2 Conceive outline of interpreter

With respect to Weiser's theory, our JipdaSlicer requires user input in the form of a line number and a variable name which denotes the slicing criterion or the start condition to drive the slicing algorithm. Based on our assumptions, we need a driving function to detect the range of sliced code. Then put nodes in this range into evaluator in order. The evaluator is used to detect the identifier and to launch the search function. A variable is always declared by using the reserved word *var*, matching a variable identifier on the state graph is always searching its *var* declaration statement by comparing the variable name. For the declaration of functions, JavaScript allows definition in two ways, one is starting with *var*, another is function declaration with function's name. So, there should be two methods of searching a function. The searching method, at last, identifies dependencies by comparing function name with its corresponding declaration. An iterator to get indirect dependences node of slicing criterion. Finally, there is a function to convert the set of sliced nodes into the target, generated code. Since the state graph results in boolean value of conditional statement, we can determine which statement in clause is executed by the boolean value in predicate. Figure 4.8 is a sketch of our conceived outline to design the algorithm. The process modeled in figure will be recursive execution until the all the node in selected line is visited.

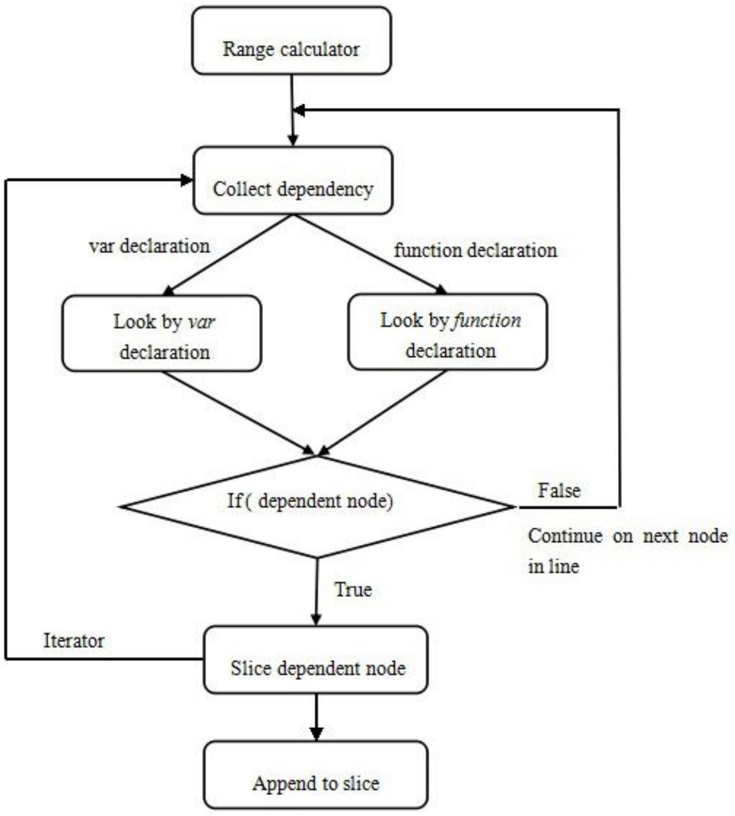


Figure 4.8 Basic idea of design

4.3 Implementation design of JipdaSlicer

We now elaborate on the implementation of JipdaSlicer. JipdaSlicer is an extension of the Stip.js tool. So the implementation is still based on the framework of the original Stip.js. At the beginning of the program slicing, JipdaSlicer collects the nodes and edges in the state graph generated by the original version. As we mentioned above, the first step of slicing is to find the range of statement based on the input line number. The implementation of calculation is quite simple, we use *split()* method of JavaScript to detect line breaks, and use *length* method of array to sum total length before the selected line. This value is the start point, and along with the length of selected line, we get the end point of statement. The computation process is in below code:

```
 var lines = (src + "\n").split('\n');
  var line_range = [0, 0];
  for(var i = 0; i < line_num - 1; i++){
          line_range[0] = line_range[0] + lines[i].length + 1;
}
 line_range[1] =
       line_range[0] + lines[line_num - 1].length - 1;
```

Listing 4.1 Range calculator

Then we pass the edges and nodes in this line to the *collect_dep* function. This function is responsible for collection data or control dependences via searching corresponding identifier. By comparing the value of *tag* on edges we can get corresponding end edge of each *block*. Then the evaluation of current identifier is end. The code start *collect_dep* function as below:

```
 for each node in select line_range[0,1]
 {
    collect_dep(stg_node, program,edges,stg_nodes,slicing_nodes)
 }
```

The definition of *collect_dep* function is in listing 4.2, the meaning of each formal parameter in list 4.2 is: *stg_node* is every single node in selected line, *program* is an array used to keep sliced nodes, *edges* is all the edge on state graph, and the last two are represent for all nodes in state graph. In order to avoid repeat query nodes in selected line, we set *skip* as a span value, to skip the nodes in the same *block* of the identifier which the *collect_dep* function has finished to evaluate. This value can improve efficiency when the selected line includes a function call. As we checked in last section on state graph, the state only go into function's body when that function is called by actual parameter pass in. So all state inside function invocation occurs in selected line. When the algorithm starts to evaluate the first identifier in the line, the iterator will go through all nodes encapsulated in the function call to reach the corresponding end edge. the span between start edge and end edge is the value of *skip*. So when the algorithm returns back to fetch the next node in the selected line, we can step over all nodes in function call *block* based upon *skip* value.

```
function collect_dep(stg_node, program,
edges,stg_nodes,slicing_nodes){
   for(var i = 0; i < edges.length; i++){
      for(var j = i + 1; j < edges.length; j++){
          dep_var is the variable name contains in node;
          base is scope
          dep_node is the dependent node we are searching for
Step 1:
      (1 First look up dep_node by var declaration
      (2 If dep_node is not found in (1, then look up
            dep_node by function declaration

Step 2:
    If (dep_node is found after execute (1 and (2 in
         step 1){
     Put dep_node with necessary information to iterator in order
      to find other code dependent on dep_node.
      }

Step 3: Looking for corresponding end edge.
        Set skip = index_of_end_edge - index_of_start_edge
    }
  }
      return skip;
}
```

List 4.2 Design of function used to collection dependency.

As we discussed above, we need two functions to identify data and control dependencies by comparing identifiers with its scope. The function is used to look up by var declaration is in listing 4.3. If a function is defined starting with the reserved word *var*, we process this declaration as an assignment. The body of the function acts as a variable declaration assigned to a function name. So the principle to find dependent variables or function declarations is the same way, by just comparing the variable name or function name with every node. After the algorithm finds corresponding identifiers, it then checks the length of `declarations` in *VariableDeclarationKont* to get the full definition. Additional, we have to take the scope of each variable into account, so after matching the name of identifier, the algorithm will verify the scope. The formal parameter `var_name` denotes for the name of identifier, and `base` is the scope.

In our assumption, we were planning to query from top to bottom, however, we found the same effect if we query from the current node to top node.

```
function lookup_dep_node_by_var(stg_nodes, start_index, var_name,
      base){
  for(var i = start_index; i > 0; i--){
    if(stg_nodes[i].node){
      var parsenode = stg_nodes[i].node;
      if(parsenode.declarations){
       for(var j = 0; j < parsenode.declarations.length; j++){
         if(parsenode.declarations[j].id.name == var_name){
           if(stg_nodes[i].benva && stg_nodes[i].benva.base ==
               base)
               return stg_nodes[i];
           }
         }
      }
    }
  }
  return false;
```

Listing 4.3 Look up dependence node by *var* declaration

Looking up by *function* declaration is the same principle as *variable* declaration, the only different operation is, we first need to check if the type of node is a *"FunctionDeclaration"* node and then check on function name and its scope. The function implementation is:

```
function lookup_dep_node_by_func_name(stg_nodes, start_index,
      var_name, base){
  for(var i = 0; i < stg_nodes.length; i++){
      if(stg_nodes[i].node){
            var parsenode = stg_nodes[i].node;
            if(parsenode.type=="FunctionDeclaration" &&
            parsenode.id && parsenode.id.name==var_name){
             if(stg_nodes[i].benva && stg_nodes[i].benva.base ==
                  base){
            return stg_nodes[i];
        }
      }
    }
  }
  return false;
}
```

Listing 4.4 Look up dependence node by *function* declaration

The iterator function used to find indirectly dependences is in listing 4.5.

```
function slice_dep_node(dep_node, program, edges, stg_nodes,
slicing_nodes){
      Step 1:
             call collect_dep function to find dependent nodes
      collect_dep(dep_node, program, edges, stg_nodes,
slicing_nodes);

      Step 2: append the current node to sliced array
  }
```

Listing 4.5 Function is responsible for finding iterative dependences.

