國立東華大學資訊工程學系
碩士論文

植基於程式語言語意之跨程序常數傳遞分析

A Semantic Method of Interprocedural Constant Propagation Analysis

研究生：余昀龍

指導教授：雍　忠 博士

中華民國九十三年七月

國立東華大學學位論文授權書

本授權書所授權之論文為本人在國立東華大學 資訊工程 系所組 九十二 學年度第 二 學期取得 碩士學位之論文。

論 文 名 稱：植基於程式語言語意之跨程序常數傳遞分析

指導教授姓名：雍忠教授

學 生 姓 名：余昀龍

學　　號：69121042

本人具有著作財產權之上列論文全文資料，基於資源共享理念、回饋社會與學術研究之目的，非專屬、無償授權國立東華大學、國家圖書館及行政院國家科學委員會科學技術資料中心，得不限地域、時間與次數，以微縮、光碟或數位化等各種方式重製散布、發行或上載網路，提供讀者非營利性質之線上檢索、閱覽、下載或列印。

上述數位化公開方式限：

■校內、校外公開。

□ 校內公開，校外因上列論文申請專利（案號：＿＿＿＿＿＿），請於 3 年後公開。

□ 校內公開，校外因＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿，請於 1 年後公開。

授權內容均無須訂立讓與及授權契約書，授權之發行權為非專屬性發行權利。依本授權所為之收錄、重製、發行及學術研發利用均為無償。**數位化公開方式若未勾選，本人同意視同授權校內、校外公開**。

研究生簽名
(親筆正楷)

日期：中華民國　九十三 年 七 月 三十 日

說明：本授權書請以黑筆撰寫並裝訂於紙本論文書名頁之次頁。

A Semantic Method of Interprocedural Constant Propagation Analysis

by

Yun-Lung Yu

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 2004

Approved: _____

Chung Yung

To my parents, Tsung-Fu Yu and Chi-Ping Pai

Acknowledgements

Two years ago, I made efforts in finding to correlate with topics of the work. I have been accompanied and supported the thesis by many people. They provide not only help in research but also encouragements in my life. I would like to express my appreciation to all those who gave me the support and cheer that gave me the power to complete the thesis. Dr. Chung Yung is the first person I would like to thank who is my direct adviser. He provided a motivating, and useful ideas during the many discussions we had. I am grateful to the thesis committee member, Dr. Tyng-Ruey Chuang and Dr. Hsin-Chou Chi, for their valuable suggestions on the thesis.

Finally, I would like to thank our laboratory members, they give me a hand during the development of the thesis. They are: Per-Jen Chuang, Ming-Sian Lin, Feng-Wei Yang, You-Liang Sun, Bing-Liang Shih, and Liang-Chie Weng.

A Semantic Method of Interprocedural Constant Propagation Analysis

Yun-Lung Yu

Advisor: Chung Yung

Abstract

A compiling system attempts to improve code for a whole program by optimizing across procedures. A compiler can generate better code for a special procedure if it knows which variables will have constant values, and what those values will be when the procedures are invoked[CCKT86]. Yung and Wang describe a constant propagation algorithm based on FUD chains of imperative programs, but they do not detect the interprocedure constants[YW02]. In this thesis, we propose an approach to solve this problem on the PLUS language.

Instead of using control flow and data flow graph, we propose a semantic method of constant propagation using abstract interpretation. We present our model with a series of semantic methods. We prove the correctness of our methods by a structural induction. We perform some experiments on the efficiency and effectiveness of this new approach.

植基於程式語言語意之跨程序常數傳遞分析

余昀龍

指導教授：雍　忠　教授

摘要

編譯系統企圖經由程序間的最佳化，對整個程式改善程式碼。如果我們可以知道哪些變數有常數值(constant value)，而且當程序呼叫時知道這些常數值具有何種值，編譯器可能對於一個特殊的程序產生更好的編碼方式。Yung 和 Wang 在命令式語言中，描述一種使用 FUD 鏈的常數傳遞(constant propagation)演算法，但是他們並沒有去處理程序間常數傳遞分析。在這篇論文，我們提出一個方法去解決 PLUS 程式語言中在這方面的問題。

我們提出使用 abstract interpretation 的常數傳遞的語義分析方法來取代使用控制和資料流程圖的方法。我們用一系列的語義方法描述我們的模型。我們將使用結構性歸納法去證明我們方法的正確性。我們對於這個新的方法作一些實驗，產生在效率和效用上的結果。

Contents

List of Figures

Chapter 1

Introduction

Constant propagation analysis is a well-known global flow analysis problem. It is an optimization based on data-flow analysis, which is a static analysis of providing global information about how a program works to manipulate its data[WZ91]. While the constant propagation problem is easily shown to be undecidable in general[KU77], there are many reasonable instances of a program that are decidable and for which computationally efficient algorithms exist.

