國立東華大學資訊工程學系

碩士論文

靜態 program slicing 的語義分析方法

A semantic method of static program slicing

研究生：蔣明豪

指導教授：雍　忠 博士

中華民國九十二年七月

A semantic method of static program slicing

by

Ming-Hau Chiang

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science and Information Engineering

National Dong Hwa University

July 2003

Approved: ______________________

Chung Yung

To my parents, Ching-Huei Chiang and Li-Hua Lin

Acknowledgements

As times goes on, two years is gone. Because many people support me, I can successfully complete the thesis. They provide not only help in research but also encouragements in my life. First of all, I heartfeltly thank my advisor, Professor Chung Yung. Dr. Yung provide motivation and helpful ideas during the many discussions we had. I am grateful to Professor Wuu Yang for his invaluable comments on the early draft of this thesis. I am also grateful to the thesis committee member, Dr. Tyng-Ruey Chuang and Dr. Shr-Jing Yen, for their valuable suggestions on the thesis.

Then, I would like to thank our laboratory members, they give me a hand during the development of the thesis. They are: Jih-Chin Yeh, Shin-Lin Chen, Per-Jen Chuang, Hsiang-Wei Kong, Li-Ban Chen, Ming-Sian Lin, Yun-Lung Yue, Feng-Wei Yang and You-liang Sun.

A semantic method of static program slicing

Ming-Hau Chiang

Advisor: Chung Yung

Abstract

Program slicing is a technique of finding the statements that affect the specific variables in a specific point of a program. It can help programmer understand complicated code. The original concept of program slicing may trace back to the notation introduced by Weiser [WM79, WM81, WM84]. They define program slicing as the solution to data-flow problem using a control flow graph as an intermediate representation. Then Ottenstein K. and Ottenstein L. provide a method based on dependence graph [OO84, RY88].

Instead of using dependence graph, we propose a semantic method of program slicing using abstract interpretation in the thesis. Our approach analyzes a program based on the define-use attributes and the indirectly relevant attributes of the values in the program. We develop augmented semantic functions for computing these attributes and extracting program slices.

Contents

List of Figures

Chapter 1

Introduction

Program slicing is a method of gathering a set of statements that influence the value of a variable V in a statement S. *Program slice* makes up of the parts of program that affect the value of a variable V in a statement S[BG96, Tip95]. Typically, we will give a *slicing criterion* $\mathcal{C}$ for computing program slice and we call the result a program slice with respect to criterion $\mathcal{C}$. Program slicing is roughly divided into static slicing and dynamic slicing. The difference between a static and a dynamic slicing is that former analysis is computed without consider a program's input, and the latter relies on some specific case.

Since then, various methods of computing program slice are proposed. The concept of program slicing may trace back to the notion introduced by Weiser [WM79, WM81, WM84]. They provides bottom-up algorithms based on dataflow equation. They gather the dataflow informations on *control flow graph*. And they use these informations to extract program slice. Then Ottenstein K. and Ottenstein L. provide a method based on *dependence graph*

[OO84, RY88]. They create a *Program dependence graph*(PDG) for using dependence graph based approaches. Then they define slicing as a reachability problem in a dependence graph representation of program.

Program slicing is an approach of collecting the statements that affect specific variables at a specific point. Generally, we call the description of the specific variables at specific point slicing goal. Weiser define program slicing as the solution to data-flow problem using a control flow graph(CFG) as an intermediate representation [WM79, WM81, WM84]. They gather four kinds of dataflow information on CFG. The *definition set* is a set of variables that are defined at a node of CFG. The *use set* is a set of variables that are used at a node of CFG. The *control set* is a set of nodes that control execution of self at a node of CFG. The *relevant set* is a set of variables that affect the variables described in the slicing goal at node of CFG. They propose algorithms to compute program slice with respect to slicing goal according to these information. They say that program slice is a program deleted zero or more statements from source program.

Another method of program slicing is based on dependence graph[OO84, RY88]. They regard program slicing as a reachability problem in a dependence graph representation of program. For computing program slice based on dependence graph, we need a Program dependence graph. PDG consists of nodes that are either assignment or predicate statement and two kinds edges of control dependence edge and flow dependence edge. The control dependence edge is labeled either true or false. The source of control dependence edge is always the entry vertex or a predicate vertex. The control dependence edge from the vertex v_1 to the vertex v_2, denoted by $v_1 \longrightarrow_c v_2$, mean that the execution of the vertex predicate on if the evaluation of the vertex equal to label of the control dependence edge. The data dependence edge from the vertex v_1 to the vertex v_2 mean that result of program computation is changed if the relative order of v_1 and v_2 is reversed. The slicing goal is a node v of PDG. The program slice with respect to slicing goal is all nodes that can be reached from node v.

1.1 Motivation

Program slicing is an approach of finding the statements that affect the values computed at some point interesting us, referred to as a slicing criterion. It can help programmer understand complicated code. The concept of program slicing may trace back to the notion introduced by Weiser [WM79, WM81, WM84]. They define program slicing as the solution to data-flow problem using a control flow graph as an intermediate representation. First, they need a control flow graph representation of program. And they compute $\mathcal{D}(i)$, $\mathcal{U}(i)$ and $\mathcal{R}(i)$ on the control flow graph. $\mathcal{D}(i)$ is a set of the variables defined at node i of CFG. $\mathcal{U}(i)$ is a set of variables which referred at node i of CFG. $\mathcal{R}(i)$ consist of the variables that maybe affect v at node i of CFG for a slicing criterion $\langle n, v\rangle$. Then they compute program slice with respect to slicing criterion according to $\mathcal{R}(i)$. We give a example of program slicing in figure 1.1. Figure 1.2 is a control flow graph of the program. Figure 1.3 is the result

Source Program
read(a);
read(b);
if a < b then
min := a;
max := b;
else
min := b;
max := a;
end if
write(min);
write(max);

Figure 1.1: The example program

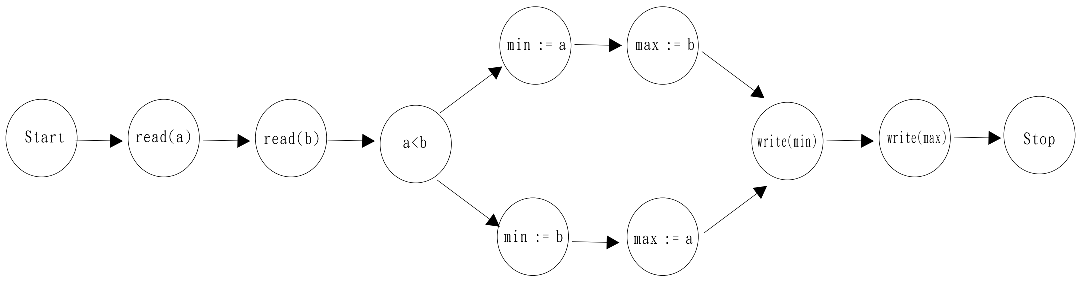

Figure 1.2: The control flow graph of source program

after analysing and the sliced program.

Then Ottenstein K. and Ottenstein L. provide a method based on dependence graph [OO84, RY88]. They construct Program dependence graph. The nodes of PDG are either assignment statement or control predicate statement. PDG has two kind edges of control dependence edge and flow dependence edge. And they define slicing as a reachability problem in a program dependence graph representation of program. Slicing criterion of PDG based slicing method is a node v of PDG. The slice with respect to v consist of all vertices

$Node$	statement	$\mathcal{D}(i)$	$\mathcal{U}(i)$	$Controlset$	$\mathcal{R}(i)$
1	read(a)	{ a }			
2	read(b)	{ b }			{ a }
3	if a < b then		{ a,b }		{ a,b }
4	min := a	{ min }	{ a }	3	{ b }
5	max := b	{ max }	{ b }	3	{ b }
6	else				
7	min := b	{ min }	{ b }	6	{ a }
8	max := a	{ max }	{ a }	6	{ a }
	end if				
9	write(min)		{ min }		{ max }
10	write(max)		{ max }		{ max }

(a) The source program

$Node$	statement
1	read(a)
2	read(b)
3	if a < b then
5	max := b
6	else
8	max := a
	end if
10	write(max)

(b) The program slice with a slicing criterion $\langle 10, \{max\}\rangle$

Figure 1.3: An example of sliced program

from v is reachable.

The above methods both compute program slice with graph representation of program. And the important information for program slicing are definition set, used set and relevant set. The graph representation of program help us to gather these information. We use semantic function and time stamp to help us instead of the graph representation of program.

Although some researches about program slicing are proposed, there have few researches with regard to static slicing of functional language. Instead of using dependence graph, we prose a semantic method of program slicing of first-order functional language using abstract interpretation. We define some semantic function to compute definition, use and relevant set for each expression. We can compute the program slices without graph representation of the program. We can get sliced program according to these information.

1.2 Organization of the thesis

The rest of this thesis is organized as follow:

- There are background knowledge and the description of our source program in Chapter 2.
- We will show our approach and define the semantic functions of computing program slicing in Chapter 3.
- We will show the implement of our approach and discussion on our approach.

- Chapter 5 provides an overview of the related work about program slicing.
- We will provide conclusions and future work in Chapter 6.

Chapter 2

Background

In the chapter, we introduce the fundamental concepts for the techniques developed in the program slicing analysis. Denotational semantics is a technique that maps the phrases of a program into program meaning. We will particularly explain about it next. Abstract Interpretation is a semantics-based program analysis method. The main idea is to construct two different meanings of a program language which are standard meaning and abstract meaning. We will show description about it in section 2.2. Our source language is PLUS which is a first-order functional language. PLUS is Programming Language Using Sets. We will show properties of it in section 2.3. Last, we introduce time stamp. It is a technique that labels expressions in a program with lexicographically ordered time stamps.

2.1 Denotational Semantics

Denotational semantic is based on the recognition that programs and the objects they manipulate are symbolic realization of abstract mathematical objects, for example, string of digits realize numbers. A denotational semantic is in three parts:

1. The syntactic world
   (a) *Syntactic domains* specify collections of syntactic objects that may occur in phrases in the definition of the syntax of the language.
   (b) *Abstract production rules* describe the ways that objects from the syntactic domains may be combined in accordance with the definition of the language.
2. The semantic world
   Semantic domains are sets of mathematical objects of a particular form.
3. The connection between syntax and semantics
   (a) *Semantic function* map objects of the syntactic domain into object in the semantic domain.
   (b) *Semantic equations* to specify how the functions act on each pattern in the syntactic definition of the language.

Figure 2.1 is an example of denotational semantic. It is an addition of integers. We show how to compute 13+23 as follows.

Abstract Syntax

$$\begin{array}{lcll} c & \in & Con & \text{constants} \\ e & \in & Exp & \text{expressions,where} \\ & & & e = c \mid e_1 + e_2 \end{array}$$

Semantic Domains

Int	the standard domain of integers

Semantic Function

$$\varepsilon \quad : \quad Int \longrightarrow Int$$

$$\begin{array}{rcl} \varepsilon[\![c]\!] & = & c\text{,constant} \\ \varepsilon[\![e_1 + e_2]\!] & = & \varepsilon[\![e_1]\!] + \varepsilon[\![e_2]\!] \end{array}$$

Figure 2.1: An example of denotational semantics

$$\begin{aligned} \varepsilon[\![13 + 23]\!] &= \varepsilon[\![13]\!] + \varepsilon[\![23]\!] \\ &= 13 + 23 \\ &= 36 \end{aligned}$$

2.2 Abstract Interpretation

Abstract interpretation is a semantics-based program analysis method. The semantics of a programming language can be specified as a mapping of program to mathematical objects that describes the input-output function for the program. In an abstract interpretation the program is given a mathematical meaning in the same way as a normal semantics. This however is not necessarily the standard meaning, but it can be used to extract information about the computational behaviour of the program.

The chief concept of abstract interpretation is to construct two different meanings of a program language. The first gives the standard meanings of programs in the language. The other can be used to answer certain questions about the runtime behaviour of programs in the language.

2.3 First-Order Language

Programming Language Using Sets(PLUS)[CY97, CY98, CY99, CY03] is a functional programming language. PULS is strongly typed and statically scoped. PLUS has a polymorphic typed system. The following notes are some of highlights of the PLUS programming language:

- PLUS is an *applicative* programming language. The principal control mechanism in PLUS is recursive function application.
- PLUS is *strongly typed.* Every legal expression has a type which is determined automatically by the compiler. Strong typing guarantees that no program con incur a type error at run time.
- PLUS has a *polymorphic* type system. Each of the legal PLUS phrase has uniquely-determined most general typing that determines the set of contexts in which that phrase may be legally used.
- PLUS is statically scoped. PLUS resolves identifier references at compile time, leading to more modular and more efficient programs.
- PLUS include syntax for list constructors and set constructors.