Above algorithm is a *general pattern* to find slices. This algorithm is able to append statements that have effect on the selected code into slices without reflecting the structure of an **if-else** statement. In order to keep a deterministic process of the predicate, we need a function to read stack information of **if-else** statements in the CESK-machine. Respecting our hypothesis in last section about **if-else** statements, we first set a function to process the boolean result of predicate for each clause. The node label with same *tag*, is in same predicate expression. By identifying the boolean result of this expression, we can decide which branch of **if-else** statement should be replaced by null. The function implementation of pre-process **if-else** statement is in listing 4.6.

```
function slicing_if_statement(stg_nodes, if_node_index){
    var tag = stg_nodes[if_node_index].node.tag;
    if(stg_nodes[i].frame && stg_nodes[i].frame.node
      && stg_nodes[i].frame.node.type == "IfStatement"
       && stg_nodes[i].frame.node.tag == tag){
         if(stg_nodes[i].value.prim.cvalue == true){
           //if the current predicate is true
          //cut off its alternate clause by fill in null;
              stg_nodes[i].node.alternate = null;
         }else{
           stg_nodes[i].node.consequent.type = "BlockStatement";
           stg_nodes[i].node.consequent.body = [];
      }
    }
    }
```

Listing 4.6 Pre-process of **if-else** statement.

The query of dependences in **if-else** statements consists of two parts, one is searching dependences of expression in predicates, another one is searching dependences of each clause after *true* predicates. The way of query is the same to *general pattern*.

4.4 Conclusion

In this chapter we illustrated how the tier splitting tool STIP.js is implemented on top of the Jipda framework. We pointed out the abstract interpreter style embedded in our tier splitting tool and sketched how they work. After parse the source code by Esprima.js, a CESK-machine based interpreter in Jipda framework to get state graph from source code. We use the ides of meta-circular evaluator, in STIP.js to construct a PDG by converting node in Jipda. In the second part, we put forward case studies on the state graph to analyse how to query correct dependences on the state graph to get the slice that corresponds to the slicing criterion. We suppose the query on the state graph is always from the top to bottom by comparing the value of identifiers with concerning of scope information. We put forward three assumptions in this section:

1. Directly influence statement and variable is found by matching identifiers and scope information on the edges.

2. Indirectly influence statement and variable is found by iteratively executing to find dependences on directly influence statement and variable we get in each stage of query.

3. We can replace the code in clause result from false predicates with *null* value.

According to the conclusion we observed from the case study, we elaborate on our implementation of JipdaSlicer in the last part of this chapter. The sketch of JipdaSlicer is shown in figure 4.8, the slicing process starts by finding the range of selected statements. In section 4.3, we list the five key functions of JipdaSlicer. The *collect_dep* function is used to detect identifiers. Then looks for the identifiers via two searching functions: *look_up_node_by_var* and *look_up_node_by_func_name*. *Look_up_node_by_var* is query function for assignment, it can determine data dependencies on variable reference and control dependencies on the function defined by *var* reference. The *look_up_node_by_func_name* is the function that looks up control dependent functions. Function *slice_dep_node* acts like an iterator to query the indirectly dependent and append the visited node to slice array. A pre-process function *slicing_if_statement* is used to replace clause in false consequence of predication by *null* value.

In the next chapter, we test our new slicing algorithm JipdaSlicer through some test code, and compare the result with the original version Stip.js.

5

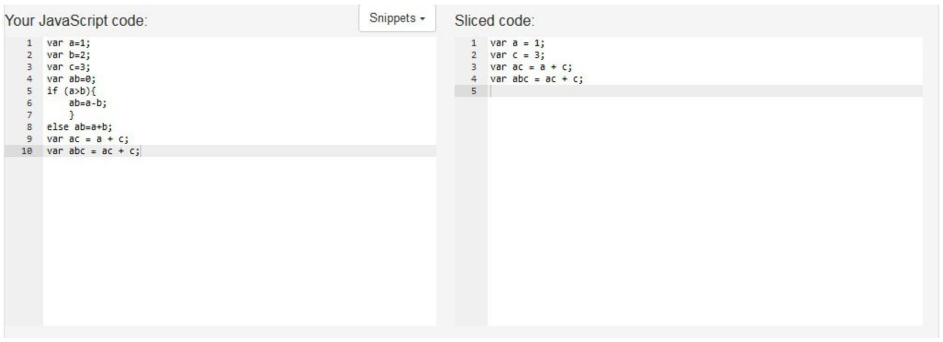
Validation

In the previous chapter, we introduced the tier splitting tool STIP.js and a new slicing algorithm JipdaSlicer. In this chapter we validate JipdaSlicer, by comparing the slices result from STIP.js and it. First we test both of them on the code in figure 2.1(a). From the testing result we can see, two sliced code result from STIP.js and JipdaSliser is conform to Weiser's slice theory. We further carry out three test case to evaluate accuracy of two tools. From the slicing result we can see the interpreter installed in two tools has different feature. When slicing on condition statement, JipdaSlicer returns more pleasant result code. Finally, we give the conclusion about our test result.

5.1 Evaluation

5.1.1 Methodology

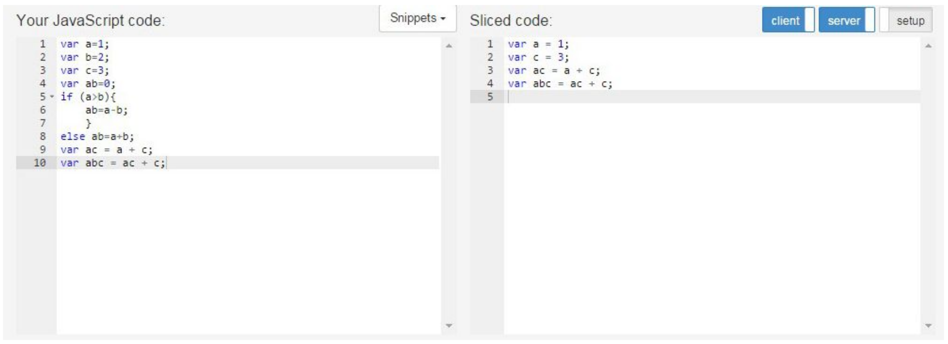
As we introduced in previous chapter, our new slicing algorithm slices program based on Weiser's slicing theory. So below, we first verify if the result we get from JipdaSlicer and Stip.js conform to Weiser's static program slicing theory or not. We use the sample code in figure 2.1(a) as input to verify the slice result. From the figure 5.1 and figure 5.2, we can see the pleasing result we get from STIP.js and JipdaSlicer. Both of them return correctness slice based on Weiser's slicing theory: by selected an interested point, the program slice can be get from deleting the statements have no effect on the selected point. The computation result of sliced code keep consistent with the original code. Hence, in two figures we can see, both two slicing tools delete line no.2 and the statements between line 5 to 8 in code in the left side. The computation result of variable *ac* and *abc* in all code segment is 4 and 7.



```
1  var a=1;
2  var b=2;
3  var c=3;
4  var ab=0;
5  if (a>b){
6      ab=a-b;
7      }
8  else ab=a+b;
9  var ac = a + c;
10 var abc = ac + c;
```

```
1  var a = 1;
2  var c = 3;
3  var ac = a + c;
4  var abc = ac + c;
5
```

Figure 5.1 Slicing result get from Jipdaslicer on slice criterion <10, {abc}>.



```
1  var a=1;
2  var b=2;
3  var c=3;
4  var ab=0;
5  if (a>b){
6      ab=a-b;
7      }
8  else ab=a+b;
9  var ac = a + c;
10 var abc = ac + c;
```

```
1  var a = 1;
2  var c = 3;
3  var ac = a + c;
4  var abc = ac + c;
5
```

Figure 5.2 Slicing result get from STIP.js on slice criterion <10, {abc}>.