A few techniques[Kil73, SGW93, YW02, WZ91] have been successful applied to the analysis of imperative language for the optimization. But there are few related and specific research with functional language for constant propagation. Though their methods above also claim that their method can apply to functional language, we discover that they are not efficient applying to it on some situations.

For example, within Wegman and Zadeck's SCC algorithm based on SSA

form is used to find constant expression, constant conditions, and unreachable code[WZ91]. Wolf proposed a simple constant propagation based on FUD chains, and presented an idea of how to perform a demand-driven analysis[Wol96]. In order to detect more conditional constants and loop constants, they also offer a version of GSA based demand-driven symbolic analysis. Yung and Wang propose an approach of demand driven constant propagation analysis based on FUD chains[YW02]. They combine the idea from Wegman and Zadeck's SCC algorithm of finding conditional constants and the idea from the method by Wolfe of finding linear induction variable with loops. Their approach may classify cycles in the data flow graph and take predicate of branch node into account. By the way, they can determine which branch will be taken to reach the join point. The above methods all compute constant propagation with graph representation of program. And the important information for constant propagation are definition chain and use chain for each expression. The graph representation of program help them to gather these information.

In the analysis of their methods, they generally used graph representation of control and data flow to solve it. First, they need a control flow graph to represent probable steps of execution of whole program. And they compute Definition-Use chain (or provide with SSA form) based on the control flow graph. Finally, they propagate constants iteratively based on data flow and control flow graph. But It was provided with sequential feature in functional languages, so we will spend a lot of time in the analysis if we would utilize the graph representation to solve it.

In a compiler system of imperative language along with interprocedure optimization, it attempts to improve code for a whole program by optimizing across procedures, the compiler can generate better code for a specific procedure if it knows which variables will have constant values, and what those values will be, when the procedure is invoked[CCKT86]. The analysis tries to eliminate the boundary of each procedure call to propagate constants as far as possible. In functional language, we know that one function always returns one result value. We take care of the return value from one function. In general, there is one variable binding for the return value which the function body computes. If the return value is a constant, we can immediately gather the constant value by the function environment computation. In this way, we can gather results in compile-time and decrease the rum-time cost.

A great deal of progress in compiler construction has been made in the last few years towards the development of a theoretical framework appropriate to perform analysis with the semantical methods of computer languages. Despite the complexity and variety exhibited by modern programming languages, it has been shown by Scott, Strachey and their colleagues at Oxford University that a remarkably small number of fundamental semantic constructs provide an adequate conceptual basis for defining concise formal models of their meaning[Ten76]. With formal semantics we give program meaning by mapping them into some abstract but precise domain of objects. Using denotational semantics, we provide the meaing in term of mathematical objects, such as integers, boolean value, values of tuples, and functions. For the reason,

denotational semantics was originally called mathematical semantics[KS]. It becomes a powerful tool for language design and implementation.

Most analysis utilize pseudo code to represent the algorithm of analysis. We make use of interpretation semantics which is similar to denotation semantics with a good theoretical framework to implement our idea. And we use interpretation semantics to abstract information of constant propagation . We show the semantics of a sample language, then we develop a semantic method that employs an environment to store our possible constant value of each variable and current environment.

1.1 Motivation

Constant propagation analysis is a well-known global flow analysis problem. The goal of constant propagation is to discover values that are constants on all possible executions of a program and to propagate these constant values as far forward through the program as possible.

Most flow analysis techniques are developed as algorithms in which implementation details are combined with the technique itself. The design process is complicated by the need to handle low level of implementation details. The design of flow analysis techniques is heavily dependent on the application. Low level development of an analysis technique makes it difficult to prevent duplication of effort in developing techniques that are closely related. The presence of implementation details make it harder to understand the algorithms in order to maintain or improve existing ones. Finally, correctness of proofs is difficult

to derive for algorithms with too much detail.

With tranditional methods, they use control flow graph for a program when representing the control flow of it. Then they compute data dependencies based on the control flow graph. Finally, they will solve our problem of the objective analysis based on those graph representation.