A PLUS consists of a sequence of declarations; the execution of each declaration modifies the environment. Inthe execution of a declaration there are

Abstract Syntax

c	$\in$	Con	consrant
x	$\in$	Bv	bound variables
op	$\in$	Pf	primitive functions
			$op = + \mid - \mid * \mid / \mid \% \mid > \mid >= \mid < \mid <= \mid == \mid ! = \mid \&\& \mid \|\|$
f	$\in$	Fv	function variables
e	$\in$	Exp	expression
			$e = c \mid x \mid !e \mid e_1 op e_2 \mid$ if e_0 then e_1 else $e_2 \mid x = e \mid$ let e_o in $e_1 \mid$
			$f_i(e_1, \ldots, e_n) \mid$ fun $f_i(e_1, \ldots, e_n) = e_j \mid (e_1, e_2) \mid [x : e_1 \mid e_2] \mid \{x : e_1 \mid e_2\} \mid$
			fst $e \mid$ snd $e \mid e_1 :: e_2 \mid$ hd $e \mid$ tl $e \mid e_1$ with $e_2 \mid$ rep $e \mid e_1$ in e_2
pr	in	Prog	program, where $pr = \{e_1; \ldots; e_k;\}$

Semantic Domains

Int	the standard flat domain of integers
Products = Int $\times$ Int	the domain of pair value
Set = 2^{Int}	the domain of set value
List = 2^{Int}	the domain of list value
Bas = Int+Pair+Set+List	the domain of basic value
Fun = $Bas^n \rightarrow Bas$	the domain of first-order functions
D = Bas+Fun + $\{ \perp \}$	the domain of denoteable values
Bve = Bv $\rightarrow$ D	the domain of bound variable environments
Fve = Fv $\rightarrow$ Fun	the domain of function environments

Semantic Function

$$\begin{aligned}
\varepsilon &: Exp \rightarrow Bve \rightarrow Fve \rightarrow D \\
P &: Pf \rightarrow Fun \\
\varepsilon_f &: Exp \rightarrow Fve
\end{aligned}$$

$$\begin{aligned}
\varepsilon[\![\theta\text{: } c]\!]\ bve\ fve &= c,\text{intreger } c \\
\varepsilon[\![\theta\text{: } x]\!]\ bve\ fve &= bve(x) \\
\varepsilon[\![\theta\text{: } (e_1, e_2)]\!]\ bve\ fve &= (\varepsilon[\![\theta_1\text{: } e_1]\!]\ bve\ fve,\ \varepsilon[\![\theta_2\text{: } e_2]\!]\ bve\ fve) \\
\varepsilon[\![\theta\text{: } !e]\!]\ bve\ fve &= P[\![!]\!](\varepsilon[\![\theta\text{: } e]\!]\ bve\ fve) \\
\varepsilon[\![\theta\text{: } e_1 \text{ op } e_2]\!]\ bve\ fve &= P[\![op]\!](\varepsilon[\![\theta_1\text{: } e_1]\!]\ bve\ fve, \varepsilon[\![\theta_2\text{: } e_2]\!]\ bve\ fve) \\
\varepsilon[\![\theta\text{: } x = e]\!]\ bve\ fve &= bve[(\varepsilon[\![\theta\text{: } e]\!]\ bve\ fve)/x] \\
\varepsilon[\![\theta\text{: fst } e]\!]\ bve\ fve &= \varepsilon[\![\theta\text{: } e_1]\!]\ bve\ fve \\
&\quad \text{where } e = (e_1, e_2) \\
\varepsilon[\![\theta\text{: snd } e]\!]\ bve\ fve &= \varepsilon[\![e_2]\!]\ bve\ fve \\
&\quad \text{where } e = (e_1, e_2) \\
\varepsilon[\![\theta\text{: } e_1 :: e_2]\!]\ bve\ fve &= (\varepsilon[\![\theta_1\text{: } e_1]\!]\ bve\ fve) :: (\varepsilon[\![\theta_2\text{: } e_2]\!]\ bve\ fve) \\
\varepsilon[\![\theta\text{: hd } e]\!]\ bve\ fve &= \varepsilon[\![\theta\text{: } c_1]\!]\ bve\ fve \\
&\quad \text{where } e = c_1 :: e_2 \\
\varepsilon[\![\theta\text{: tl } e]\!]\ bve\ fve &= \varepsilon[\![\theta\text{: } e_2]\!]\ bve\ fve \\
&\quad \text{where } e = c_1 :: e_2
\end{aligned}$$

Figure 2.2: PULS syntax and semantic

Semantic Function

$$\begin{array}{rcl}
\varepsilon\llbracket\theta\text{: } e_1 \text{ with } e_2\rrbracket\ bve\ fve & = & (\varepsilon\llbracket\theta_1\text{: } e_1\rrbracket\ bve\ fve) \text{ with } (\varepsilon\llbracket\theta_2\text{: } e_2\rrbracket\ bve\ fve) \\
\varepsilon\llbracket\theta\text{: rep } e\rrbracket\ bve\ fve & = & \varepsilon\llbracket\theta_2\text{: } c_2\rrbracket\ bve\ fve \\
 & & \text{where } e = e_1 \text{ with } c_2 \\
\varepsilon\llbracket\theta\text{: } e_1 \text{ less } e_2\rrbracket\ bve\ fve & = & \varepsilon\llbracket\theta_3\text{: } e_3\rrbracket\ bve\ fve \\
 & & \text{where } e_1 = e_3 \text{ with } c_4 \text{ and } e_2 = c_4 \\
\varepsilon\llbracket\theta\text{: } e_1 \text{ in } e_2\rrbracket\ bve\ fve & = & (\varepsilon\llbracket c_3 == c_5\rrbracket\ bve\ fve) \rightarrow 1,(\varepsilon\llbracket c_3 \text{ in } e_4\rrbracket\ bve\ fve) \\
 & & \text{where } e_1 = c_3 \text{ and } e_2 = e_4 \text{ with } c_5 \\
\varepsilon\llbracket\theta\text{: if } e_0 \text{ then } e_1 \text{ else } e_2\rrbracket\ bve\ fve & = & (\varepsilon\llbracket\theta_0\text{: } e_0\rrbracket bve) \rightarrow (\varepsilon\llbracket\theta_1\text{: } e_1\rrbracket bve), (\varepsilon\llbracket\theta_2\text{: } e_2\rrbracket bve) \\
\varepsilon\llbracket\theta\text{: let } e_0 \text{ in } e_1\rrbracket\ bve\ fve & = & \varepsilon\llbracket\theta_1\text{: } e_1\rrbracket bve[\varepsilon\llbracket\theta_0\text{: } e_0\rrbracket bve] \\
\varepsilon\llbracket f_i(e_1,\ldots,e_n)\rrbracket\ bve\ fve & = & fve\llbracket f_i\rrbracket(\varepsilon\llbracket e_1\rrbracket bve,\ldots,\varepsilon\llbracket e_n\rrbracket bve) \\
\varepsilon_f\llbracket\text{fun } f_i(x_1,\ldots,x_n) = e_i\rrbracket & = & fve[(\lambda(y_1,\ldots,y_n).\varepsilon\llbracket e_i\rrbracket[y_k/x_k]fve)/f_i]
\end{array}$$

Figure 2.3: PLUS syntax and semantic continue

three phases:

1. *Parsing* determines the grammatical form of a declaration.

2. *Elaboration* is the static phase which determines whether it is well-typed and well-formed.

3. Evaluation is the dynamic phase which determines the value of the declaration in the environment.

We show the syntax and semantic of PLUS is showed in figure 2.3 and 2.3. PLUS has two kinds of derived lexical forms: one for **list** and the other for **set**. They are showed in the following table.

Derived Form	Equivalent Form
$[lit_1,\ldots,lit_n]$	lit_1 :: … :: lit_n :: nil
$\{lit_1,\ldots,lit_n\}$	emp with lit_1 with … with lit_n
$[lit_1..lit_n]$	lit_1 :: … :: lit_n :: nil
$\{lit_1..lit_n\}$	emp with lit_1 with … with lit_n

The **fst** and **snd** are products operator. The **fst** or **snd** extracts first or second element from a product,respectively. The **hd** and **tl** are list operator. The **hd** extracts first element from a list. The **tl** extract the part of list except first element from a list. The **rep**, **with** and **in** are set operator. The **with** add an element to a set. The **in** test if the specific element is in the set.

2.4 Time Stamp

Here we use time stamps [CY00, CY01] functions to label expressions in a program with lexicographically ordered time stamps. The functions are defined in figure 2.4. The time stamps have following structure. $\theta = \langle i, \alpha \rangle :: \theta | nil$. where i is the simple time stamp at current if-level and α is defined as

$$\alpha = \begin{cases} 1 \quad if \quad in \quad the \quad consequence \quad of \quad an \quad if \quad expression \\ 2 \quad if \quad in \quad the \quad alternative \quad of \quad an \quad if \quad expression \\ 0 \quad otherwise \end{cases}$$

We use an auxiliary function $Inc(\theta)$ to give the time stamp after the expression.

$$Inc(\langle i, \alpha \rangle :: \theta) = \begin{cases} \langle i+1, \alpha \rangle & if \quad \alpha = 0 \\ \langle i, \alpha \rangle :: Inc(\theta) & otherwise \end{cases}$$

The auxiliary function $C(\theta_1, \theta_2)$ for comparing two time stamps is defined as follows.

$C(< i, \alpha_1 >:: \theta_1, < i, \alpha_2 >:: \theta_2)$

$$= \begin{cases} -1 & if \quad i_1 < i_2 \quad or(i_1 = i_2 \quad and \quad \alpha_1 = 0 \quad and \quad \alpha_2 = 0) \\ 1 & if \quad i_1 > i_2 \quad or(i_1 = i_2 \quad and \quad \alpha_2 = 0 \quad and \quad \alpha_1 > 0) \\ 0 & if \quad i_1 = i_2 \quad (i_1 = i_2 \quad and \quad \alpha_1 = \alpha_2 = 0) \\ 2 & if \quad i_1 = i_2 \quad (i_1 = i_2 \quad and \quad \alpha_1 + \alpha_2 = 3) \\ C(\theta_1, < i, \theta_2) & otherwise \end{cases}$$

We may compare two program points with their time stamps θ_1 an θ_2. $\theta_1 < \theta_2$ means the single threaded execution goes to θ_1 earlier than θ_2. $\theta_1 > \theta_2$ means the single threaded execution goes to θ_1 later than θ_2. $\theta_1 = \theta_2$ means the two program points happen at the same time in a single threaded execution. We use $\theta.i(\alpha)$ for a shorthand of $\ldots \langle i, \alpha \rangle :: nil$. We use $\theta(\alpha)$ for a shorthand of $\theta.i(\alpha)$ when the value of i is insignificant. We use $\theta.i$ for a shorthand of $\theta.i(\alpha)$ when $\alpha = 0$.