In next section, further test is carried out with three test cases. The sample code we set in test case include data dependences, such as assignments, data reference in predicate. The control dependences is checked by function invocation. Also, the design of test case concerns with scope issue and syntax structure of conditional statement.

In order to validate our new slicing algorithm, we comparing the slice result in JipdaSlicer with the slice we get from Stip.js. The validation includes checking the correctness of result and examining the feature of interpreter.

5.2.2 Case testing

In this section, we put forward three test cases in order to we can give an overall verification. For each test case, we first use Weiser's theory to analysis the slicing result of test code, then we show the result we get from two algorithm. The first test case is set for function invocation in order to check if two algorithm can determine correct data dependences and control dependences. The last two are used to validate slicing on **if-else** statement in different case.

Test case 1: slicing on function invoking.

This testing aims to evaluate the correctness of function invocation in different ways. We make sample code to call function in different way in the sake of examining if JipdaSlicer and Stip.js powerful enough to handle control dependences and if the algorithm can bind right actual parameters when slice on source code. Therefore, we set four sub cases to define functions by using reserved word *var* as well as function declarations. As we elaborate our implementation, JipdaSlicer includes two searching function, one is used to look up variable declaration, another on is used to look up function declaration. In order to check if the two searching functions can work well, the first two sub-cases are set for this purpose. So in the first one, we make the code invoke a function define by *var*, and in second one invoke a function defined by function declaration. The last two sub-cases are used to test on more complex situation. The third one tests on inter procedural function invocation, and the last one used to examine if the two algorithm can bound right scope of each identifiers. So in last one, we define a function inside another function, and also set a parameter within function with same variable name in global environment.

Sub-case 1

The first sub-case is shown in figure 5.3. This sample code is used to test function invocation by *var* declaration. This is a simple case to call a function, the actual parameters are in global environment. As we use Weiser's theory analysis on the test code, the slicing result is shown in figure(b).

```
 1 var a=1;
 2 var b=2;
 3 var sum=function(x,y){
 4     var xx=x*x+a;
 5     var yy=y*y+b;
 6     var xy=xx+yy;
 7     return xy;
 8     }
 9 function foo(z){
10     return z+1;
11 }
12 var s=sum(a,b);
```

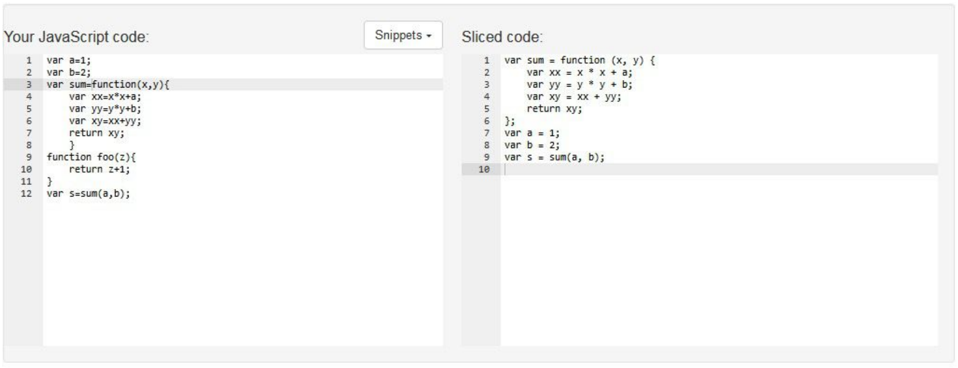
(a)

```
var a=1;
var b=2;
var sum=function(x,y){
    var xx=x*x+a;
    var yy=y*y+b;
    var xy=xx+yy;
    return xy;
    }
~~function foo(z){~~
~~    return z+1;~~
~~}~~
var s=sum(a,b);
```

(b)

Figure 5.3 (a) test code; (b) slicing with criteria<12,{s}>;

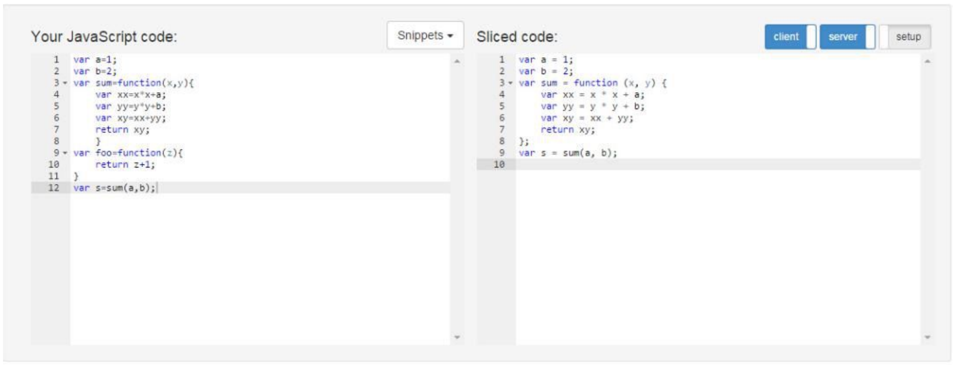
The following pictures are two algorithms resulting in sliced code of above example. In the right part of figures, we can see the correctness of the two algorithms.



```
1   var a=1;
2   var b=2;
3   var sum=function(x,y){
4       var xx=x*x+a;
5       var yy=y*y+b;
6       var xy=xx+yy;
7       return xy;
8       }
9   function foo(z){
10      return z+1;
11  }
12  var s=sum(a,b);
```

```
1   var sum = function (x, y) {
2       var xx = x * x + a;
3       var yy = y * y + b;
4       var xy = xx + yy;
5       return xy;
6   };
7   var a = 1;
8   var b = 2;
9   var s = sum(a, b);
10
```

Figure 5.4(a) Slicing result on <12,{s}> of code in figure 5.3 (a) by JipdaSlicer.



```
1   var a=1;
2   var b=2;
3   var sum=function(x,y){
4       var xx=x*x+a;
5       var yy=y*y+b;
6       var xy=xx+yy;
7       return xy;
8       }
9   var foo=function(z){
10      return z+1;
11  }
12  var s=sum(a,b);
```

```
1   var a = 1;
2   var b = 2;
3   var sum = function (x, y) {
4       var xx = x * x + a;
5       var yy = y * y + b;
6       var xy = xx + yy;
7       return xy;
8   };
9   var s = sum(a, b);
10
```

Figure 5.4(b) Slicing result on <12,{s}> of code in figure 5.3 (a) by STIP.js.