We observe a fact that it is inefficient if we use graph representation on functional programs. With the denotational framework, we can use concept of environment store to solve the constant propagation problem and to help us instead of the graph representation of program. We can decrease time that building graph of blocks, nodes and edges.

Constant propagation is a code reduction technique that determines whether all assignments to a particular variable is a constant and provides the same constant value at certain points. In that case, a use of the variable at that point can be replaced by the constant. For example, given an assignment $x = c$ for a variable x and a constant c, we may replaces the later uses of x with c only if it does not change the value of x in the later assignments. Constant propagation can result in better instruction selection where you can use imediate machine code instructions which embed constants in the instruction, and thus result in faster execution in the case that a register is used to store the constant. Constants in code allow compilers to generate more efficient codes and hence increase the effectiveness of several other optimizations such as constant expression evaluation, and dead-code elimination. We give an example of a source program in Figure 1.1. Figure 1.2 shows the control

flow graph providing with FUD chain of the program. There are two phases in constructing factored use-def chains as follow:

1. Adding the ϕ-terms at control flow merge points for each variable with multiple reaching definitions, and

2. Linking each variable use to the unqiue reaching definition or ϕ-term.

We apply Wolfe's constant propagation algorithm, and the result is shown Figure 1.3.

In this thesis, We will develop a semantic method of interprocedural constant propagation for a functional language. We define semantic functions to compute constant values of those variables for each expression using abstract interpretation. We compute the constant propagation without graph representation of the program. And we can get optimized program based on these information.

Source Program

a = 5;
b = 3;
read (c);
if b < a
then
e = b;
d = a;
else
e = a+1;
d = c+1;
end if
f= e+d;

Figure 1.1: The Source Program

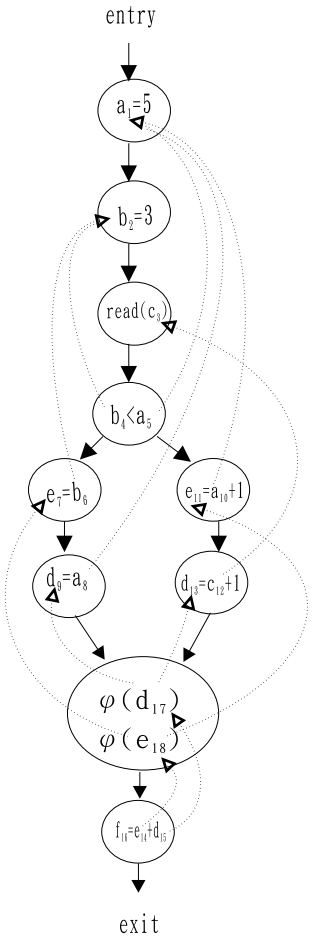

Figure 1.2: Control Flow Graph with FUD Chain

Node	FUD Chain solution	meet operation	Const
1	$a_1 = 5$	$C_{a_1} = \top \wedge C_5$	$\mathcal{C}$
2	$b_2 = 3$	$C_{b_2} = \top \wedge C_3$	$\mathcal{C}$
3	read(c_3)	$C_{c_3} = \bot$	
4	if $b_{4\rightsquigarrow 2} < a_{5\rightsquigarrow 1}$		True
5	then		
6	$e_7 = b_{6\rightsquigarrow 2}$	$C_{e_7} = \top \wedge C_{b_2}$	$\mathcal{C}$
7	$d_9 = a_{8\rightsquigarrow 1}$	$C_{d_9} = \top \wedge C_{a_1}$	$\mathcal{C}$
8	else		
9	$e_{11} = a_{10\rightsquigarrow 1}$	$C_{e_{11}} = \top$	
10	$d_{13} = c_{12\rightsquigarrow 3}$	$C_{d_{13}} = \top$	
11	end if		
12	$d_{17} \leftarrow \phi(d_9, d_{13})$	$C_{d_{17}} = \top \wedge C_{d_9}$	$\mathcal{C}$
13	$e_{18} \leftarrow \phi(e_7, e_{11})$	$C_{e_{18}} = \top \wedge C_{e_7}$	$\mathcal{C}$
14	$f_{16} = e_{14\rightsquigarrow 18} + d_{15\rightsquigarrow 17}$	$C_{f_{16}} = \top \wedge C_{d_{17}} \wedge C_{e_{18}}$	$\mathcal{C}$

(a) Analysis Process of The Source Program

Source Program
$a_1 = 5;$
$b_2 = 3;$
$e_7 = 3;$
$d_9 = 5;$
$f_{16} = 8;$

(b) A Result of Optimized Program

Figure 1.3: The Process of Optimized Program with Constant Propagation

1.2 Organization of the thesis

This thesis is organized as follows:

- In Chapter 2, we introduce the background knowledge and the description of our source program. We explain what the denotation semantics and lattice framework are, and what behavior can be represented by abstract interpretation. We introduce a lattice framework how to use it for constant propagation. Moreover, we will illustrate some characteristics in our sample language with functional properties.