Semantic Domains

Θ the domain of lexicograplhically ordered time stamps

Semantic Function

$\varepsilon_t : Exp \rightarrow \theta \rightarrow \theta : Exp$

$$
\begin{array}{rcl}
\varepsilon_t[\![c]\!]\ \theta & = & \theta : n,[Inc(\theta)/\theta] \\
\varepsilon_t[\![x]\!]\ \theta & = & \theta : x,[Inc(\theta)/\theta] \\
\varepsilon_t[\![!\ e]\!]\ \theta & = & \theta :!\ e,[Inc(\theta)/\theta] \\
\varepsilon_t[\![(e_1, e_2)]\!]\ \theta & = & \theta : (e_1, e_2),[Inc(\theta)/\theta] \\
\varepsilon_t[\![\text{fst}\ e]\!]\ \theta & = & \theta :\text{fst}\ e,[Inc(\theta)/\theta] \\
\varepsilon_t[\![\text{snd}\ e]\!]\ \theta & = & \theta :\text{snd}\ e,[Inc(\theta)/\theta] \\
\varepsilon_t[\![e_1\ op\ e_2]\!]\ \theta & = & \theta : e_1\ \text{op}\ e_2,[Inc(\theta)/\theta] \\
\varepsilon_t[\![x = e]\!]\ \theta & = & \theta : x = e,[Inc(\theta)/\theta] \\
\varepsilon_t[\![e_1 :: e_2]\!]\ \theta & = & \theta : e_1 :: e_2,[Inc(\theta)/\theta] \\
\varepsilon_t[\![hd\ e]\!]\ \theta & = & \theta :\text{hd}\ e,[Inc(\theta)/\theta] \\
\varepsilon_t[\![tl\ e]\!]\ \theta & = & \theta :\text{tl}\ e,[Inc(\theta)/\theta] \\
\varepsilon_t[\![(e_1\ \text{with}\ e_2)]\!]\ \theta & = & \theta : e_1\ \text{with}\ e_2,[Inc(\theta)/\theta] \\
\varepsilon_t[\![\text{rep}\ e]\!]\ \theta & = & \theta :\text{rep}\ e,[Inc(\theta)/\theta] \\
\varepsilon_t[\![(e_1\ \text{less}\ e_2)]\!]\ \theta & = & \theta : e_1\ \text{less}\ e_2,[Inc(\theta)/\theta] \\
\varepsilon_t[\![(e_1\ \text{in}\ e_2)]\!]\ \theta & = & \theta : e_1\ \text{in}\ e_2,[Inc(\theta)/\theta] \\
\varepsilon_t[\![f\ (e_1\ \ldots\ e_n)\ ]\!]\ \theta & = & f\ ((\theta_1 : e_1)\ \ldots\ (\theta_n : e_n)),[Inc(\theta)/\theta] \\
 & & \text{where } \theta_1 = \theta, \text{ and } [\theta_{i+1} = Inc(\theta_i)]_1^{n-1} \\
\varepsilon_t[\![\ \text{let}\ e_0\ \text{in}\ e_1]\!]\ \theta & = & \text{let}\ \varepsilon_t[\![e_0]\!]\ \theta\ \text{in}\ \varepsilon_t[\![e_1]\!]\ \theta_1,[Inc(\theta_1)/\theta] \\
 & & \text{where } \theta_1 = Inc(\theta) \\
\varepsilon_t[\![\ \text{if}\ e_0\ \text{then}\ e_1\ \text{else}\ e_2]\!]\ \theta & = & \text{if}\ \varepsilon_t[\![e_0]\!]\ \theta\ \text{then}\ \varepsilon_t[\![e_1]\!]\ \theta_1\ \text{else}\ \varepsilon_t[\![e_2]\!]\ \theta_2,[Inc(\theta)/\theta] \\
 & & \text{where } \theta_1 = \theta(1).1, \text{ and } \theta_2 = \theta(2).1 \\
\varepsilon_t[\![\text{fun}\ f_i(x_1, \ldots, x_n) = e_i]\!]\ \theta & = & \text{fun}\ f_i(x_1, \ldots, x_n) = \varepsilon_t[\![e_i]\!]\theta
\end{array}
$$

Figure 2.4: Augmented time stamp semantics

Chapter 3

Semantic Functions

Program slices consist of statements that affect the variables interesting us. The original concept of program slicing is introduced by Wiser. They define program slicing as date-flow problem. And they compute data-flow information on control flow graph. Then K. Ottenstein and L. Ottenstein compute program slicing with program dependence graph. They define program slicing as reachable problem on program dependence graph. Instead of using dependence graph, we propose a semantic method of program slicing using abstract interpretation. We introduce our approach of computing program slice in this chapter. We explain how we extract necessary information from each expression for computing program slice and show these semantic functions. First, we compute the attributes $\mathcal{D}$ and $\mathcal{U}$ of slicing information for each expression. Second,we compute the attribute $\mathcal{R}$ of slicing information of all expression. Last,we can get the program slicing according to these information.

3.1 Augmented Functions

Program slicing is an approach of gathering the expression that affects the value of variables at specific point. The original definition of program slicing is proposed by Wiser [WM79, WM81, WM84]. Typically, we give a slicing goal G for computing program slice. G is two tuple element $\langle \theta_g, V_g \rangle$. Where, θ_g is the time stamp of the expression interesting us and V_g is a set of variables interesting us in the expression with time stamp θ_g. Program slices with respect to slicing goal $\langle \theta_g, V_g \rangle$ consist of statements that affect the variables of V_g in the expression with time stamp θ_g. For the purpose of computing program slice, we need some information to help us decide which expression is selected with respect to the slicing goal. We call the information *slicing information.* For computing program slice with respect to slicing goal $\langle \theta_g, V_g \rangle$, we use *relevant set* denoted by $\mathcal{R}$ to keep the variables which affect the variables in V_g.

Lemma 1 *For the segment of program, "$a = b + 1; c = a + 1;$", we say that the variable a directly affects the variable c and the variable b indirectly(transitively) affects the variable c.*$\square$

So we must know what variables are defined and used in an expression. We use *definition set* denoted by $\mathcal{D}$ or *use set* denoted by $\mathcal{U}$ to keep the variables which defined or used in an expression,respectively. Slicing information of an expression $\theta : exp$, $\iota = ibve(\theta)$ is a element of three tuple,$\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta) \rangle$. The attribute $\mathcal{D}(\theta)$ is a set of variables that defined at the expression. The attribute $\mathcal{U}(\theta)$ is a set of variables that used at the expression. The attribute

$\mathcal{R}(\theta)$ is a set of variables that directly or indirectly affect the variables in V_g. Because we compute attribute $\mathcal{R}$ with attributes $\mathcal{D}$ and $\mathcal{U}$,we must first evaluate attributes $\mathcal{D}$ and $\mathcal{U}$.

Theorem 1 $\forall_{\theta:e\in E}$, $\varepsilon_i[\![\theta\colon e]\!]$ *ibve can correctly extract the variables that are defined or used in the expression* e *with time stamp* θ*.*

Proof: We prove it by structural induction.

case 1. $\varepsilon_i[\![\theta\colon c]\!]\ \mathit{ibve} = \mathit{ibve}$

No variable is in the expression. No change is needed for *ibve*.

case 2. $\varepsilon_i[\![\theta\colon x]\!]\ \mathit{ibve} = \mathit{ibve}[\langle\mathcal{D}(\theta), \mathcal{U}(\theta) \cup \{\theta\}, \mathcal{R}(\theta)\rangle/\theta]$

A variable x appears in the expression, which means that x is used in the expression. We add the variable to use set.

case 3. $\varepsilon_i[\![\theta\colon x = e_0]\!]\ \mathit{ibve} = \varepsilon_i[\![\theta_0\colon e_0]\!]([\langle\{x\}, \mathcal{U}(\theta_0), \mathcal{R}(\theta)\rangle])$

For a binding expression, the expression defines a variable and may use variables. So we first test if any variables are used in subexpression e_0. By case 1 and case 2, we know that we can correctly extract information from subexpression e_0. Then we add the variable that is defined to the definition set.

case 4. $\varepsilon_i[\![\theta\colon \mathrm{tl}\ e]\!]\ \mathit{ibve} = \varepsilon_i[\![\theta\colon e]\!]\ \mathit{ibve}$

For the expression, it update no variables. So we only test if any variables that are used in the subexpression. By case 1, case 2 and case 3, we know that we know that we can correctly extract information from subexpression.

case 5. $e = !e \mid \mathrm{tl}\ e$

It is similar to case 4.

case 6. $\varepsilon_i[\![\theta\colon e_1 :: e_2]\!]\ ibve = \varepsilon_i[\![\theta_2\colon e_2]\!](\varepsilon_i[\![\theta_1\colon e_1]\!]\ ibve)$

For a list construction expression, we test if any variables are used in the subexpressions. By repetitively applying the above cases, we know that we can correctly extract information from the subexpressions.

case 7. $e = e_1\ op\ e_2$

It is similar to case 6.

case 8.$\varepsilon_i[\![\mathsf{let}\ \theta_0\colon e_0\ \mathsf{in}\ \theta_1\colon e_1]\!]\ ibve = \varepsilon_i[\![\theta_1\colon e_1]\!](\varepsilon_i[\![\theta_0\colon e_0]\!]\ ibve)$

A let expression consists of two subexpressions. We respectively extract information for the two subexpressions. By repetitively applying the above cases, we know that we can correctly extract information from the subexpressions.

case 9. $\varepsilon_i[\![\ \mathrm{if}\ \theta_0\colon e_0\ \mathrm{then}\ \theta_1\colon e_1\ \mathrm{else}\ \theta_2\colon e_2]\!]\ ibve = \varepsilon_i[\![\theta_2\colon e_2]\!](\varepsilon_i[\![\theta_1\colon e_1]\!](\varepsilon_i[\![\theta_0\colon e_0]\!]\ ibve))$

An if expression consists of three subexpressions. We respectively extract information for the three subexpressions. By repetitively applying the above cases, we can correctly extract informations from the subexpressions.

Hence, we know that $\forall_{\theta:e\in E}$, $\varepsilon_i[\![\theta\colon e]\!]\ ibve$ can correctly extract informations for each expression. $\square$

In figure 3.1, we define Augmented functions to compute attributes $\mathcal{D}$ and

Semantic Domains

$$\begin{array}{lcl} \mathcal{D} & = & 2^{Bv} \\ \mathcal{U} & = & 2^{Bv} \\ \mathcal{R} & = & 2^{Bv} \\ Ibve & = & \theta \longrightarrow \langle \mathcal{D}, \mathcal{U}, \mathcal{R} \rangle \end{array}$$

Semantic Function

$$\varepsilon_i[\![\theta : exp]\!] \rightarrow Ibve \rightarrow Ibve$$

$$\begin{array}{rcl}
\varepsilon_i[\![\theta\text{: } c]\!]\ ibve & = & ibve \\
\varepsilon_i[\![\theta\text{: } x]\!]\ ibve & = & ibve[\langle \phi, \mathcal{U}(\theta) \cup \{x\}, \phi \rangle / \theta] \\
\varepsilon_i[\![\theta\text{: } !e]\!]\ ibve & = & \varepsilon[\![\theta\text{: } e]\!]\ ibve \\
\varepsilon_i[\![\theta\text{: } e_1 \text{ op } e_2]\!]\ ibve & = & \varepsilon_i[\![\theta_2\text{: } e_2]\!](\varepsilon_i[\![\theta_1 : e_1]\!]\ ibve) \\
\varepsilon_i[\![\theta\text{: } x = e_0]\!]\ ibve & = & (\varepsilon_i[\![\theta_0\text{: } e_0]\!]\ ibve)[\langle \{x\}, \mathcal{U}(\theta_0), \phi \rangle / \theta] \\
\varepsilon_i[\![\theta\text{: } e_1 :: e_2]\!]\ ibve & = & \varepsilon_i[\![\theta_2\text{: } e_2]\!](\varepsilon_i[\![\theta_1\text{: } e_1]\!]\ ibve) \\
\varepsilon_i[\![\theta\text{: hd } e]\!]\ ibve & = & \varepsilon_i[\![\theta\text{: } e]\!]\ ibve \\
\varepsilon_i[\![\theta\text{: tl } e]\!]\ ibve & = & \varepsilon_i[\![\theta\text{: } e]\!]\ ibve \\
\varepsilon_i[\![\theta\text{: } e_1 \text{ with } e_2]\!]\ ibve & = & \varepsilon_i[\![\theta_2\text{: } e_2]\!](\varepsilon_i[\![\theta_1\text{: } e_1]\!]\ ibve) \\
\varepsilon_i[\![\mathsf{let}\ \theta_0\text{: } e_0\ \mathsf{in}\ \theta_1\text{: } e_1]\!]\ ibve & = & \varepsilon_i[\![\theta_1\text{: } e_1]\!](\varepsilon_i[\![\theta_0\text{: } e_0]\!]\ ibve) \\
\varepsilon_i[\![\text{ if } \theta_0\text{: } e_0 \text{ then } \theta_1\text{: } e_1 \text{ else } \theta_2\text{: } e_2]\!]\ ibve & = & \varepsilon_i[\![\theta_2\text{: } e_2]\!](\varepsilon_i[\![\theta_1\text{: } e_1]\!](\varepsilon_i[\![\theta_0\text{: } e_0]\!]\ ibve)) \\
\mathcal{A}_i[\![\{\theta_1\text{: } e_1; \ldots; \theta_k\text{: } e_k;\}]\!] & = & \varepsilon_i[\![\theta_k\text{: } e_k]\!](\ldots(\varepsilon_i[\![\theta_1\text{: } e_1]\!]\ ibve)\ldots)
\end{array}$$

Figure 3.1: The Augmented Functions of computing attributes $\mathcal{D}$ and $\mathcal{U}$ of slicing information

$\mathcal{U}$ of slicing information for each expression. We can get attributes $\mathcal{D}$ and $\mathcal{U}$ for each expression by scannig source program one time. There $\mathcal{D}$ of exprs-sions except assignment expression are empty,because these expressions don't update any variable. In this phase, we don't evaluate the $\mathcal{R}$. So we initially assign ϕ to attribute $\mathcal{R}$.