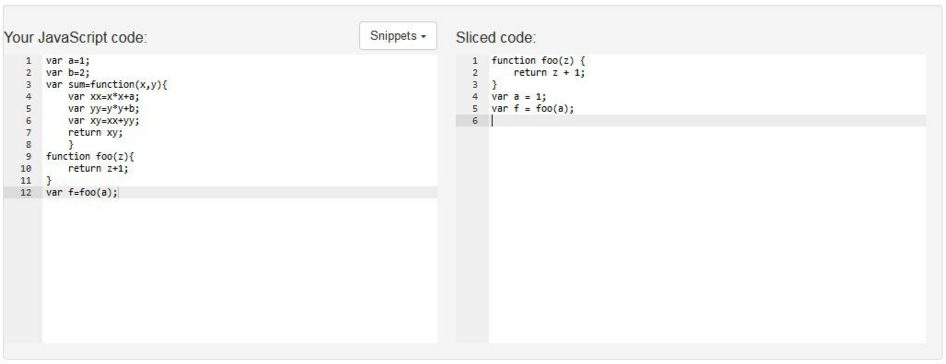
Sub-case 2

The second test case we set in figure 5.5 is used to test on invoking a function declaration. Compare with the test code in figure 5.3, we only change the last line of code. In this time the sliced code should get rid of function declaration of *sum* function. Figure 5.6 shows the slice results. As both results are correct, we can say the two slicing tools is able to find control dependences of a function declaration.

```
 1 var a=1;
 2 var b=2;
 3 var sum=function(x,y){
 4       var xx=x*x+a;
 5       var yy=y*y+b;
 6       var xy=xx+yy;
 7       return xy;
 8       }
 9 function foo(z){
10       return z+1;
11 }
12 var f=foo(a,b);
```
(a)

var a=1;
var b=2;
~~var sum=function(x,y){~~
~~var xx=x*x+a;~~
~~var yy=y*y+b;~~
~~var xy=xx+yy;~~
~~return xy;~~
~~}~~
```
function foo(z){
    return z+1;
}
var f=foo(a,);
```
(b)

Figure 5.5 (a) test code; (b) slicing with criteria<12,{f}>;

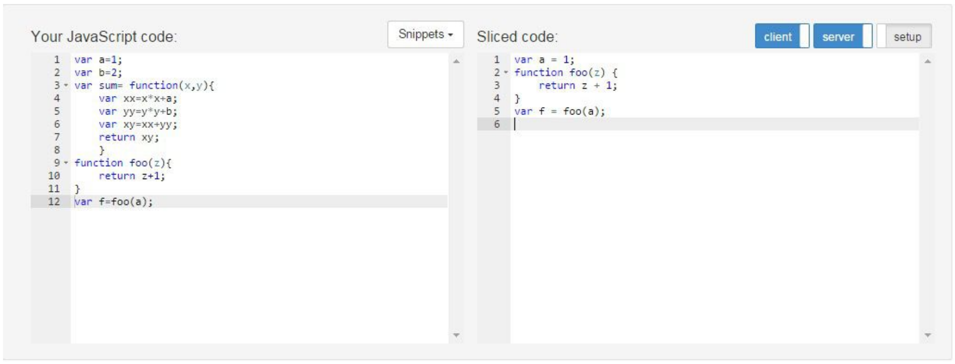

Your JavaScript code:

```
var a=1;
var b=2;
var sum=function(x,y){
    var xx=x*x+a;
    var yy=y*y+b;
    var xy=xx+yy;
    return xy;
    }
function foo(z){
    return z+1;
}
var f=foo(a);
```

Sliced code:

```
function foo(z) {
    return z + 1;
}
var a = 1;
var f = foo(a);
```

Figure 5.6 (a) Slicing result on <12,{f}> of code in figure 5.5 (a) by JipdaSlicer.


Your JavaScript code:

```
var a=1;
var b=2;
var sum= function(x,y){
    var xx=x*x+a;
    var yy=y*y+b;
    var xy=xx+yy;
    return xy;
    }
function foo(z){
    return z+1;
}
var f=foo(a);
```

Sliced code:

```
var a = 1;
function foo(z) {
    return z + 1;
}
var f = foo(a);
```

Figure 5.6 (b) Slicing result on <12,{f}> of code in figure 5.5 (a) by STIP.js.

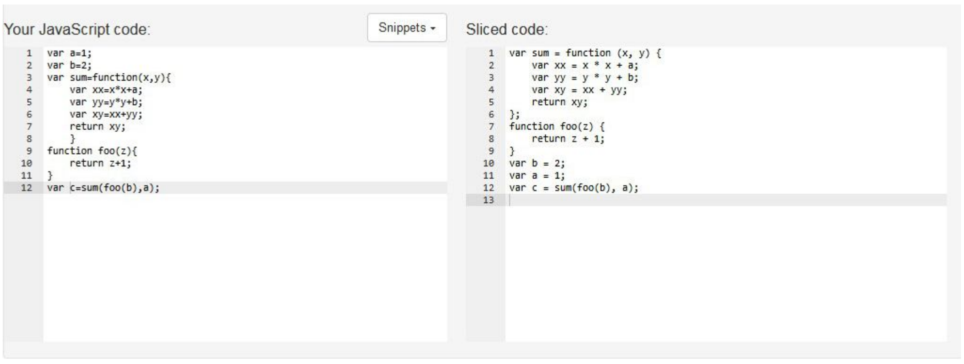
Sub-case 3

The next test case shows slicing with inter procedural call.

```
 1 var a=1;
 2 var b=2;
 3 var sum=function(x,y){
 4     var xx=x*x+a;
 5     var yy=y*y+b;
 6     var xy=xx+yy;
 7     return xy;
 8     }
 9 function foo(z){
10     return z+1;
11 }
12 var c=sum(foo(b),a);
```

Figure 5.7 Test code used to evaluate inter procedural call.

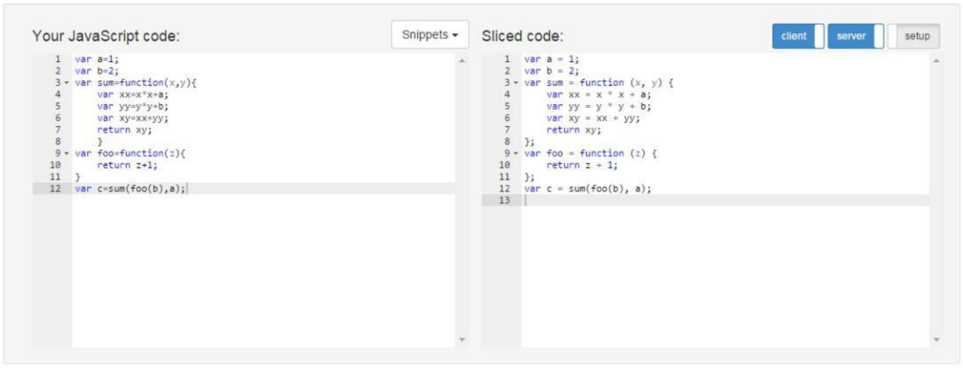
The following pictures show the slicing result on slicing criterion *<12,{c}>*, since in line 12, the expression includes two function invocation, the result code should keep all code in slicing result.



```
var a=1;
var b=2;
var sum=function(x,y){
    var xx=x*x+a;
    var yy=y*y+b;
    var xy=xx+yy;
    return xy;
    }
function foo(z){
    return z+1;
}
var c=sum(foo(b),a);
```

```
var sum = function (x, y) {
    var xx = x * x + a;
    var yy = y * y + b;
    var xy = xx + yy;
    return xy;
};
function foo(z) {
    return z + 1;
}
var b = 2;
var a = 1;
var c = sum(foo(b), a);
```

Figure 5.8 (a) Slicing result on <12,{c}> of code in figure 5.7 by JipdaSlicer.



```
var a=1;
var b=2;
var sum=function(x,y){
    var xx=x*x+a;
    var yy=y*y+b;
    var xy=xx+yy;
    return xy;
    }
var foo=function(z){
    return z+1;
}
var c=sum(foo(b),a);
```

```
var a = 1;
var b = 2;
var sum = function (x, y) {
    var xx = x * x + a;
    var yy = y * y + b;
    var xy = xx + yy;
    return xy;
};
var foo = function (z) {
    return z + 1;
};
var c = sum(foo(b), a);
```

Figure 5.8 (b) Slicing result on <12,{c}> of code in figure 5.7 by STIP.js.