- In Chapter 3, we describe the details of our approach and define the semantic functions of computing constant propagation. We will discuss dividing with two parts: the former is the formal semantics; the latter is the semantic-based analysis. We distinguish the difference between compile-time and run-time analysis.

- In Chapter 4, we show our experimentation environment and the implementation of our approach. Our experiments are based on two environments one of which is the function environment and the other is a bound variable environment. Then we also compare analysis time with the previous method which is Yung and Wangs'.

- In Chapter 5, we will provide an perspective of the related work about constant propagation. We will introduce three difference in the approach by Wegam and Zadeck, by Wolf, and by Yung and Wang.

- In Chapter 6, we will present conclude and future work.

Chapter 2

Background

In this chapter, we introduce the fundamental concepts for the techniques developed in the constant propagation analysis. Denotational semantics is a technique that maps the syntactic object of a program into program meaning. We will particularly explain about it later. 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 describe it in section 2.2. Our sample 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. Finally, we will introduce lattice framework. The output of a constant propagation algorithm is an output assignment if lattice values to variables at each expression in the program.

2.1 Denotational Semantics

The foundation of denotational semantics[KS] is, the identification that programs and the objects they operate are significant understanding of abstract mathematical objects. The idea of denotational semantics is to associate an appropriate mathematical object with each phrase of the language, for example, string of digits realize numbers. In the denotational semantics, there are 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. These rules can be defined using the syntactic categories or using the metavariables for elements of the categories as an abbreviation mechanism.
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 objects of the semantic domain.
   (b) *Semantic equations* to specify how the functions act on each pattern in the syntactic definition of the language.

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

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

Figure 2.1: An Example of Denotational Semantics

Figure 2.1 is an example of denotational semantics. It is an addition of integers. We show how to compute 30*40 as follows.

$$
\begin{aligned}
\varepsilon[\![3 * 40]\!] &= \varepsilon[\![3]\!] * \varepsilon[\![40]\!] \\
&= 3 * 40 \\
&= 120
\end{aligned}
$$

2.2 Abstract Interpretation

The framework of abstract interpretation provides the basis for a semantic approach to dataflow analysis. Abstract interpretation is a general theory and a semantics-based program analysis method, introduced by P. Cousot and R. Cousot, for describing the relationship among semantics of programming languages at different levels of abstraction. 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. However this is not necessarily the standard meaning, but it can be used to extract information about the computational behaviour of the program.

In this framework, program analysis is defined as nonstandard program semantics, obtained from the standard one by substituting its domain of computation, called concrete, (and the basic operations on it) with an abstract domain (and corresponding abstract operations). The concrete and the abstract domains are always complete lattices, where the ordering relations describe the relative precision of the denotations, the top elements representing no information. 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. This can be description as

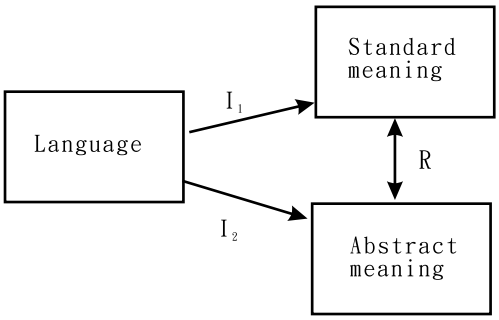

Figure 2.2: Abstract Interpretation Framework

It is an example of abstract interpretation at the back. It is the familiar rule-of-sign interpretation of integer expressions. In some situations we can be

able to predict whether the result of an expression is positive or negative by only using the sign of the constants in the expression.

Abstraction Interpretation. Sometime It is possible to predict the sign of an expression from the sign of constants. The result of adding two positive numbers must be positive. It also is not always possible to predict the sign of an expression from the sign of constants. For example, the result of adding a positive and a negative number can both be positive and negative. In the sign interpretation we will operate with four different value: {zero, + , - , none} where zero indicates that the number is zero, + indicates that the number is positive, - indicates that the number is negative, and non indicates that the number we don't know. We will call the set of these value **Sign**.