3.2 Intra-procedural Analysis

In this section, we will show our approach of computing program slice. And we assume the source program is intra-procedural program. For program slicing analysis, we usually give a slicing goal. A slicing goal is $\langle \theta_g, V_g \rangle$. A slicing goal $\langle \theta_g, V_g \rangle$ means that we gather the statements that affect the variables in V_g in the statement with time stamp θ_g. The $\mathcal{R}$ consists of the variables that affecting directly or indirectly value of variables in V_g.

Lemma 2 *An expression affects the variables in V_g if there is a definition of a variable that is in $\mathcal{R}$ in the expression.* $\Box$

Program slice with respect to the slicing goal consist of expressions that define the variables that are observed for the slicing goal. We define an augmented functions of computing attributes $\mathcal{R}$ of slicing information and gather the statements that affect V_g in figure 3.2. Where $\Im$ is a set of time stamps of the expressions that affect the observed variables. We use the augmented functions to find the variables which affect variables observed with respect to slicing goal $\langle \theta_g, V_g \rangle$. We start to compute backwardly program slice from the expression with time stamp θ_g and assign V_g to $\mathcal{R}(\theta_g)$. An expression maybe indirectly affects the value of variables observed for slicing goal when the expression is an assignment. If an expression can not affect the value of the observed variables, one's relevant set inherit from the relevant set of its immediate successor. So we can use a formula[BG96, Tip95, WM79, WM81, WM84] to compute $\mathcal{R}$ for each expression. We show it as follows. $\mathcal{R}(\theta) = ((\mathcal{R}(\theta_1) - \mathcal{D}(\theta)) \cup (\mathcal{U}(\theta)$ if $\mathcal{D}(\theta) \cap \mathcal{R}(\theta_1) \neq \phi)$. Where θ_1 is a immediate successor of θ.

How do we know if an expression affect the observed variable? For an expression, $\theta : x = e$, we can use the operation, $\mathcal{D}(\theta) \cap \mathcal{R}(\theta_1)$, to test whether the expression affects the value of the observed variables or not. If the result of the operation is empty, we say that the expression will not affect the value of the observed variable. If not, we say that the expression will affect the value of the observed variable. We can know that all definitions of variable x before the expression can't affect the value of the observed variables by a concept of reaching definition. So we remove $\{x\}$ from the relevant set. If an expression affects the value of the observed variables, the variables in $\mathcal{U}(\theta)$ will indirectly affect the value of the observed variables. The ones must be added to relevant set. For an assignment expression: $\theta : exp$, $\mathcal{R}(\theta) = ((\mathcal{R}(\theta_1) - \mathcal{D}(\theta)) \cup \mathcal{U}(\theta))$. And relevant set whose others expressions don't update variables inherit from its immediate successor.

Please note the If-expression. It has two immediate successor.So the relevant set of If-expression is union of relevant sets of two immediate successor. The other focus of If-expression is that the variables which are used in condition of If-expression indirectly affect the observed variables when either of its two immediate successor affects the observed variables. So the ones must be added to relevant set.

Theorem 2 *$\forall_{\theta:e\in E}$, $\varepsilon_r[\![\theta\colon e]\!]$ env can correctly compute $\mathcal{R}$.*

Proof: We prove it by structural induction.

case 1. $\varepsilon_r[\![\theta\colon c]\!]\ env = \langle[\langle\mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle/\theta], \Im\rangle$, where $\theta_1 = Inc(\theta)$

By $\mathcal{R}(\theta) = (\mathcal{R}(\theta_1) - \mathcal{D}(\theta)) \cup (\mathcal{U}(\theta)$ if $\mathcal{D}(\theta) \cap \mathcal{R}(\theta_1) \neq \phi)$. There are no definition of variables in the expression. So $\mathcal{D}(\theta) = \phi, \mathcal{D}(\theta) \cap \mathcal{R}(\theta_1) = \phi$. Then we can known $\mathcal{R}(\theta) = \mathcal{R}(\theta)$. And the expression is not be selected.

case 2. $e = x \mid !e \mid e_1$ op $e_2 \mid \mathsf{hd}\ e \mid \mathsf{tl}\ e$

It is similar to case 1.

case 3. $\varepsilon_r[\![\theta\colon x = e_0]\!]$ *env*

There is a definition of a variable in the expression. We will test if the definition affects any of the observed variables.

(a) $\mathcal{D}(\theta) \cap \mathcal{R}(\theta_1) \neq \phi$

It means that there is a definition of a variable which is in the relevant set. So we say that the expression affects one of the observed variables. We must update the relevant set and add the time stamp of the expression to $\Im$. So *env* is equal to $\langle[\langle\mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}'\rangle/\theta], [\Im \cup \{\theta\}]\rangle$. Where $\mathcal{R}' = (\mathcal{R}(\theta_1) - \mathcal{D}(\theta)) \cup \mathcal{U}(\theta)$

(b) $\mathcal{D}(\theta) \cap \mathcal{R}(\theta_1) = \phi$

It is means that the variable defined is not in the relevant set. So we say the expression don't affect the observed variables. The *env* is remained $\langle[\langle\mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle], \Im\rangle$

case 4. $\varepsilon_r[\![\theta\colon \text{if } e_0 \text{ then } e_1 \text{ else } e_2\ ]\!]$ *env*

Because expression e_0 affects the execution of e_1 and e_2. We say e_0 indirectly affects the observed variables if either of e_1 and e_2 affects the observed variables. So we test if either of e_1 and e_2 affects the observed variable. BY case 1, 2, and 3, we can correctly extract the information of relevant set and statements that affect the observed variables.

(a) If either of e_1 and e_2 affects the observed variables.

The e_0 indirectly affects the observed variables, and hence the variables in e_0 are also included as the observed variables. So we add the time stamp of e_0 to $\Im$ and variables in e_0 to the relevant set.

Therefore, $env = \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{R}(\theta_1)\cup\mathcal{R}(\theta_2)\cup\mathcal{U}(\theta)\rangle/\theta],[\Im\cup\{\theta\}]\rangle$, where $\theta_1 = Inc(\theta)$.

(b) If none of e_1 and e_2 affects the observed variables.

The e_0 does not affects the observed variables. No change is needed for the relevant set. So $env = \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{R}(\theta_1)\cup\mathcal{R}(\theta_2)\rangle/\theta],\Im\rangle$

case5 $\varepsilon_r[\![\theta\text{: let } e_0 \text{ in } e_1]\!]\ env = \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{R}(\theta_1)\rangle/\theta],\Im\rangle$, where $\theta_1 = Inc(\theta)$

We analyze the subexpressions e_0 and e_1. By repetitively applying the above cases, we can correctly extract the information.

Hence, $\forall_{\theta:e\in E}$, $\varepsilon_r[\![\theta\text{: } e]\!]\ env$ can correctly compute $\mathcal{R}$. □

Theorem 3 *$\forall_{\theta:e\in E}$, $\varepsilon_r[\![\theta\text{: } e]\!]\ env$ can correctly gather the expressions that affect variables in V_g.*

Proof: Theorem 3 is just a corollary of Theorem 2. □

We use function $\mathcal{P}$ to compute program slice with respect to slicing goal G. It compute backwardly program slicing from the expression with time stamp θ_g. First, we compute $\mathcal{D}$ and $\mathcal{U}$ for each expressions. Then we compute $\mathcal{R}$ and gather the statements that affect the variables in V_g. We assign V_g to $\mathcal{R}$ and θ_g to $\Im$ in the beginning of computing relevant set. We define $\mathcal{P}$ as follows.

Definition 1 Static Program Slicing

$G : \theta \times 2^{bv}$

$\Im : 2^{\theta}$

$\mathcal{P} : Pr \times Env \times Goal \rightarrow Env$

$\mathcal{P}(Pr, \langle Ibve, \Im\rangle, \langle\theta_g, V_g\rangle)$

$$= \varepsilon_r[\![\theta_1\colon e_1]\!](\ldots(\varepsilon_r[\![\theta_{g-1}\colon e_{g-1}]\!]\ \langle \mathcal{A}_i(Pr)[\langle \mathcal{D}(\theta_g), \mathcal{U}(\theta_g), \mathcal{U}(\theta_g)\rangle/\theta_g], [\Im \cup \{\theta_g\}]\rangle)\ldots) \blacksquare$$

G is a slicing goal. θ_{g-1} is equal to $Dec(\theta_g)$. We define $Dec(\theta)$ as follows.

$$Dec(\theta :: \langle i, \alpha\rangle) = \begin{cases} \theta :: \langle i-1, \alpha\rangle & if \quad \alpha = 0 \quad and \quad i-1 \neq 0 \\ Desc(\theta) & if \quad \alpha = 0 \quad and \quad i-1 = 0 \\ \langle i-1, 0\rangle & otherwise \end{cases}$$

We use $Dec(\theta)$ to compute the immediate predecessor of θ. If the expression affects the observed variables, we add the time stamp to Ps. And $\Im$ is a set of time stamps of the expressions which affect the variables in V_g. Program slice with respect to slicing goal make up of the expressions whose time stamp is in $\Im$.

We show an example in figure 3.5. We explain our approach by the example. For a given program, we compute $\mathcal{D}$ and $\mathcal{U}$ for each expression. We show the work of computing $\mathcal{D}$ and $\mathcal{U}$ for each expression in figure 3.3. And we show the result in figure 3.5(b). Then we compute program slice with respect to slicing goal $\langle\langle 6, 0\rangle, \{e\}\rangle$. We start from the expression with time stamp $\langle 6, 0\rangle$ and compute backwardly program slice. We assign $\{e\}$ to $\mathcal{R}(\langle 6, 0\rangle)$ and add the expression to sliced program. Then we compute $\mathcal{R}(\langle 5, 0\rangle)$. Because of $\mathcal{D}(\langle 5, 0\rangle) \cap \mathcal{R}(\langle 6, 0\rangle) \neq \phi$, $\mathcal{R}(\langle 5, 0\rangle) = \mathcal{R}(\langle 6, 0\rangle) = \{e\}$. We don't add the expression to sliced program. The others can be computed by the method show in figure 3.2. And we show the result of computing program slice in figure 3.5.

Semantic Domain

$$\begin{array}{rcl} Ibve & = & \theta \rightarrow \langle \mathcal{D}, \mathcal{U}, \mathcal{R} \rangle \\ \Im & = & 2^{\theta} \\ Env & = & Ibve \times \Im \end{array}$$

Semantic Function

$\varepsilon_r[\![\theta : exp]\!] \rightarrow Env \rightarrow Env$

$$\begin{array}{rcl}
\varepsilon_r[\![\theta\text{: } c]\!]\ env & = & \langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle/\theta], \Im\rangle, \text{ where } \theta_1 = Inc(\theta) \\
\varepsilon_r[\![\theta\text{: } x]\!]\ env & = & \langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle/\theta], \Im\rangle, \text{ where } \theta_1 = Inc(\theta) \\
\varepsilon_r[\![\theta\text{: } !e]\!]\ env & = & \langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle/\theta], \Im\rangle, \text{ where } \theta_1 = Inc(\theta) \\
\varepsilon_r[\![\theta\text{: } e_1 \text{ op } e_2]\!]\ env & = & \langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle/\theta], \Im\rangle, \text{ where } \theta_1 = Inc(\theta) \\
\varepsilon_r[\![\theta\text{: } x = e_0]\!]\ env & = & \begin{cases} \langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}'\rangle/\theta], [\Im \cup \{\theta\}]\rangle & \text{if } \mathcal{D}(\theta) \cap \mathcal{R}(\theta_1) \neq \phi \\ \text{where } \mathcal{R}' = (\mathcal{R}(\theta_1) - \mathcal{D}(\theta)) \cup \mathcal{U}(\theta) & , \theta_1 = Inc(\theta) \\ \\ \langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle/\theta], \Im\rangle & \text{if } \mathcal{D}(\theta) \cap \mathcal{R}(\theta_1) = \phi \\ \text{where} \theta_1 = Inc(\theta) \end{cases} \\
\varepsilon_r[\![\theta\text{: } e_1 :: e_2]\!]\ env & = & \langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle/\theta], \Im\rangle, \text{ where } \theta_1 = Inc(\theta) \\
\varepsilon_r[\![\theta\text{: hd } e]\!]\ env & = & \langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle/\theta], \Im\rangle, \text{ where } \theta_1 = Inc(\theta) \\
\varepsilon_r[\![\theta\text{: tl } e]\!]\ env & = & \langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle/\theta], \Im\rangle, \text{ where } \theta_1 = Inc(\theta) \\
\varepsilon_r[\![\theta\text{: let } e_0 \text{ in } e_1]\!]\ env & = & \langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1)\rangle/\theta], \Im\rangle, \text{ where } \theta_1 = Inc(\theta) \\
\varepsilon_r[\![\theta\text{: if } e_0 \text{ then } e_1 \text{ else } e_2\ ]\!]\ env & = & \begin{cases} (\varepsilon_r[\![\theta_1 : e_1]\!](\epsilon_r[\![\theta_2 : e_2]\!]env)) \\ [\langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1) \cup \mathcal{R}(\theta_2) \cup \mathcal{U}(\theta)\rangle/\theta], [\Im \cup \{\theta\}]\rangle] \\ \text{if } \mathcal{D}(\theta_1) \cap \mathcal{R}(\theta_3) \neq \phi \text{ or } \mathcal{D}(\theta_2) \cap \mathcal{R}(\theta_4) \neq \phi \\ \text{where} \theta_3 = Inc(\theta_1), \theta_4 = Inc(\theta_2) \\ \\ (\varepsilon_r[\![\theta_1 : e_1]\!](\epsilon_r[\![\theta_2 : e_2]\!]env)) \\ [\langle [\langle \mathcal{D}(\theta), \mathcal{U}(\theta), \mathcal{R}(\theta_1) \cup \mathcal{R}(\theta_2)\rangle/\theta], \Im\rangle] \\ \text{otherwise} \\ \text{where} \theta_3 = Inc(\theta_1), \theta_4 = Inc(\theta_2) \end{cases}
\end{array}$$