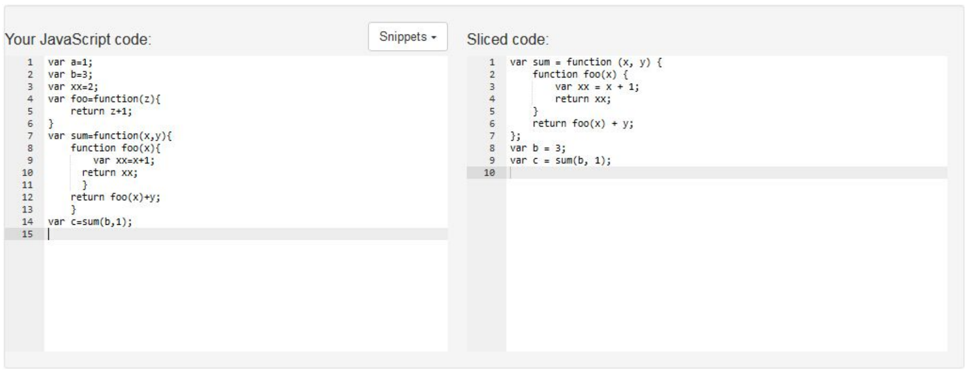
Sub-case 4

The last case of function invocation is by nesting. We set this sub case in order to check if the two algorithms bind with the right scope when analyzing the source code. Also, we test if the algorithm can search right data dependence in the case of duplication of variable name, like we define *xx* in line 3 and line 9. The test code is in figure 5.9. In the test code we define a function *foo* not only in global environment, but also inside the *sum* function. If we slice the code on line 14, only the nested function should involved in sliced code.

```
 1 var a=1;
 2 var b=3;
 3 Var xx=2;
 4 var foo=function(z){
 5     return z+1;
 6 }
 7 var sum=function(x,y){
 8     function foo(x){
 9         var xx=x+1;
10         return xx;
11         }
```

Figure 5.9 Test code.

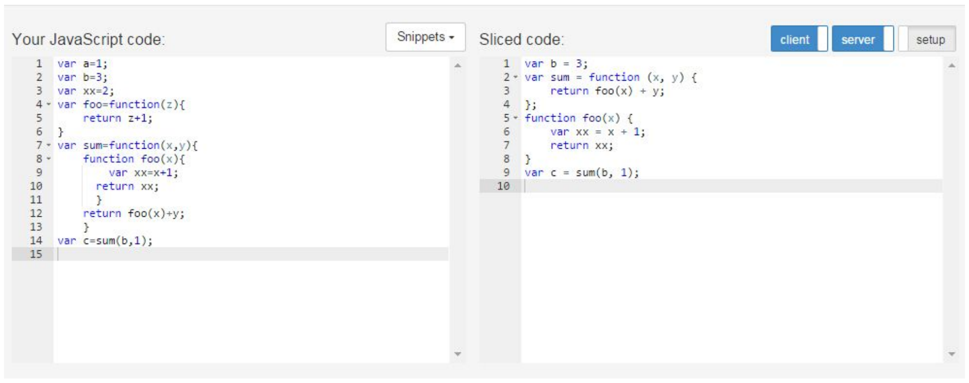
The following figures show us we can get correct sliced code by two algorithm slicing on criterion <14,{c}>. The different is that JipdaSlicer keeps the function nested but STIP.js decomposes the two function at same level. This is because when Stip.js build PDG, it creates an entry node for each function and after traversing all related node, the algorithm prints the function based on each entry node. In JipdaSlicer, after the algorithm find corresponding function declaration, it just fetches the whole body of that function. In the line used to start the slicing, is doesn't contain an identifier point to *foo* function define in the same level with *sum*, so JipdaSlicer didn't check on the function declaration in 4th line.



```
 1  var a=1;
 2  var b=3;
 3  var xx=2;
 4  var foo=function(z){
 5      return z+1;
 6  }
 7  var sum=function(x,y){
 8      function foo(x){
 9          var xx=x+1;
10        return xx;
11        }
12      return foo(x)+y;
13      }
14  var c=sum(b,1);
15
```

```
var sum = function (x, y) {
    function foo(x) {
        var xx = x + 1;
        return xx;
    }
    return foo(x) + y;
};
var b = 3;
var c = sum(b, 1);
```

Figure 5.10(a) Result of slicing on <14,{c}> of the code in figure 5.9 by JipdaSlicer.



```
 1  var a=1;
 2  var b=3;
 3  var xx=2;
 4  var foo=function(z){
 5      return z+1;
 6  }
 7  var sum=function(x,y){
 8      function foo(x){
 9          var xx=x+1;
10        return xx;
11        }
12      return foo(x)+y;
13      }
14  var c=sum(b,1);
15
```

```
var b = 3;
var sum = function (x, y) {
    return foo(x) + y;
};
function foo(x) {
    var xx = x + 1;
    return xx;
}
var c = sum(b, 1);
```

Figure 5.10(b) Result of slicing on <14,{c}> of the code in figure 5.9 by STIP.js.

Looking at the results we get from the two algorithms in above test cases, we can know both of them return us a correct slice. The difference is the order of statements in the sliced code. The different way of querying results in this kind of different ordering. In JipdaSlicer, after abstract interpreter invoked by getting the range of selected line, the algorithm evaluates each node in that line in sequence. If the node is bundled with identifier, the node on which it dependent is always found by traversal on state graph to matching the name of variable with scope. So the order of code in the result from JipdaSlicer depends on the position of identifier in selected line.

Test case 2: slicing on slicing on conditional statement.

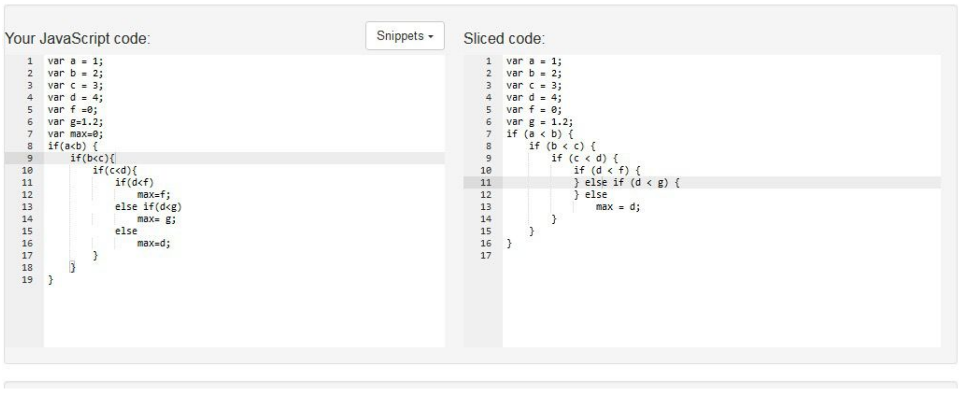
We make this example in order to check grammatical structure of a conditional statement in the resulting code. The test code is given in figure

5.11(a). As shown in figure 5.11, if we want to get the slice with slicing criterion *<16,{max}>*, the result would like in figure 5.11(b). The resulting code gets rid of the false branch of **if-else** statement based on computation result in conditional expression. Based on our design of JipdaSlicer, in the pre-process phase of conditional statement, it will first cuts off branch which isn't accessed when slicing on program. In this example, since the statement in 16th is in **if-else** statement, it should return an empty block of each false branch. Running the result of JipdaSlicer is shown in figure 5.12.

```
1    var a = 1;
2    var b = 2;
3    var c = 3;
4    var d = 4;
5    var f =0;
6    var g=1.2;
7    var max=0;
8    if(a<b) {
9        if(b<c){
10           if(c<d){
11               if(d<f)
12                   max=f;
13               else if(d<g)
                     max= g;
                 else
```

```
var a = 1;
var b = 2;
var c = 3;
var d = 4;
var f =0;
var g=1.2;
var max=0;
if(a<b) {
    if(b<c){
        if(c<d){
            if(d<f)
                ~~max=f;~~
            else if(d<g)
                ~~max= g;~~
            else
```

Figure 5.11 (a) sample code; (b) slicing with criteria<16,{max}>


Your JavaScript code:

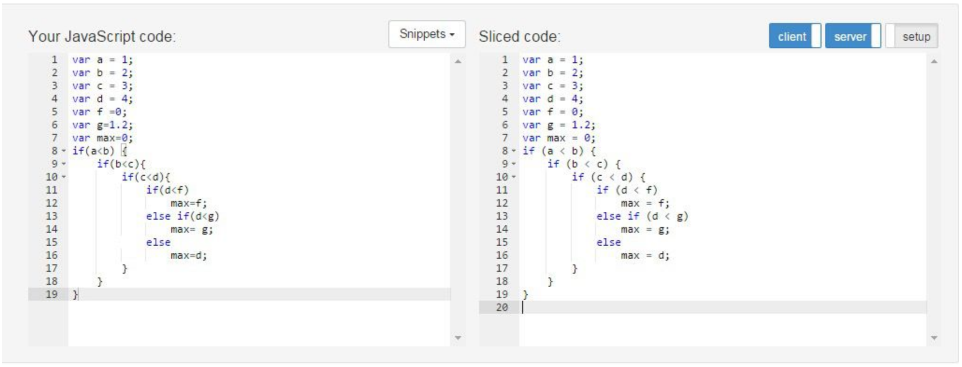
```
1   var a = 1;
2   var b = 2;
3   var c = 3;
4   var d = 4;
5   var f =0;
6   var g=1.2;
7   var max=0;
8   if(a<b) {
9       if(b<c){
10          if(c<d){
11              if(d<f)
12                  max=f;
13              else if(d<g)
14                  max= g;
15              else
16                  max=d;
17          }
18      }
19  }
```

Sliced code:

```
1   var a = 1;
2   var b = 2;
3   var c = 3;
4   var d = 4;
5   var f = 0;
6   var g = 1.2;
7   if (a < b) {
8       if (b < c) {
9           if (c < d) {
10              if (d < f) {
11              } else if (d < g) {
12              } else
13                  max = d;
14          }
15      }
16  }
17
```

Figure 5.12 Slicing result of figure 5.11(b) by JipdaSlicer

The following picture shows the result we get from STIP.js. From the figure we can see STIP.js keeps all source code. In the sliced PDG used to generate this result code, we see that STIP.js marks correct node in PDG, as all the nodes in reach. So STIP.js also traces on right data and control dependences. However, when generating the target code from the PDG, STIP.js keeps all code in order to sustain the **if-else** structure. The different between the two solutions is JipdaSlice use *null* value fill in the clause when it isn't accessed based on prediction.



```
var a = 1;
var b = 2;
var c = 3;
var d = 4;
var f =0;
var g=1.2;
var max=0;
if(a<b) {
    if(b<c){
        if(c<d){
            if(d<f)
                max=f;
            else if(d<g)
                max= g;
            else
                max=d;
        }
    }
}
```

```
var a = 1;
var b = 2;
var c = 3;
var d = 4;
var f = 0;
var g = 1.2;
var max = 0;
if (a < b) {
    if (b < c) {
        if (c < d) {
            if (d < f)
                max = f;
            else if (d < g)
                max = g;
            else
                max = d;
        }
    }
}
```

Figure 5.13 (a) Slicing result of figure 5.11 (b) by STIP.js

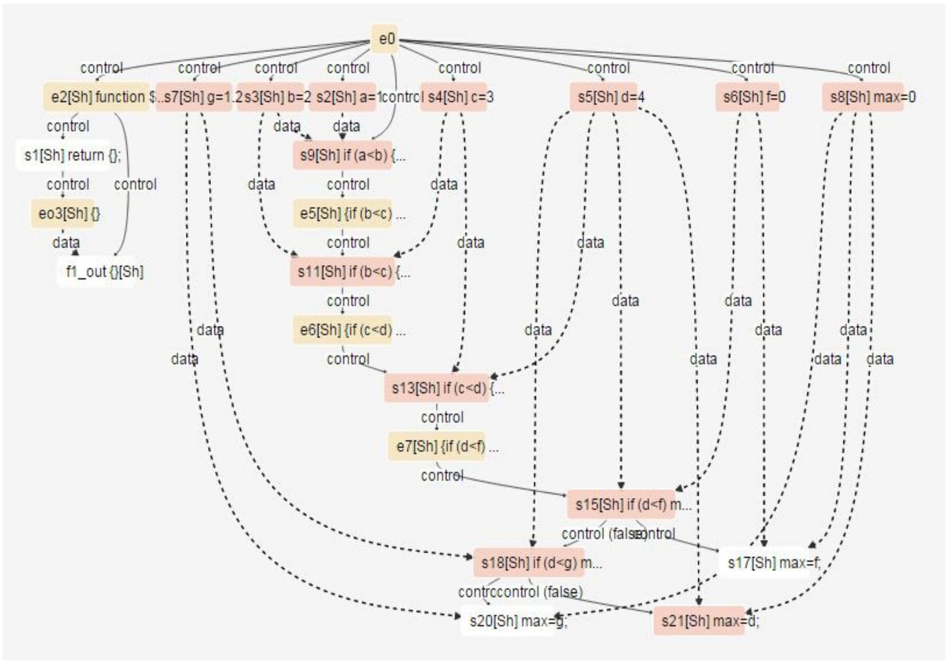


Figure 5.13 (b) PDG generate by STIP.js of the sample code in figure 5.11(a).

Test case 3: slicing on function invoking with conditional statement.

This example shows another situation where the process of slicing involves conditional statements. The difference with test case 2 is the slicing criterion is not in any clause of **if-else** statement. We set this example because JipdaSlicer has a different mechanism in this situation.

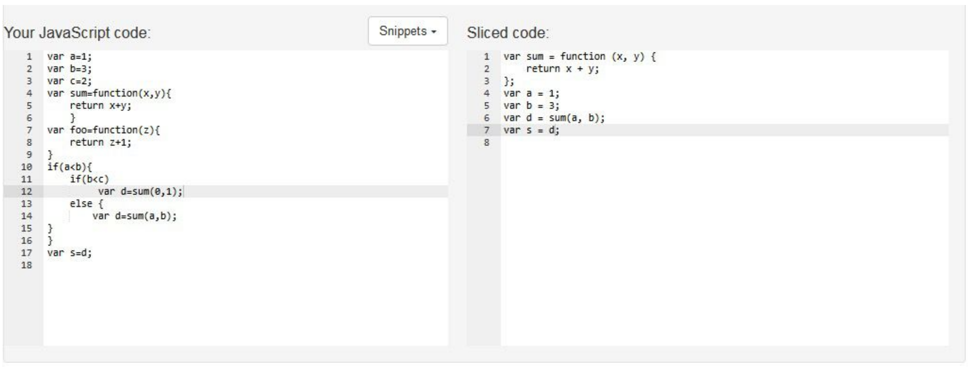
```
1  var a=1;                      var a=1;
2  var b=3;                      var b=3;
3  var c=2;                      var c=2;
4  var sum=function(x,y){        var sum=function(x,y){
5         return x+y;                   return x+y;
6      }                             }
7  var foo=function(z){          ~~var foo=function(z){~~
8         return z+1;            ~~       return z+1;~~
9  }                             ~~}~~
10 if(a<b){                      if(a<b){
11        if(b<c)                       ~~if(b<c)~~
12        var d=sum(0,1);               ~~var d=sum(0,1);~~
13        else {                        else {
14        var d=sum(a,b);               var d=sum(a,b);
15 }                             }
16 }                             }
17 var s=d;                      var s=d;
```

Figure 5.14 (a) Sample code; (b)slice on criteria<17,{s}>.

As analysed in figure 5.14(b), line 17th relies on variable *d* whose value is decided by predicate result of two nested **if-else** statement. The result of code in figure 5.14(b) we get from JipdaSlicer is in figure 5.15 and STIP.js is in figure 5.14(a).

As we can see from the result, JipdaSlicer gets rid of all clauses of the **if-else** statement and only leaves the statement on which the identifier in line 17th is dependent. It's all because we configure the interpreter of JipdaSlicer to only extract relevant statements if the identifiers contained in slice criterion isn't involved in the **if-else** statement. In such a design, we assume each predicate result of a clause in an **if-else** statement is not relevant to the target code, only the clause that really gets executed should be concerned.

We can also see the same issue with test case 2 in the result from STIP.js in figure 5.16(a). By check the corresponding PDG of this testing example, we can find the node represents for statement in line 17 isn't binding with correctness data dependency.