Sign={zero, +, -, none}

The rule-of-sign interpretation of addition and multiplication can be specified two tables in Figure 2.3. The interpretation may be viewed as "abstract" additions and multilications. We will write them as operations: $\oplus$ and $\otimes$

$\oplus$	*zero*	+	−	*none*
zero	zero	+	−	*none*
+	+	+	*none*	*none*
−	−	*none*	−	*none*
none	*none*	*none*	*none*	*none*

$\otimes$	*zero*	+	−	*none*
zero	zero	*zero*	*zero*	*zero*
+	*zero*	+	−	*none*
−	*zero*	−	+	*none*
none	*zero*	*none*	*none*	*none*

Figure 2.3: The Table of Sign Operation

respectively.

The abstract version of -123 $\star$ (125+127) is $- \otimes (+ \oplus +)$ which can be computed as $-$. We can compute that the result is negative without actually performing the addition and the multiplication since we know that adding two positive numbers give a positive result and that multiplying a negative and a positive number gives a negative result.

2.3 A Sample Language

Programming Language Using Sets(PLUS)[Yun03, Yun97, Yun98, Yun99] 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 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 semantics of PLUS is showed in Figure 2.4. 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
$[lit_1, \ldots, lit_n]$	$lit_1 :: \ldots :: lit_n ::$ nil
$\{lit_1, \ldots, lit_n\}$	emp with lit_1 with ... with lit_n
$[lit_1..lit_n]$	$lit_1 :: \ldots :: lit_n ::$ 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

Abstract Syntax

c	$\in$	Con	constant
x	$\in$	Bv	bound variables
op	$\in$	Pf	primitive functions
			$op = + \mid - \mid * \mid / \mid \%\ \mid > \mid >= \mid < \mid <= \mid == \mid != \mid \&\& \mid \mid\mid$
f	$\in$	Fv	function variables
e	$\in$	Exp	expression
			$e = c \mid x \mid !e \mid e_1 ope_2 \mid$ 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
List = 2^{Int}	the domain of list value
Bas = Int+Pair+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

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

$$\begin{array}{rcl}
\varepsilon[\![\theta\text{: } c]\!]\ bve\ fve & = & c\text{,intreger } c \\
\varepsilon[\![\theta\text{: } x]\!]\ bve\ fve & = & bve(x) \\
\varepsilon[\![\theta\text{: } !e]\!]\ bve\ fve & = & P[\![!]\!](\varepsilon[\![\theta\text{: } e]\!]\ bve\ fve) \\
\varepsilon[\![\theta\text{: } e_1 \text{ op } e_2]\!]\ bve\ fve & = & P[\![op]\!](\varepsilon[\![\theta_1\text{: } e_1]\!]\ bve\ fve, \varepsilon[\![\theta_2\text{: } e_2]\!]\ bve\ fve) \\
\varepsilon[\![\theta\text{: } x = e]\!]\ bve\ fve & = & bve[(\varepsilon[\![\theta\text{: } e]\!]\ bve\ fve)/x] \\
\varepsilon[\![\theta\text{: } e_1 :: e_2]\!]\ bve\ fve & = & (\varepsilon[\![\theta_1\text{: } e_1]\!]\ bve\ fve) :: (\varepsilon[\![\theta_2\text{: } e_2]\!]\ bve\ fve) \\
\varepsilon[\![\theta\text{: hd } e]\!]\ bve\ fve & = & \varepsilon[\![\theta\text{: } c_1]\!]\ bve\ fve \\
 & & \text{where } e = c_1 :: e_2 \\
\varepsilon[\![\theta\text{: tl } e]\!]\ bve\ fve & = & \varepsilon[\![\theta\text{: } e_2]\!]\ bve\ fve \\
 & & \text{where } e = c_1 :: e_2 \\
\varepsilon[\![\theta\text{: if } e_0 \text{ then } e_1 \text{ else } e_2]\!]\ bve\ fve & = & (\varepsilon[\![\theta_0\text{: } e_0]\!]bve) \rightarrow (\varepsilon[\![\theta_1\text{: } e_1]\!]bve), (\varepsilon[\![\theta_2\text{: } e_2]\!]bve) \\
\varepsilon[\![f_i(e_1, \ldots, e_n)]\!]\ bve\ fve & = & fve[\![f_i]\!](\varepsilon[\![e_1]\!]bve, \ldots, \varepsilon[\![e_n]\!]bve) \\
\varepsilon_f[\![\text{fun } f_i(x_1, \ldots, x_n) = e_i]\!] & = & fve[(\lambda(y_1, \ldots, y_n).\varepsilon[\![e_i]\!][y_k/x_k]fve)/f_i]
\end{array}$$