Figure 3.2: The semantic function of computing $\mathcal{R}$ and program slices

$\varepsilon_i[\![\langle 1,0\rangle : a = [5,4,3]\ ]\!]ibve = (\varepsilon_i[\![\langle 1,0\rangle : 5 :: 4 :: 3]\!]ibve)[\langle\{x\},\mathcal{U}(\langle 1,0\rangle),\phi\rangle/\langle 1,0\rangle]$
$(\varepsilon_i[\![\langle 1,0\rangle : 4 :: 3]\!](\varepsilon_i[\![\langle 1,0\rangle : 5]\!]ibve))[\langle\{x\},\mathcal{U}(\langle 1,0\rangle),\phi\rangle/\langle 1,0\rangle]$
$(\varepsilon_i[\![\langle 1,0\rangle : 4 :: 3]\!]ibve)[\langle\{x\},\mathcal{U}(\langle 1,0\rangle),\phi\rangle/\langle 1,0\rangle]$
$= (\varepsilon_i[\![\langle 1,0\rangle : 3]\!](\varepsilon_i[\![\langle 1,0\rangle : 4]\!]ibve[\langle\phi,\phi,\phi\rangle/\langle 1,0\rangle]))[\langle\{a\},\mathcal{U}(\langle 1,0\rangle),\phi\rangle/\langle 1,0\rangle]$
$= (\varepsilon_i[\![\langle 1,0\rangle : 3]\!]ibve)[\langle\{a\},\mathcal{U}(\langle 1,0\rangle),\phi\rangle/\langle 1,0\rangle]$
$= ibve[\langle\{a\},\phi,\phi\rangle/\langle 1,0\rangle]$

$\varepsilon_i[\![\langle 2,0\rangle : c = 6 :: a]\!]ibve = (\varepsilon_i[\![\langle 2,0\rangle : 6 :: a]\!]ibve)[\langle\{c\},\mathcal{U}(\langle 2,0\rangle),\phi\rangle/\langle 2,0\rangle]$
$= (\varepsilon_i[\![\langle 2,0\rangle : a]\!](\varepsilon_i[\![\langle 2,0\rangle : 6]\!]ibve))[\langle\{c\},\mathcal{U}(\langle 2,0\rangle),\phi\rangle/\langle 2,0\rangle]$
$= (\varepsilon_i[\![\langle 2,0\rangle : a]\!]ibve)[\langle\{c\},\mathcal{U}(\langle 2,0\rangle),\phi\rangle/\langle 2,0\rangle]$
$= (ibve[\langle\phi,\phi\cup\{a\},\phi\rangle/\langle 2,0\rangle])[\langle\{c\},\mathcal{U}(\langle 2,0\rangle),\phi\rangle/\langle 2,0\rangle]$
$= ibve[\langle\{c\},\{a\},\phi\rangle/\langle 2,0\rangle]$

$\varepsilon_i[\![\langle 3,0\rangle : d = [a,c]\ ]\!]ibve = (\varepsilon_i[\![\langle 3,0\rangle : a :: c]\!]ibve)[\langle\{d\},\mathcal{U}(\langle 3,0\rangle),\phi\rangle/\langle 3,0\rangle]$
$= ((\varepsilon_i[\![\langle 3,0\rangle : c]\!](\varepsilon_i[\![\langle 3,0\rangle : a]\!]ibve))[\langle\{d\},\mathcal{U}(\langle 3,0\rangle),\phi\rangle/\langle 3,0\rangle]$
$= (\varepsilon_i[\![\langle 3,0\rangle : c]\!]ibve[\langle\phi,\phi\cup\{a\},\phi/\langle 3,0\rangle])[\langle\{d\},\mathcal{U}(\langle 3,0\rangle),\phi\rangle/\langle 3,0\rangle]$
$= (ibve[\langle\phi,\{a\}\cup\{c\},\phi/\langle 3,0\rangle])[\langle\{d\},\mathcal{U}(\langle 3,0\rangle),\phi\rangle/\langle 3,0\rangle]$
$= ibve[\langle\{d\},\{a,c\},\phi\rangle/\langle 3,0\rangle]$

$\varepsilon_i[\![\langle 4,0\rangle : e = \text{tl}\ d]\!]ibve = (\langle 4,0\rangle : \varepsilon_i[\![\text{tl}\ d]\!]ibve)[\langle\{e\},\mathcal{U}(\langle 4,0\rangle),\phi\rangle/\langle 4,0\rangle]$
$= (\langle 4,0\rangle : \varepsilon_i[\![\langle 4,0\rangle : d]\!]ibve)[\langle\{e\},\mathcal{U}(\langle 4,0\rangle),\phi\rangle/\langle 4,0\rangle]$
$= (ibve[\langle\phi,\phi\cup\{d\},\phi/\langle 4,0\rangle])[\langle\{e\},\mathcal{U}(\langle 4,0\rangle),\phi\rangle/\langle 4,0\rangle]$
$= ibve[\langle\{e\},\{d\},\phi\rangle/\langle 4,0\rangle]$

$\varepsilon_i[\![\langle 5,0\rangle : f = \text{tl}\ c]\!]ibve = (\varepsilon_i[\![\langle 5,0\rangle : tl\ c]\!]ibve)[\langle\{f\},\mathcal{U}(\langle 5,0\rangle),\phi\rangle/\langle 5,0\rangle]$
$= (\varepsilon_i[\![\langle 5,0\rangle : c]\!]ibve)[\langle\{f\},\mathcal{U}(\langle 5,0\rangle),\phi\rangle/\langle 5,0\rangle]$
$= (ibve[\langle\phi,\phi\cup\{c\},\phi/\langle 5,0\rangle])[\langle\{f\},\mathcal{U}(\langle 5,0\rangle),\phi\rangle/\langle 5,0\rangle]$
$= ibve[\langle\{f\},\{c\},\phi\rangle/\langle 5,0\rangle]$

$\varepsilon_i[\![\langle 6,0\rangle : b = \text{hd}\ e]\!]ibve = (\varepsilon_i[\![\langle 6,0\rangle : \text{hd}\ e]\!]ibve)[\langle\{b\},\mathcal{U}(\langle 6,0\rangle),\phi\rangle/\langle 6,0\rangle]$
$= (\varepsilon_i[\![\langle 6,0\rangle : e]\!]ibve)[\langle\{b\},\mathcal{U}(\langle 6,0\rangle),\phi\rangle/\langle 6,0\rangle]$
$= (ibve[\langle\phi,\phi\cup\{e\},\phi/\langle 6,0\rangle])[\langle\{b\},\mathcal{U}(\langle 5,0\rangle),\phi\rangle/\langle 6,0\rangle]$
$= ibve[\langle\{b\},\{e\},\phi\rangle/\langle 6,0\rangle]$

Figure 3.3: Work of computing $\mathcal{D}$ and $\mathcal{U}$ for each expression

$\langle 6,0\rangle$
$\langle[\langle\mathcal{D}(\langle 6,0\rangle),\mathcal{U}(\langle 6,0\rangle),\{e\}\rangle/\langle 6,0\rangle],[\Im\cup\{\langle 6,0\rangle\}]$

$\varepsilon_r[\![\langle 5,0\rangle : f = \mathrm{tl}\ c]\!]env$
$= \langle[\langle\mathcal{D}(\langle 5,0\rangle),\mathcal{U}(\langle 5,0\rangle),\mathcal{R}(\langle 6,0\rangle)\rangle/\langle 5,0\rangle,\Im\rangle$
$= \langle[\langle\mathcal{D}(\langle 5,0\rangle),\mathcal{U}(\langle 5,0\rangle),\{e\}\rangle/\langle 5,0\rangle,\Im\rangle$

$\varepsilon_r[\![\langle 4,0\rangle : e = \mathrm{tl}\ d]\!]env$
$= \langle[\langle\mathcal{D}(\langle 4,0\rangle),\mathcal{U}(\langle 4,0\rangle),(\mathcal{R}(\langle 5,0\rangle)-\mathcal{D}(\langle 4,0\rangle))\cup\mathcal{U}(\langle 4,0\rangle)\rangle/\langle 4,0\rangle],[\Im\cup\{\langle 4,0\rangle\}]\rangle$
$= \langle[\langle\mathcal{D}(\langle 4,0\rangle),\mathcal{U}(\langle 4,0\rangle),\{e\}-\{e\}\cup\{d\}/\langle 4,0\rangle],[\{\langle 6,0\rangle\}\cup\{\langle 4,0\rangle\}]\rangle$
$= \langle[\langle\mathcal{D}(\langle 4,0\rangle),\mathcal{U}(\langle 4,0\rangle),\{d\}/\langle 4,0\rangle],[\{\langle 6,0\rangle,\langle 4,0\rangle\}]\rangle$

$\varepsilon_r[\![\langle 3,0\rangle : d = [a,c]\ ]\!]env$
$= \langle[\langle\mathcal{D}(\langle 3,0\rangle),\mathcal{U}(\langle 3,0\rangle),(\mathcal{R}(\langle 4,0\rangle)-\mathcal{D}(\langle 3,0\rangle))\cup\mathcal{U}(\langle 3,0\rangle)\rangle/\langle 3,0\rangle],[\Im\cup\{\langle 3,0\rangle\}]\rangle$
$= \langle[\langle\mathcal{D}(\langle 3,0\rangle),\mathcal{U}(\langle 3,0\rangle),\{d\}-\{d\}\cup\{a,c\}/\langle 3,0\rangle],[\{\langle 6,0\rangle,\langle 4,0\rangle\}\cup\{\langle 3,0\rangle\}]\rangle$
$= \langle[\langle\mathcal{D}(\langle 3,0\rangle),\mathcal{U}(\langle 3,0\rangle),\{a,c\}/\langle 4,0\rangle],[\{\langle 6,0\rangle,\langle 4,0\rangle,\langle 3,0\rangle\}]\rangle$

$\varepsilon_r[\![\langle 2,0\rangle : c = 6 :: a\ ]\!]env$
$= \langle[\langle\mathcal{D}(\langle 2,0\rangle),\mathcal{U}(\langle 2,0\rangle),(\mathcal{R}(\langle 3,0\rangle)-\mathcal{D}(\langle 2,0\rangle))\cup\mathcal{U}(\langle 2,0\rangle)\rangle/\langle 2,0\rangle],[\Im\cup\{\langle 2,0\rangle\}]\rangle$
$= \langle[\langle\mathcal{D}(\langle 2,0\rangle),\mathcal{U}(\langle 2,0\rangle),\{a,c\}-\{c\}\cup\{a\}/\langle 2,0\rangle],[\{\langle 6,0\rangle,\langle 4,0\rangle,\langle 3,0\rangle\}\cup\{\langle 2,0\rangle\}]\rangle$
$= \langle[\langle\mathcal{D}(\langle 2,0\rangle),\mathcal{U}(\langle 2,0\rangle),\{a\}/\langle 2,0\rangle],[\{\langle 6,0\rangle,\langle 4,0\rangle,\langle 3,0\rangle,\langle 2,0\rangle\}]\rangle$