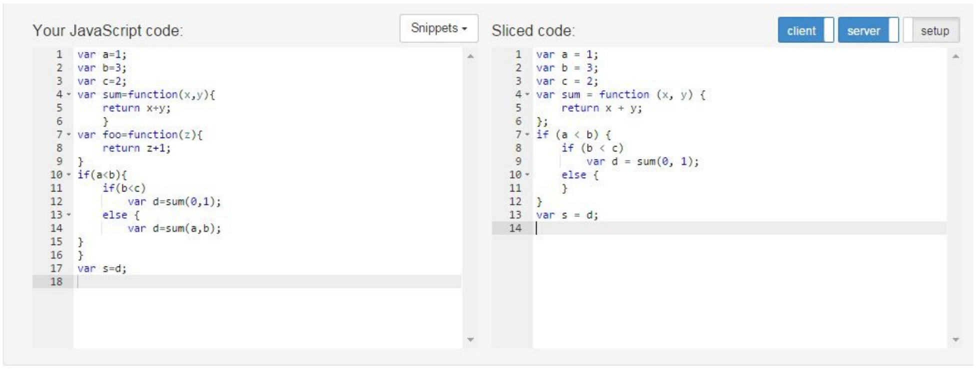
Your JavaScript code:

```
var a=1;
var b=3;
var c=2;
var sum=function(x,y){
    return x+y;
    }
var foo=function(z){
    return z+1;
}
if(a<b){
    if(b<c)
         var d=sum(0,1);
    else {
        var d=sum(a,b);
}
}
var s=d;
```

Sliced code:

```
var sum = function (x, y) {
    return x + y;
};
var a = 1;
var b = 3;
var d = sum(a, b);
var s = d;
```

Figure 5.15 Slicing result of figure 5.14 (b) by JipdaSlicer.



Your JavaScript code:

```
var a=1;
var b=3;
var c=2;
var sum=function(x,y){
    return x+y;
    }
var foo=function(z){
    return z+1;
}
if(a<b){
    if(b<c)
        var d=sum(0,1);
    else {
        var d=sum(a,b);
}
}
var s=d;
```

Sliced code:

```
var a = 1;
var b = 3;
var c = 2;
var sum = function (x, y) {
    return x + y;
};
if (a < b) {
    if (b < c)
        var d = sum(0, 1);
    else {
    }
}
var s = d;
```

Figure 5.16 (a) Slicing result of figure 5.15 (b) by STIP.js.

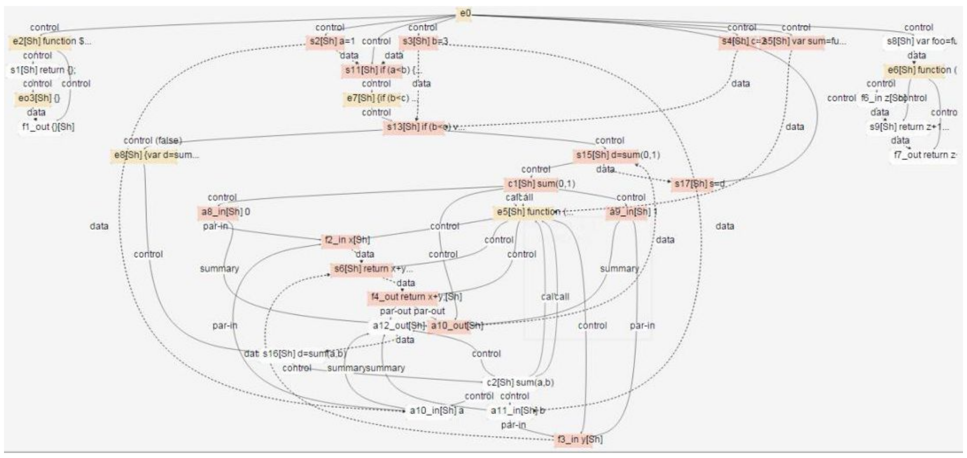


Figure 5.16 (b) Corresponding PDG of code in figure 5.14 (a) by STIP.js.

5.2 Conclusion

First of all, both JipdaSlicer and STIP.js respect Weiser's static slicing theory. From the testing result, we can see the different properties of abstract interpretation in JipdaSlicer and STIP.js. As we discussed in the first test case, the order of sliced code in JipdaSlicer is dependent on the identifiers' position in slicing criterion, whereas, STIP.js can keep the sequence of statements the same the as source code. In the last two cases, we discuss about different mechanisms to deal with clauses in **if-else** statements in JipdaSlicer based on if slicing criterion involved in **if-else** statement or not. If the slicing criterion is involved in the clause of the predicate, we regard for grammatical structure of **if-else** statement is relevant for sliced code. In this case, we configure JipdaSlicer fill *null* value in the clause which isn't executed based on the predicate. The syntax structure of **if-else** statement is still kept in the resulting code so that we can trace clauses based on condition expression. In contrast, we only need to fetch the executed clause in the **if-else** statement if the slicing criterion isn't in any clauses. By comparing the resulting code, we can say JipdaSlicer is better in analysis of **if-else** statements. Both JipdaSlicer and STIP.js are powerful enough to deal with function invocation in any method.

6

Conclusion

Writing a web application is not easy since it requires developers to have knowledge about each tier and its technologies. This complexity can be reduced by tierless programming languages, which enable developers to use a single language, then distribute the tierless code into different logic tiers, such as client and server tiers in web development. Instead of investing in a novel programming language, the aim is to use an existing general-purpose language, JavaScript, to allow tierless programming. To achieve tier splitting in the case of JavaScript, only a minimum of code must be annotated with @client and @server annotations.

STIP.js is a tool built on top of the Jipda framework that generates distributed code by program slicing on the PDG. The development of this slicing tool is mainly based on two technologies: program slicing and abstract interpretation. The Jipda framework results in a state graph by performing a static analysis of source code based on a machine-based abstract interpreter. On top of the state graph, STIP.js constructs a PDG by reconstituting each node and edge through an evaluator. The tierless code is then split by performing a static program slice to generate sliced code for server and client tier.

In this thesis, we develop a *new algorithm* to slice on the state graph instead of the PDG, we call it JipdaSlicer. The slice result of JipdaSlicer can act as prerequisite to accomplish tier splitting. After illustrating our implementation design based on our assumption in section 4.2.1, we set up the evaluation based on three case studies in order to check if JipdaSlicer could compute correct program slices and keep the syntax structure of if-else statements. Finally, we validate Jipdaslicer by comparing the slicing result with STIP.js.

6.1 Contribution

The focus of this dissertation was applying program slicing technology on a state graph directly. Therefore we make an implementation of JipdaSlicer, a tool that slices a state graph directly without constructing a Program Dependency Graph first. We give an overview of contributions in this section.

Improvement

JipdaSlicer breaks with the traditional slicing algorithm that perform a traversal on a Program Dependency Graph. When Stip.js first constructs a Program Dependency Graph, it has to consider the type of node, connections with the successor and predecessor based on data and control dependences or syntax structure. Instead of building a complex tree structure via analysis of each node to present data and control dependencies, JipdaSlicer develops a *general pattern* to deal with dependency matching. This is beneficial from the feature of state graph, all dependent relationships can be identified with identifier information marked on edges. Queries can be executed from top to bottom, as well as, from the slicing node to top. This *general pattern* isolates dependencies query from analysis complexity, such as, syntax structure in conditional statement, context of controls. In order to keep the syntax of conditional statements in sliced code when slicing on statement inside conditional structure, we developed a component in the interpreter to deal with predicates and clauses in conditional statements by analysis stack information in the CESK-machine. By this component, the clause is evaluated if the predicate is *true* and fill *null* in its alternative clause; vice versa.

Correctness

As we can see from evaluation in chapter 5, the slice we got from JipdaSlicer respects Weiser's theory and the computation result is correct. By comparing with STIP.js, JipdaSlicer returns more pleasing result in the case of dealing with **if-else** statements. The result from JipdaSlicer is not only a projection of each consequent of predicates with syntax structure, but also refers to the right scope of variables between block levels.

6.2 Future work

In this section we will discuss possible improvements and extensions for JipdaSlicer.