Figure 2.4: PULS Syntax and Semantic

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 A Theoretic Framework of Constant Lattice

The problem of constant propagation can be formulated in a standard 3-level lattice, originally introduced by Kildall [Kil73], as shown in Figure 2.5 The

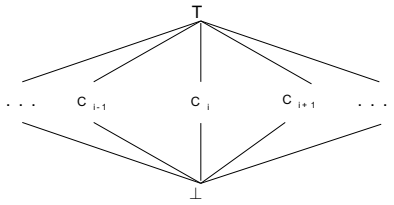

Figure 2.5: Constant Propagation Lattice

output of a constant propagation algorithm is an output assignment of lattice values to variables at each expression in the program. With each variable it associates a *constant propagation lattice* which is one of three types:

- a lattice *top* element $\top$,
- a lattice *bottom* element $\bot$, and
- a constant value c.

The operation between lattices is defined by the following list of rules for the lattice operation *meet* $\sqcap$:

$$
\begin{aligned}
\text{any} \sqcap \top &= \text{any} \\
\text{any} \sqcap \bot &= \bot \\
c_i \sqcap c_j &= c_i \quad \text{if } c_i = c_j \\
c_i \sqcap c_j &= \bot \quad \text{if } c_i \neq c_j
\end{aligned}
$$

Each symbol has an initial value $\top$, which means it has an unknown value yet. The middle layer of the lattice is the constant layer, in which all constants are unordered and are comparable only for equality. Due to the meet operator $\sqcap$, we know that values can only move down in the lattice. Although the lattice is infinite in size, when $\bot$ meet any symbol, it will always be $\bot$. Hence, in the course of intraprocedural constant propagation each symbol is lowered at most twice.

Chapter 3

Constant Propagation

Constant propagation is defined as the process to discover values that are constants on all possible executions of a program and to propagate these values through the program. Optimizing compilers for imperative languages apply data flow optimizations to improve the performance of programs. Data-flow optimizations such as constant propagation based on data flow information that is propagated along the control flow paths of the program. Most of flow analysis techniques are developed from graph representation of the program[Kil73, SGW93, YW02, WZ91]. Instead of using graph representation, we propose a sematic method of constant propagation using abstract interpretation. In this chapter, we introduce a semantics-based analysis for constant propagation in functional languages. Our analysis may be easily adapted into the constant propagation in imperative languages. We assume the language uses strict evaluation and bound variables. We will compare with the other methods in the following chapter.

3.1 A Formal Semantics for Constant Propagation

Denotational semantics is an exact and concise method of attaching meaning to syntax[Blo94]. In the context of constant propagation, we want the information that which variables are bound to constant values. The standard semantics defines the meaning of a program and the value computes, but no information about that which variables are bound to constant values. In this section, we introduce the formal semantics of a sample language, which is an exact but nonstandard semantics that describes the operational notation for computing which variables are bound to constant values in a program. Since the evaluation of expressions at runtime depend on the values themselves, formal semantics contains the information of the standard semantics, as well as the constant environment.

The formal semantics is modeled with a environment cbve is a map from one bound variable to a value L where is concrete value. Our constant propagation analysis works a method that systematically operates the program points so that identifies which variables are bound to constants.

The formal semantics that we capture may be summarized as follow. A program begins with an empty environment *cbve*. Within analysis processing, we update the environment *cbve* when meeting the defined variables of expressions or gathering constant information of variables in the *cbve* when meeting used variables. In whole program, if we meet a constant within one expression, we will return a tuple $\langle c, cbve \rangle$ with the semantic function. It is said that we update no the current *cbve*, and its concrete value is value c. If we meet a

variable x, we will look up the current *cbve* according to the variable x. In this way, we can gather its value L, the element is the concrete value of variable, and we do not update current environment *cbve*.

We compute subexpression of the right-hand according to the current *cbve* when we compute a binding expression. Then, we update the variable within the *cbve* according to result that we already computed. We will compute the consequential or alternative arm after computing the result of relation expression in a branch expression. We go toward the consequential arm when the result of computing relation expression is true. Otherwise, we go toward the alternative arm. For a function call, we pass the actural value through function body and create a new environment *cbve*. We will compute the constant propagation of the function body according to new *cbve* and return the computed result. In next section, we will discuss about its definition. We present the foramal semantic domains and equations of the semantic functions in Figure 3.4.