$\varepsilon_r[\![\langle 1,0\rangle : a = [5,4,3]\ ]\!]env$
$= \langle[\langle\mathcal{D}(\langle 1,0\rangle),\mathcal{U}(\langle 1,0\rangle),(\mathcal{R}(\langle 2,0\rangle)-\mathcal{D}(\langle 1,0\rangle))\cup\mathcal{U}(\langle 1,0\rangle)\rangle/\langle 1,0\rangle],[\Im\cup\{\langle 1,0\rangle\}]\rangle$
$= \langle[\langle\mathcal{D}(\langle 1,0\rangle),\mathcal{U}(\langle 1,0\rangle),\{a\}-\{a\}\cup\phi/\langle 1,0\rangle],[\{\langle 6,0\rangle,\langle 4,0\rangle,\langle 3,0\rangle,\langle 2,0\rangle\}\cup\{\langle 1,0\rangle\}]\rangle$
$= \langle[\langle\mathcal{D}(\langle 2,0\rangle),\mathcal{U}(\langle 2,0\rangle),\phi/\langle 2,0\rangle],[\{\langle 6,0\rangle,\langle 4,0\rangle,\langle 3,0\rangle,\langle 2,0\rangle,\langle 1,0\rangle\}]\rangle$

Figure 3.4: Work of computing $\mathcal{R}$ and program slices

Time stamp	Expression
$\langle 1,0\rangle$	a = [5, 4, 3];
$\langle 2,0\rangle$	c = 6 :: a;
$\langle 3,0\rangle$	d = [a, c];
$\langle 4,0\rangle$	e = tl d;
$\langle 5,0\rangle$	f = tl c;
$\langle 6,0\rangle$	b = hd e;

(a) an example program

Time Stamp	Expression	$\mathcal{D}$	$\mathcal{U}$	$\mathcal{R}$
$\langle 1,0\rangle$	a = [5, 4, 3];	$\{a\}$	ϕ	ϕ
$\langle 2,0\rangle$	c = 6 :: a;	$\{c\}$	$\{a\}$	ϕ
$\langle 3,0\rangle$	d = [a, c];	$\{d\}$	$\{a,c\}$	ϕ
$\langle 4,0\rangle$	e = tl d;	$\{e\}$	$\{d\}$	ϕ
$\langle 5,0\rangle$	f = tl c;	$\{f\}$	$\{c\}$	ϕ
$\langle 6,0\rangle$	b = hd e;	$\{b\}$	$\{e\}$	ϕ

(b) Computing $\mathcal{D}$,$\mathcal{U}$ of an example program.

Time Stamp	Expression	$\mathcal{D}$	$\mathcal{U}$	$\mathcal{R}$	$\Im$
$\langle 1,0\rangle$	a = [5, 4, 3];	$\{a\}$	ϕ	ϕ	$\{\langle 6,0\rangle, \langle 4,0\rangle, \langle 3,0\rangle, \langle 2,0\rangle, \langle 1,0\rangle\}$
$\langle 2,0\rangle$	c = 6 :: a;	$\{c\}$	$\{a\}$	$\{a\}$	$\{\langle 6,0\rangle, \langle 4,0\rangle, \langle 3,0\rangle, \langle 2,0\rangle\}$
$\langle 3,0\rangle$	d = [a, c];	$\{d\}$	$\{a,c\}$	$\{a,c\}$	$\{\langle 6,0\rangle, \langle 4,0\rangle, \langle 3,0\rangle\}$
$\langle 4,0\rangle$	e = tl d;	$\{e\}$	$\{d\}$	$\{d\}$	$\{\langle 6,0\rangle, \langle 4,0\rangle\}$
$\langle 5,0\rangle$	f = tl c;	$\{f\}$	$\{c\}$	$\{e\}$	$\{\langle 6,0\rangle\}$
$\langle 6,0\rangle$	b = hd e;	$\{b\}$	$\{e\}$	$\{e\}$	$\{\langle 6,0\rangle\}$

(c) Computing $\mathcal{R}$ of an example program.

Time stamp	Expression
$\langle 1,0\rangle$	a = [5, 4, 3];
$\langle 2,0\rangle$	c = 6 :: a;
$\langle 3,0\rangle$	d = [a, c];
$\langle 4,0\rangle$	e = tl d;
$\langle 6,0\rangle$	b = hd e;

(d) Program slice with respect to $\langle\langle 6,0\rangle, \{e\}\rangle$.

Figure 3.5: An example of program slicing

$$\begin{array}{rcl}
\varepsilon_i[\![\theta : f(e_1,\ldots,e_n)]\!]\ ibve & = & (\varepsilon_i[\![\theta_n : e_n]\!](\ldots(\varepsilon_i[\![\theta_1 : e_1]\!]ibve)\ldots))[\langle\mathcal{D}(\theta),\mathcal{U}(\theta)\cup\{f\},\phi\rangle/\theta] \\
\varepsilon_i[\![\theta : \text{fun } f_j(x_1,\ldots,x_n) = e_j]\!]\ ibve & = & \varepsilon_i[\![\theta_j : e_j]\!]\ ibve
\end{array}$$

Figure 3.6: Semantic function of computing $\mathcal{D}$ and $\mathcal{U}$

3.3 Interprocedural Analysis

In this section, we will deal with interprocedural program. And we assume call-by-value parameter passing. So we need some new semantic functions for handling function calls and function definitions. For a function call $f(e_1,\ldots,e_n)$, we say that the expression use function variable f and the variables $\{x|x \in e_1 \mid \ldots \mid e_n\}$. For a function definition, we compute $\mathcal{D}$ and $\mathcal{U}$ of the expressions in the function body. So We add two semantic function to figure 3.1. we show the semantic function in figure 3.6.

we need new semantic function to compute relevant set,too. Because a function call update no variable, its relevant set inherit from its immediate successor. For a function definition, we must find the statements in function body that affect its return value. We define semantic functions of gathering the expressions that affects the return value in figure 3.7. We analyze the expressions in function body by these semantic functions. The return value of function is last expression in the function body. So the semantic functions is to find the last expression and gather backwardly the expressions that affect the return value. And we add two semantic function of computing $\mathcal{R}$ and $\Im$ to figure 3.2. We show them in figure 3.8.

We show an example in figure 3.9(a). Figure 3.9(b) is $\mathcal{D}$,$\mathcal{U}$ and $\mathcal{R}$ for each expression. The method is similar to the method described in section 3.2. We

Semantic Function

$$\varepsilon_f[\![\theta : exp]\!] \rightarrow Env \rightarrow Env$$

$$\begin{array}{rcl}
\varepsilon_f[\![\theta\text{: } c]\!]\ env & = & \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{U}(\theta)\rangle/\theta], [\Im \cup \{\theta\}]\rangle \\
 & & \text{where } \theta_1 = Inc(\theta) \\
\varepsilon_f[\![\theta\text{: } x]\!]\ env & = & \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{U}(\theta)\rangle/\theta], [\Im \cup \{\theta\}]\rangle \\
 & & \text{where } \theta_1 = Inc(\theta) \\
\varepsilon_f[\![\theta\text{: } !e]\!]\ env & = & \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{U}(\theta)\rangle/\theta], [\Im \cup \{\theta\}]\rangle \\
 & & \text{where } \theta_1 = Inc(\theta) \\
\varepsilon_f[\![\theta\text{: } e_1 \text{ op } e_2]\!]\ env & = & \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{U}(\theta)\rangle/\theta], [\Im \cup \{\theta\}]\rangle \\
 & & \text{where } \theta_1 = Inc(\theta) \\
\varepsilon_f[\![\theta\text{: } x = e_0]\!]\ env & = & \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{U}(\theta)\rangle/\theta], [\Im \cup \{\theta\}]\rangle \\
 & & \text{where } \theta_1 = Inc(\theta) \\
\varepsilon_f[\![\theta\text{: } e_1 :: e_2]\!]\ env & = & \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{U}(\theta)\rangle/\theta], [\Im \cup \{\theta\}]\rangle \\
 & & \text{where } \theta_1 = Inc(\theta) \\
\varepsilon_f[\![\theta\text{: hd } e]\!]\ env & = & \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{U}(\theta)\rangle/\theta], [\Im \cup \{\theta\}]\rangle \\
 & & \text{where } \theta_1 = Inc(\theta) \\
\varepsilon_f[\![\theta\text{: tl } e]\!]\ env & = & \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{U}(\theta)\rangle/\theta], [\Im \cup \{\theta\}]\rangle \\
 & & \text{where } \theta_1 = Inc(\theta) \\
\varepsilon_f[\![\theta \text{ :let } e_0 \text{ in } e_1]\!]\ env & = & \varepsilon_r[\![\theta_0 : e_0]\!](\varepsilon_f[\![\theta_1 : e_1]\!]env) \\
\varepsilon_f[\![\theta : \text{if } e_0 \text{ then } e_1 \text{ else } e_2]\!]\ env & = & \left\{\begin{array}{l}
(\varepsilon_f[\![\theta_1 : e_1]\!](\epsilon_f[\![\theta_2 : e_2]\!]env)) \\
[\langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{R}(\theta_1) \cup \mathcal{R}(\theta_2) \cup \mathcal{U}(\theta)\rangle/\theta], [\Im \cup \{\theta\}]\rangle] \\
\text{if} \quad \mathcal{D}(\theta_1) \cap \mathcal{R}(\theta_3) \neq \phi \quad \text{or} \quad \mathcal{D}(\theta_2) \cap \mathcal{R}(\theta_4) \neq \phi \\
\text{where}\theta_3 = Inc(\theta_1), \theta_4 = Inc(\theta_2) \\
\\
(\varepsilon_f[\![\theta_1 : e_1]\!](\epsilon_f[\![\theta_2 : e_2]\!]env)) \\
[\langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{R}(\theta_1) \cup \mathcal{R}(\theta_2)\rangle/\theta], \Im\rangle] \\
\text{otherwise} \\
\text{where}\theta_3 = Inc(\theta_1), \theta_4 = Inc(\theta_2)
\end{array}\right.
\end{array}$$

Figure 3.7: The semantic function of computing $\mathcal{R}$ for function definition

pay attention to the expression with time stamp $\langle 5, 0\rangle$. There is a function call in the assignment. We add the function variable to use set. In second phase, we pay attention to function definition f_1 and f_2. Because function f_2 isn't called, we don't analyze the function. Function f_1 is called, so we gather the expressions in the function body that affect the return value.

$$
\begin{array}{rcl}
\varepsilon_r[\![\theta : f(e_1,\ldots,e_n)]\!]\ env & = & \langle[\langle\mathcal{D}(\theta),\mathcal{U}(\theta),\mathcal{R}(\theta_1)\rangle/\theta],\Im\rangle \\
 & & \text{where } \theta_1 = Inc(\theta) \\
\varepsilon_r[\![\theta : \text{fun } f_j(x_1,\ldots,x_n) = e_j]\!]\ env & = & \varepsilon_f[\![\theta_j : e_j]\!]env \text{ if } f_j \in \mathcal{R}(\theta_{j+1})
\end{array}
$$

Figure 3.8: Semantic function of computing $\mathcal{R}$

Time stamp	Expression
$\langle 1,0\rangle$	fun f1(a,b)=hd a :: hd b;
$\langle 2,0\rangle$	fun f2(c,d)=tl a :: tl b;
$\langle 3,0\rangle$	e=[1,2,3];
$\langle 4,0\rangle$	f=[4,5,6];
$\langle 5,0\rangle$	g=f1(e,f);
$\langle 6,0\rangle$	h=g::tl f;

(a) an interprocedural progam

Time stamp	Expression	$\mathcal{D}$	$\mathcal{U}$	$\mathcal{R}$	$\Im$
$\langle 1,0\rangle$	fun f1(a,b)=hd a :: hd b;	ϕ	{ a,b }	$\{a,b\}$	$\{\langle 6,0\rangle,\langle 5,0\rangle,\langle 4,0\rangle,$ $\langle 3,0\rangle,\langle 1,0\rangle\}$
$\langle 2,0\rangle$	fun f2(c,d)=tl c :: tl d;	ϕ	{ c,d }	$\{f1\}$	$\{\langle 6,0\rangle,\langle 5,0\rangle,\langle 4,0\rangle,\langle 3,0\rangle\}$
$\langle 3,0\rangle$	e=[1,2,3];	{ e }	ϕ	$\{f1\}$	$\{\langle 6,0\rangle,\langle 5,0\rangle,\langle 4,0\rangle,\langle 3,0\rangle\}$
$\langle 4,0\rangle$	f=[4,5,6];	{ f}	ϕ	$\{f1,e\}$	$\{\langle 6,0\rangle,\langle 5,0\rangle,\langle 4,0\rangle\}$
$\langle 5,0\rangle$	g=f1(e,f);	{ g }	{ e,f,f1}	$\{f1,e,f\}$	$\{\langle 6,0\rangle,\langle 5,0\rangle\}$
$\langle 6,0\rangle$	h=g::tl f;	$\{h\}$	{ g,f}	$\{g\}$	$\{\langle 6,0\rangle\}$

(b) After computing $\mathcal{D}$,$\mathcal{U}$ and $\mathcal{R}$

Time stamp	Expression
$\langle 1,0\rangle$	fun f1(a,b)=hd a :: hd b;
$\langle 3,0\rangle$	e=[1,2,3];
$\langle 4,0\rangle$	f=[4,5,6];
$\langle 5,0\rangle$	g=f1(e,f);
$\langle 6,0\rangle$	h=g::tl f;

Figure 3.9: Interprocedural analysis example program

Chapter 4

Implementation and Experiment

This chapter presents the layout of our implementation and experimental results of the algorithms that are proposed in previous chapter. Our implementation of program slicing analysis is in a Red Hat Linux (release 6.1, Cartman) and CLE (version 0.9, Yami) environment. The C compiler we use is GCC version 2.96. In our implementation, the constant propagation is performed on *C* code which follows the requirements for ANSI C.