Interpreter design

Our current version of JipdaSlicer correctly deals with conditional statements. Next step we will extend JipdaSlicer to process more syntax structure, such as loop statement like **while**, **for**. It is not difficult to accomplish such an extension. The general patterns of searching data and control dependences also fit for loop statements. What remains to be done is to design a component in the interpreter to handle the syntax structure of loop statement, similar to what we did for **if-else** statement.

Distributed code

To achieve the goal of tier splitting, we also need to extend JipdaSlicer to deal with distributed node. The next version of JipdaSlicer should be able to generate distributed sliced code in Node.js and Meteor.js framework.

Algorithm performance

As discussed above, slicing on a state graph simplifies the process of determining data and control dependences. However, querying on a line structure is more time-expensive than querying a tree structure. So in further work, we need to develop a dynamic programming algorithm to reduce time complexity of our algorithm.

Reference

[1] M. Weiser. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, Ann Arbor, 1979.

[2] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[3] Tip F. A Survey of Program Slicing Techniques.[J]. Journal of Programming Languages, 1994, 3(93):121--189.

[4] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In Conference Record of the Eighth ACMSymposium on Principles of Programming Languages, pages 207–218, 1981.

[5] "Experience with a data flow datatype" J. Comput. Languages, to be published, 1983.

[6] http://www.tonymarston.net/php-mysql/3-tier-architecture.html

[7] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3):319–349, 1987.

[8] J.M. Barth. A practical interprocedural data flow analysis algorithm. ommunications of the ACM, 21(9):724–736, 1978.

[9] Ottenstein K, L. Ottenstein: "The program dependence graph in a software development environment[J]. Acm Sifsoft, 1984.

[10] Wang L M, Xian Y, Zhang L, et al. JSSlicer: A Static Program Slicing Tool for JavaScript[J]. Applied Mechanics & Materials, 2013, 241-244:2690-2695.

[11] Rodrigues N F, Barbosa L S. Component Identification Through Program Slicing[J]. Electronic Notes in Theoretical Computer Science, 2006:291–304.

[12] Weiser M. Programmers Use Slices When Debugging.[J]. Communications of the Acm, 1982, 25(7):446-452.

[13] Mohapatra D P, Mall R, Kumar R. An Overview of Slicing Techniques for Object-Oriented Programs[C]// INFORMATICA. 2006.

[14] Zhao J. Applying Slicing Technique to Software Architectures[C]// In Proc. of 4th IEEE International Conferencei on Engineering of Complex Computer Systems. 1998:87--98.

[15] Meijer, Henricus Johannes Maria, Manolescu, Dragos. Tier splitting for occasionally connected distributed applications: WO, WO2011100171 A2[P]. 2011.

[16] M. Weiser. Private communication, 1994.

[17] J.-F. Bergeretti and B.A Carre. Information-flow and data-flow analysis of while-programs. ACM Transactions on Programming Language and Systems, 7(1):37-61, 1985.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems, 12(1):26–61, 1990.

[19] Liang D, Harrold M J. Slicing objects using system dependence graphs[C]// Software Maintenance, 1998. Proceedings., International Conference on. IEEE, 1998:358-367.

[20] Larsen L, Harrold M J. Slicing object-oriented software[C]// icse. IEEE Computer Society, 1996:495.

[21] Korel B, Laski J. Dynamic program slicing[J]. Information Processing Letters, 1988, 29(88):155–163.

[22] Lakhotia A. Improved Interprocedural Slicing Algorithm[J]. Report Cacs Tr, 1992.

[23] Ball T, Horwitz S. Slicing Programs with Arbitrary Control Flow[C]// 1ST CONFERENCE ON AUTOMATED ALGORITHMIC DEBUGGING. 1993:206--222.

[24] G. Canfora, et al. Conditioned program slicing. Information and Software Technology, 40(11-12):595~607 1988.

[25] Choi J D, Ferrante J. Static Slicing in the Presence of GOTO Statements[J]. Acm Transactions on Programming Languages & Systems, 1994, 16(4):1097--1113.

[26] Korel B, Laski J. Dynamic program slicing[C]. Information Processing Letters, 1988, 29(88):155–163.

[27] T. Reps and W. Yang. The semantics of program slicing and program integration. In Proceedings of the Colloquium on Current Issues in Programming Languages, volume 352 of Lecture Notes in Computer Science, pages 60–74. Springer Verlag, 1989.

[28] Agrawal H. On Slicing Programs with Jump Statements[J]. Acm Sigplan Notices, 1994, 29(6):302-312.

[29] Landi W, Ryder B G. A safe approximate algorithm for interprocedural pointer aliasing[J]. Acm Sigplan Notices, 2004, 39(7):473-489.

[30] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In Proceedings of the Conference on Software Maintenance, pages 299–308, 1992.

[31] Cheng J. Slicing Concurrent Programs - A Graph-Theoretical Approach[C]// Proceedings of the First International Workshop on Automated and Algorithmic Debugging. Springer-Verlag, 1993:223--240.

[32] Agrawal H, Horgan J R. Dynamic Program Slicing[J]. Sigplan Notices, 1990, 25(6):246-256.

[33] Weihl W E. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables.[J]. POPL '80 Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1980.

[34] J.R. Lyle. Evaluating Variations on Program Slicing for Debugging. PhD thesis, University of Maryland, 1984.

[35] R. Compilers : principles, techniques, and tools[J]. Compilers Principles Techniques & Tools, 1988.

[36] ALLEN, F. E. Control flow analysis. In Proceedings of a Symposium on Compiler Optimization. SZGPLAN Not. 5, 7 (July 1970), 1-19.

[37] Ottenstein K J. Data-Flow Graphs as an Intermediate Program Form[J]. Purdue University, 1978.

[38] Ottenstein K J. And Ottenstein L M.An intermediate program form based on a cyclic data-dependence graph. CS-TR 81-1,Dept. Of Computer Science, Michgan Technological Univ., Houghton, MI, Oct. 1981; July 1982, errata.

[39] E. Duesterwald, R. Gupta, and M.L. Soffa. Rigorous data flow testing through output influences. In proceedings of the Second Irvine Software Symposium ISS'92, pages 131–145, California, 1992.

[40] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In Proceedings of the Conference on Software Maintenance, pages 299–308, 1992.

[41] Binkley D. Using Semantic Differencing to Reduce the Cost of Regression Testing[C]// In Proceedings of the Conference on Software Maintenance1992:41--50.

[42] Horwitz S. Identifying the Semantic and Textual Differences Between Two Versions of a Program[J]. Proceedings of the Acm Sigplan Conference on Programming Language Design & Implementation, 1990:234--245.

[43] Gallagher K B, Lyle J R. Using Program Slicing in Software Maintenance[C]// IEEE Transactions on Software Engineering1991:751--761.

[44] OO Ekabua， BE Isong. On Choosing Program Refactoring and Slicing Re-engineering Practice Towards Software Quality.

[45] Weiser M. Reconstructing Sequential Behavior from Parallel Behavior Projections.[J]. Information Processing Letters, 1983, 17:129-135.

[46] Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints[C]// Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languagesACM, 1977:238-252.

[47] http://esprima.org/index.html

[48] https://github.com/estree/estree/blob/master/spec.md

[49] https://en.wikipedia.org/wiki/Interpreter_(computing)#Self-interpreter

[50] https://en.wikipedia.org/wiki/Syntax_(programming_languages)

[51] Scott D. The lattice of flow diagrams[M]// Symposium on Semantics of Algorithmic LanguagesSpringer Berlin Heidelberg, 1971:311-366.

[52] Floyd R W. Assigning meanings to programs[J]. Studies in Cognitive Systems, 1993, 14:19-32.

[53] http://matt.might.net/articles/cesk-machines/

[54] https://en.wikipedia.org/wiki/Meta-circular_evaluator

[55] Clements J, Felleisen M. A tail-recursive machine with stack inspection[J]. Acm Transactions on Programming Languages & Systems, 2004, 26(6):1029-1052.

[56] Abelson H, Sussman G, Sussman J. Structure and interpretation of computer programs[M]// MIT Press , McGraw-Hill Companies, 1996:73 - 80.

[57] http://santos.cis.ksu.edu/schmidt/Escuela03/WSSA/talk1p.pdf