3.2 A Semantic-Based Analysis for Constant Propagation

The semantics presented in the last section provides the constant information which variables are constants. But, since it relies on standard semantics, some of the variables are not computable at compile-time. Ideally, we would like to simply omit the computation of standard semantics, but continue to compute constant propagation as before. Unfortunately, this is not possible

Semantic Domains

$$
\begin{array}{rcl}
L & = & \text{Int} + \text{Bool} \\
Fun & = & L^n \to L \\
Cbve & = & Bv \to L \\
Cfve & = & Fv \to Fun
\end{array}
$$

Semantic Function

$$
\begin{array}{rcl}
\varepsilon_c & : & Exp \to Cbve \to Cfve \to (L \times Cbve) \\
P_c & : & Pf \to L^n \to (L \times Cbve) \\
\varepsilon_{cf} & : & Exp \to Cfve
\end{array}
$$

$$
\begin{array}{rcl}
\varepsilon_c[\![c]\!]\ cbve\ cfve & = & \langle c, cbve \rangle \\
\varepsilon_c[\![x]\!]\ cbve\ cfve & = & \langle cbve(x), cbve \rangle \\
\varepsilon_c[\![read\ x]\!]\ cbve\ cfve & = & \langle c, cbve \rangle \\
\varepsilon_c[\![pf_1\ e]\!]\ cbve\ cfve & = & P_c[\![pf_1]\!](\pi_1(\varepsilon_c[\![e]\!]\ cbve\ cfve))\ (\pi_2(\varepsilon_c[\![e]\!]cbve\ cfve)) \\
\varepsilon_c[\![e_1\ pf_2\ e_2]\!]\ cbve\ cfve & = & P_c[\![pf_2]\!](\pi_1(\varepsilon_c[\![e_1]\!]cbve\ cfve),\ \pi_1(\varepsilon_c[\![e_2]\!]cbve_1\ cfve)) \\
 & & (\pi_2(\varepsilon_c[\![e_2]\!]cbve_1\ cfve)) \text{ where } cbve_1 = \pi_2(\varepsilon_c[\![e_1]\!]cbve\ cfve) \\
\varepsilon_c[\![x = e]\!]\ cbve\ cfve & = & \langle v_1, cbve_1[v_1/x] \rangle \\
 & & \text{where } \langle v_1, cbve_1 \rangle = \varepsilon_c[\![e]\!]cbve\ cfve \\
\varepsilon_c[\![\text{ if } e_0 \text{ then } e_1 \text{ else } e_2]\!]\ cbve\ cfve & = & (\varepsilon_c[\![e_0]\!]cbve\ cfve) \to (\varepsilon_c[\![e_1]\!]cbve\ cfve), (\varepsilon_c[\![e_2]\!]cbve\ cfve) \\
\varepsilon_c[\![f_i(e_1, \ldots, e_n)]\!]\ cbve\ cfve & = & \langle cfve[\![f_i]\!](v_1, \ldots, v_n), cbve \rangle \text{ where} \\
 & & v_1 = \pi_1(\varepsilon_c[\![e_1]\!]cbve\ cfve), \ldots, v_n = \pi_1(\varepsilon_c[\![e_n]\!]cbve_{n-1}\ cfve) \\
\varepsilon_{cf}[\![\text{fun } f_i(x_1, \ldots, x_n) = e_i]\!] & = & cfve\ whererec \\
 & & cfve = [(\lambda(y_1, \ldots, y_n).\varepsilon_c[\![e_i]\!][y_k/x_k]_{k=1}^n)/f_i]
\end{array}
$$

Figure 3.1: Formal Semantics of Interprocedural Constant Propagation

since in the conditional expression consults result in the standard semantics for the value of predicates before going downward. Without constant values from the predicate, the constant propagation must be approximated with the assumption that either arm could be taken.

We adopt the following conventional notation in the section. Double brackets are used to enclose syntatic elements, as in $\varepsilon_c[\![exp]\!]$. Square brackets are used for environment update, as in $cbve[v/x]$. The notation $cbve[y_i/x_i]_1^n$ is shorthand for $cbve[y_1/x_1, ..., y_n/x_n]$, where the subscript bounds are inferred from context. Angle brackets are used for tupling, as in $\langle v, cbve \rangle$.