Our Program slicing analysis have following phase.

1. The parser of our program analyzer is generated by Flex and Bison that.
2. In semantic analysis phase, we compute the definiton set and use set. It gather the information which variables are updated or used for each expression.
3. In program slicing analysis phase, we compute the relevant set and con-

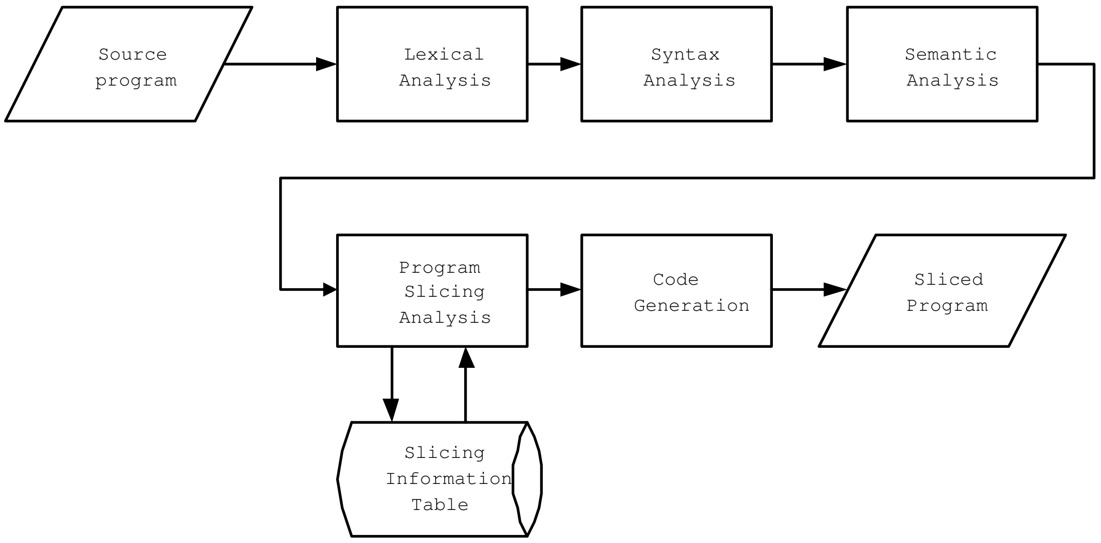

Figure 4.1: The implement layout of program slicing analysis

struct sliced program. The relevant set is the information which variables affect the variables interest us. And we construct sliced program according to the information of relevant set.

4.1 Slicing information table

Slicing information table is the data structures that we construct during the process of program slicing analysis. It keep the information that we need for compute program slice. There are slicing information for each expression in the table. It is a hash table. Our approach is to analyze program by these infromatin. We show it as follows.

Slicing information	table structure
Time stamp	the time stamp of expression
Define set	the variables that defined in the expression
Use set	the variables that used in the expression
Relevant set	the variables that affect the variables interest us in the expression

The following are basic operations on slicing information table.

- **Table lookup:** we use the time stamp to lookup slicing information table.
- **Table update:** we update the table when we find the variables that is defined, used,or affect variables interst us in a expression.

4.2 Experiment

Here we show the call graph of our implementation in figure 4.2. We describe the functions in the figure as follows.

1. **Program Slicing:** The function is main part of our approach. It compute definition and use set for each expression by calling function **DUAnal**. Then we compute relevant set for each expression and gather the expressions that affect the observed variables by calling the function **RelAnal**.
2. **DUAnal:** The function is to compute the definition and use set for each expression. And store these information in slicing table by calling

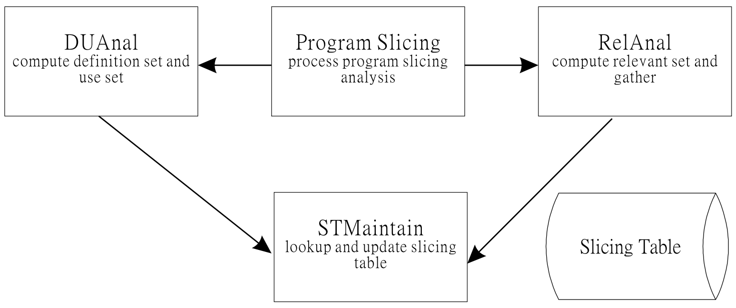
DUAnal
compute definition set and use set
Program Slicing
process program slicing analysis
RelAnal
compute relevant set and gather
STMaintain
lookup and update slicing table
Slicing Table

Figure 4.2: Call graph of program slicing analysis

function **STMaintain**. We show the flowchar in figure 4.3.

3. **RelAnal:** The function is to compute relevant set for each expression and gather the expressions that affect the observed variables. And we keep these information in the slicing table. We show the flowchar in figuare 4.4.

4. **STMaintain:** The function is to maintain slicing table. We access the slicing information tables by the function. It is use to store or load the data that we need.

We show some experiment result as follows. The program **Maxmin** is to find the maximum and maximum. There are 30 token in original program. After analyzing by our approach, there are 24 tokens in the sliced program. The program **GCD** is to compute greatest common factor. There are 55

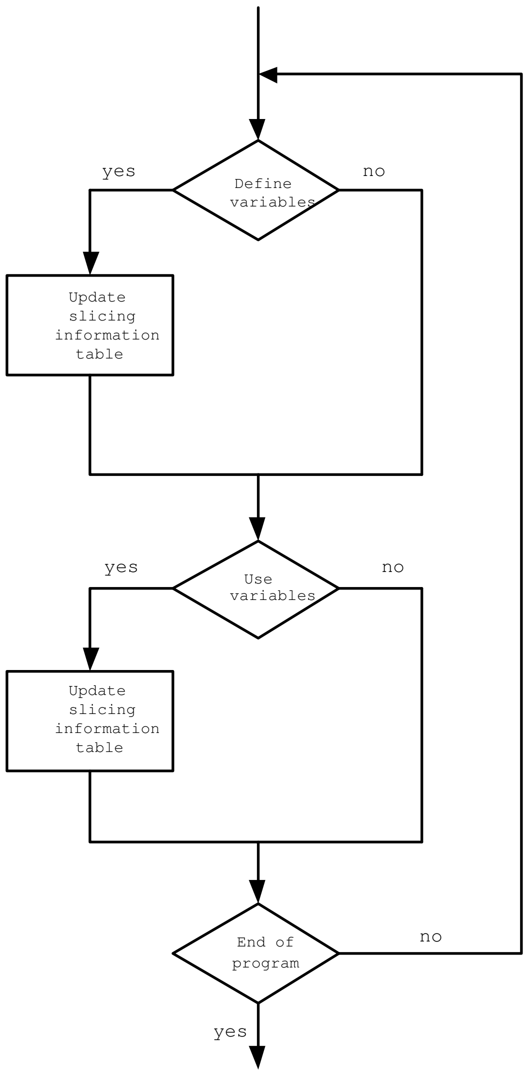
yes
Define variables
no
Update slicing information table
yes
Use variables
no
Update slicing information table
End of program
no
yes

Figure 4.3: Flowchar for computing Slicing Information table in Semantic analysis

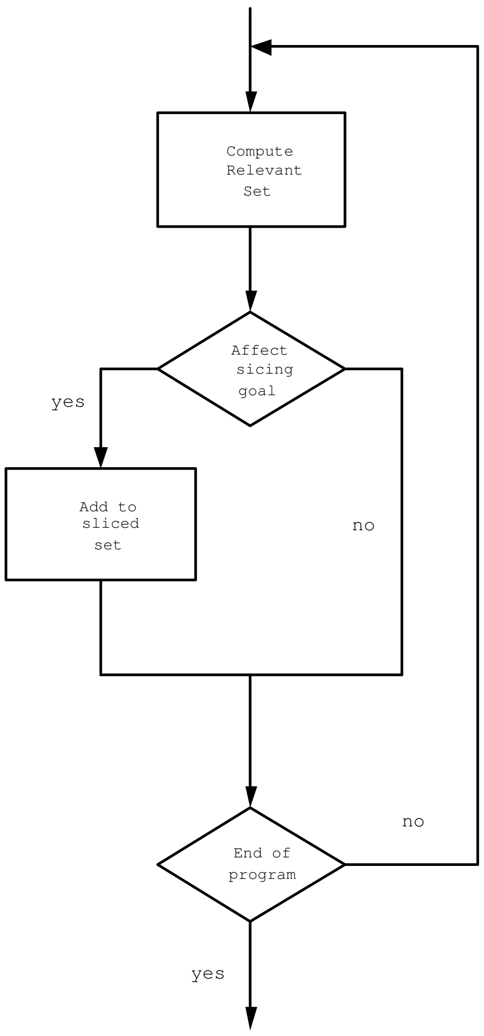

Figure 4.4: Flowchar for program slicing

token in original program. After analyzing by our approach, there are 52 tokens in the sliced program. The results show that we can correctly gather the expressions that affect the observed variables.

	♯ of tokens.	
Program name	Original program	Slicd Program
Maxmin	30	24
Factorial	33	27
DaytoTime	48	45
GCD	55	52

Chapter 5

Related Work

In the chapter, we introduce related work about program slicing. We will introduce program slicing based on data-flow equation and dependence graph.

5.1 Program slicing based on data-flow equation

The program slicing is a approach of gathering the statements that affect the variables v at statement s. Typically, we give a pair $\langle n, v \rangle$ called *slicing criterion.* The concept of program slicing may trace back to the notion introduced by Weiser [WM79, WM81, WM84]. They compute program slice using *control flow graph*(CFG) as an intermediate representation. A nodes of CFG consist of each statement and control predicate in the program. An edge from node i to node j denote the possible flow of control from the former to the latter[BG96, Tip95]. We give an example program in figure 5.1. And CFG of the program is showed in figure 5.2. The set $\mathcal{D}(i)$ and $\mathcal{U}(i)$ indicate the sets of variables defined and referenced at node i of CFG, respectively.

Source Program
read(a);
read(b);
if a < b then
min := a;
max := b;
else
min := b;
max := a;
end if
write(min);
write(max);

Figure 5.1: The example program

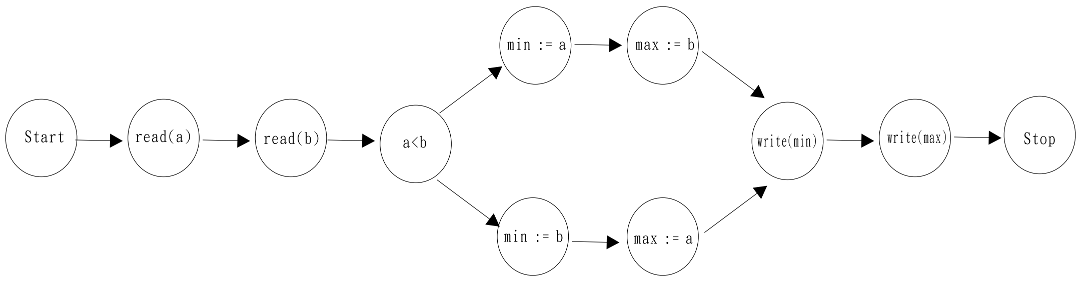

Figure 5.2: The control flow graph of source program

They compute *relevant set* denoted by $\mathcal{R}(i)$ and then they use relevant set to extract the program slice. For the program slice with respect to $\langle n, v\rangle$, the relevant set consist of the variables whose values affect the computation of v at s. The relevant set consist of the variables that directly or indirectly affect the variables interesting us. For a segment of program: $a = b + 1; c = a + 2;$, we say that the variable a directly affects the variable c and the variable b indirectly affect the variable c. The relevant set for the program slice with respect to $\langle n, v\rangle$ are computed as follows.

1. $\mathcal{R}(i) = v$ when $i = n$
2. $\mathcal{R}(i) = (\mathcal{R}(j) - \mathcal{D}(i)) \bigcup (\mathcal{U}(i)$ if $\mathcal{R}(j) \bigcap \mathcal{D}(i) \neq \phi)$, where node j is the immediate successor of node i.

In a structured program, a statement may have multiple control predecessors. We need some modifications to handle the cases that a statement has more than one predecessor in the control flow. First, we compute the control set for each statement i, which is a set of statements that directly control the execution of statement i. Second, we compute the relevant set at each join point i, where the two nodes j_1 and j_2 have the same predecessor. The relevant set of i is the union of the relevant sets of j_1 and j_2. Once these two sets are computed, we may find the program slice. When a statement is added to the slice, all the members in its control set are also added to the slice.

Figure 5.3 shows the example of a program slicing criterion $\langle 10, \{max\}\rangle$. For computing the program slice, we start from line 10. $\mathcal{R}(10) = \{max\}$. Because there is no definition in line 9, the relevant set of node 9 is relevant

$Node$	statement	$\mathcal{D}(i)$	$\mathcal{U}(i)$	$Controlset$	$\mathcal{R}(i)$
1	read(a)	{ a }			
2	read(b)	{ b }			{ a }
3	if a < b then		{ a,b }		{ a,b }
4	min := a	{ min }	{ a }	3	{ b }
5	max := b	{ max }	{ b }	3	{ b }
6	else				
7	min := b	{ min }	{ b }	6	{ a }
8	max := a	{ max }	{ a }	6	{ a }
	end if				
9	write(min)		{ min }		{ max }
10	write(max)		{ max }		{ max }

(a) The source program

$Node$	statement
1	read(a)
2	read(b)
3	if a < b then
5	max := b
6	else
8	max := a
	end if
10	write(max)

(b) The program slice with a slicing criterion $\langle 10, \{max\} \rangle$

Figure 5.3: An example of program slicing with slicing criterion $\langle 10, \{max\} \rangle$

set of node 10. $\mathcal{R}(9) = \mathcal{R}(10) = \{max\}$. Because node 3 has two immediate successor,the one's relevant set is union of relevant set of two immediate successor. $\mathcal{R}(3) = \mathcal{R}(4) \cup \mathcal{R}(7) = \{b\} \cup \{a\} = \{a, b\}$. In a same way, we can compute all relevant sets. A statement i is add to the slice if $\mathcal{D}(i) \bigcap \mathcal{R}(j) \neq \phi$ where j is the immediate successor of i, or i is a member in a control set.

5.2 program slicing based on dependence graphs

Another approach to compute program slice is based on dependence graphs[KKPLW81, HRB90, RY88]. Both of these algorithms have three steps:

1. Construct a dependence graph from the program.
2. Slice the dependence graph.
3. Obtain a sliced program from the sliced graph.

The program dependence graph(PDG) is a directed graph whose vertices are connected by several kinds of edges. The vertices of PDG consist of each assignment statement and control predicates in program. In addition, PDG includes three other class of vertices[HRB90, **?**]:

1. There is a specific vertices called the *entry vertex.*
2. For each variable x which is used before being defined, there is a vertex called the *initial definition of x.* This vertex represents an assignment to x from the initial state.
3. For each variable x used in the program, there is a vertex called the *final use of x.* This vertex represents an access to the final value of x, and is labeled "*FinalUse(x).*"

The edges of PDG represent either a control dependence or data dependence. The control dependence edge is labeled either true or false. The source of control dependence edge is always the entry vertex or a predicate vertex. The control dependence edge from the vertex v_1 to the vertex v_2, denoted by $v_1 \longrightarrow_c v_2$, mean that the execution of the vertex predicate on if the evaluation of the vertex equal to label of the control dependence edge. There is a method of determining control dependence edge in [FOW87]. PDG contains a control dependence edge from v_1 to v_2 iff on of the following holds[HRB90]:

1. v_1 is the entry vertex and v_2 represents a component of program that is not nested within any loop of conditional; these edges are labeled true.

2. v_1 is a control predicate and v_2 is a component of program immediately nested with in the loop of conditional whose predicate is represented by v_1. If v_1 is the predicate of a while-loop, the edge $v_1 \longrightarrow_c v_2$ is labeled true; if v_1 is the predicate of a conditional statement, the edge $v_1 \longrightarrow_c v_2$ is labeled true or false according to if v_2 occurs in the then branch or the else branch, respectively.

The data dependence edge from the vertex v_1 to the vertex v_2 mean that result of program computation is changed if the relative order of v_1 and v_2 is reversed. PDG contain *flow dependence edge* and *def-order dependence edge.* PDG contain a flow dependence edge from v_1 to v_2 iff all of the following hold[HRB90]:

1. v_1 is a vertex that define variable x.

2. v_2 is a vertex that uses x.

3. There is no intervening definition of x in the execution path from v_1 to v_2.

The flow dependence can be classified as *loop carried* and *loop independent.* The flow dependence is a loop carried dependence if in addition to 1,2 and 3 above,the following also hold[HRB90]:

1. There is an execution path that both satisfies the conditions of 3 above and include a backedge to the predicate of loop.

```
procedure MarkVerticesOfSlice(G,S)
declare
  G: a program dependence graph
  S: a set of vertices in G
  WorkList: a set of vertices in G
  $v, w$:vertices in G
begin
  WorkList := S
  while WorkList $\neq \phi$ do
    Select and remove vertex from WorkList
    Mark $v$
    for each unmarked vertex $w$ such that edge $w \longrightarrow_f v$ or edge $w \longrightarrow_c v$ is in $E(G)$ do
      Insert $w$ to WorkList
    od
  od
end
```

Figure 5.4: An algorithm of program slicing based on dependence graph

2. Both v_1 and v_2 enclosed in loop.

A flow dependence is a loop-independent if in addition to 1,2 and 3 above,there is an execution path that satisfied 3 above and includes no backedge to the predicate of a loop.

PDG contain a def-order dependence edge from v_1 to v_2 iff all of the following hold[HRB90]:

1. v_1 and v_2 both define the same variable.

2. v_1 and v_2 are in the same branch of any conditional statement that encloses both of them.

3. There exists a program component v_3 such that $v_1 \longrightarrow_f v_3$ and $v_2 \longrightarrow_f v_3$

4. v_1 occurs to the left of v_2 in the program's abstract syntax tree.

After constructing PDG, they propose an algorithm of program slicing based on dependence graph showed in figure 5.4. There are the details of slicing based on dependence graph in [BG96, HRB90, RY88, Tip95].

Chapter 6

Conclusions

Program slicing is a technique of gathering the expression that affect specific variables at a specific point in a program. It can be used to help programmer understands complex code, debug and so on. Wiser define program slicing as dataflow equation problem on control flow graph[WM79, WM81, WM84]. Then Ottenstein K. and Ottenstein L. provide a method based on dependence graph [OO84, RY88]. Their methods needs the graph representation of a program. Instead of using dependence graph, we propose a semantic method of program slicing using abstract interpretation in the thesis. Our approach analyzes a program based on the define-use attributes and the indirectly relevant attributes of the values in the program. We can compute program slice without graph representation of program. We show the semantic functions and prove it in chapter 3. And we show our Implementation in chapter 4. The result of experiments show that we can correctly compute program slice.

Bibliography

[AH87] S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Language,* Ellis-Horwood, 1987.

[ASU86] Aho, Sethi and Ullman, "Compilers: Principles, Techniques, and Tools,"*Addison-Wesley Press,* 1986.

[BG96] B. W. Binkley, K. B. Gallagther, "Programing Slicing," *Advances of Computing*, vol.43, pp.1-50, 1996.

[Blo89] A. Bloss, "Update Analysis and the Efficient Implementation of Functional Aggregates,"*Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pp. 26-38, ACM, 1989.

[Cou96] P. Cousot, "Abstract Interpretation," *ACM Computing Surveys (CSUR)*, Vol. 28, No.2, pp. 324-328, 1996.

[DIKU95] M. R. MIKU, "Introduction to Abstract Interpretation," Computer Science Uninversity of Copenhagen,1995.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in Optionization ," *ACM Transactions on Programming Languages and System*, Vol. 9, NO. 3, PP.319-349,1987.

[HAU89] P. A. Hausler, "Denotational Program Slicing," *Proceedings of the Twenty-Second Annual Hawaii International Conference*, Vol. 2, pp. 489-494, 1989.

[HRB90] S. Horwitz, T. Reps, and D. Binkley,"Interprocedual Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, Vol.12, No.1, pp.26-61, 1990.

[KKPLW81] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wlofe, "Dependence graphs and complier optimizations," *In Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pp.207-218, 1981.

[KS] B. L. Kurtz and K. Slonneger, "Denotational Semantics," *Formal Syntax and Semantics of Programming Languages*, Chapter 9, pp.271-340.

[Lyl84] J.R.Lyle, "Evaluationing Variations of Program Slicing for Debugging,"*PhD thesis*,University of Maryland,College Park,Maryland,December 1984.

[OO84] K. Ottenstein and L. Ottenstein, "The program dependence graph in a software development environment", *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering sym-*

posium on Practical software development environments, pp.177-184,1984.

[Pit97] A. M. Pitts, "Denotational Semantics," *Lecture Notes on Denotational Semantics for Part* II *of Computer Secience Tripos*, Cambridge University Computer Laboratory, 1997.

[RM89] C. Robert, F. Matthias, "The semantics of program dependence," *ACM SIGPLAN Notices , Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*,Vol. 24, NO. 7,Portland, Oregon, United States,1989.

[RY88] T. Reps and W. Yang, "The semantics of program slicing" *Technical Report 777*, University of Wisconsin - Madison, June 1988

[Tip95] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Language*, Vol.3, No.3, pp.121-189, 1995.

[TJ95] T.Johnsson, "Lambda Lifting," *Proceedings on Aspenas Workshop on Implementations of Functional Language*, Chalmers Institute,Gotebord,sweden,Jan 1985.

[WC98] M. Wand and W. D. Clinger, "Set constraints for destructive array update optimization," *Proceedings of IEEE Conference on Computer Languages*, pp. 184-193, 1998.

[WM79] M. Weiser, "Program slicing:Formal,Psychological and Practical Investigation of anAutomatic Program Abstraction

Method," *PhD thesis,*The universit of Michigan, ANN Arbor,Michigan,1979.

[WM81] M. Weiser, "Program Slicing," *Proceeding of the Fifth International Conference on Software Engineering*, pp.439-449, May 1981.

[WM84] M. Weiser, "Program Slicing".*IEEE Transaction of Software Engineering*, July 1984.

[CY97] C. Yung, "Extending Lambda-Calculus to Sets," *Proceedings on 1997 MASPLAS (in cooperation with ACM SIGPLAN)*, East Stroudsburg University, Pennsylvania, April 1997.

[CY98] C. Yung, "EAS: An Experimental Applicative Language With Sets," *Proceedings on 1998 MASPLAS (in cooperation with ACM SIGPLAN)*, Rutgers University, New Jersey, April 1998.

[CY99] C. Yung, "Destructive Effect Analysis and Finite Differencing for Strict Functional Languages", *Ph.D. dissertation*, Computer Science Department, New York University, August 1999.

[CY00] C. Yung, "Dynamic Copy Elimination For Strict Functional Languages," *roceedings on the Sixth Workshop on Compiler Techniques for High-Performance Computing* , pp. 206-216, National Sun Yat-sen University, Kaohsiung, Taiwan, March 2000.

[CY01] C. Yung, "A Last-Use Analysis for Pure Functional Programs," *Proceedings on the Seventh Workshop on Compiler Techniques* *for High-Performance Computing* , pp. 136-142, National Chiao Tung University, Hsinchu, Taiwan, March 2001.

[CY03] C.Yung, "The Iuvestigation and Development of a Compiler for a Programming Language using Sets,"*National Science Council Project Report*, to appear on August 2003.