In the previous section, we did not carefully explain domain L. Actually, it only has a concrete domain for run-time analysis. In run-time analysis, every variable is constant value because they compute the standard semantics. However, we must consider the situation of reading from a file in compile-time analysis. So we extend domin L to three levels of lattice framework. It mainly contains several elements; D, $\top$, and $\bot$. At the process of constant propagation, their specification is as follow.

- $c \in D$ means that the associated result always has invariant value when the expression existed in all possible execution of the program.
- $\bot$ means that this result is act at the program point according to when the run-time condition decided.
- $\top$ means that is not reachable code.

Now we define a semantic function ε_c that computes information which variables are bound to constants in a program.

Definition Constant propagation function ε_c:

Let Exp be a finite set of text expressions, L be a finite-height meet lattice with a bottom element $\perp$ and a top element $\top$, a *cbve* be a set of containing value L of variables of all reaching-definition. The function ε_c is the set of function from Exp to $L \times Cbve$.

There is a problem that we will encounter in the static analysis of constant propagation. If the value of a variable is given by system I/O, it is impossible for us to know its actual value in compile time.

```
if (p) then          p=true;
       z=1;
       x=z+2;        if(p)then
else                        x=1;
       z=2;          else
       x=z+1;              x=2;
y=x;                 y=x;
```

(a)all paths (b)possible paths

Figure 3.2: Examples of Constant Propagation

Here, we present constant propagation as an example of forward dataflow analysis . Consider Figure 3.2(a). The first use of z can be replaced by 1 and the second by 2. The right hand sides of the two definitions of x can now be simplified to the constant 3, and the final use of x can be replaced by 3. Most constant propagation algorithms in the literature, such as the def-use chain algorithm, discover such all-paths constants. However, additional constants may be found if we ignore definitions inside dead regions of code. In Figure 3.2(b), the predicate of the conditional can be determined to be constant.

By ignoring the definition on the unexecuted branch, the use of x in the last expression can be determined to have value 1. This is, we may ignore any definition that reaches a use via a program flow edge that is never executed. So it can potentially discover more constants than the simple constant propagation analysis developed by Kildall[Kil73]

Other parts of previous analyses all are same, except for the difference in the analysis of conditional expressions. In a static analysis, if each operand in its relational expression is a constant, then we take the branch according to its boolean value. However, we must analyze its both arms when the computation result of the relation expression is $\perp$. Furthermore, they are divided into two new *cbve* in the analysis process for these two arms. We use the $\wedge$ operation to compute the *cbve* when the branch computations merge. The computation of $\wedge$ may be defined as the follows:

$$cbve_1 \wedge cbve_2 \quad = \quad \{\langle x, c\rangle | \langle x, c\rangle \in cbve_1 \text{ and } \langle x, c\rangle \in cbve_2\}$$

$$cbve = cbve_1 \wedge cbve_2 \quad s.t. \quad \begin{cases} cbve(x) = c \quad if\ cbve_1(x) = c\ and \\ \qquad\qquad\qquad cbve_2(x) = c \\ cbve(x) = \perp \quad otherwise \end{cases}$$

We will define a semantic function P_c. There are two types of domain: pf_1, and pf_2 in the the domain of P_c. The pf_1 is a unitary primitive operation and pf_2 is a binary primitive operation.

Definition operator abstracting semantic function P_c:

The P_c can correctly abstract constant information by the elements of P_c after the computed expressions by environment *cbve*.

In the sample language, there are 3 unary primitive functions:

! : we compute the expression by current environment *cbve* if we meet the operator. It returns concrete value and current environment *cbve* when it are constant. Once there exists non-constant in variables of the expression, it returns $\perp$ directly.

hd : we compute first element whether is a constant by current environment *cbve* if we meet the operator, then it returns concrete value and current environment *cbve* if the first element is constant, otherwise returns $\perp$.

tl : we will compute some elements except the first element by current environment *cbve* if we meet the operator. we compute by left to right. It returns concrete value and current environment *cbve* when all elements provide with constant. Once there exists non-constant in variable of the expression, it returns $\perp$ directly.

The operation in elements of pf_2 is similar to the operation *tl* in elements of pf_1. We explain the execution of our function P_c with a simple example. Suppose that the element existing in *cbve* at present have $(a, 1)$,$(b, 2)$,$(c, 3)$ only. For expression $tl\ a :: b :: c :: nil$, we will reason out by $\varepsilon_c[\![tl\ a :: b :: c]\!]cbve\ cfve = P_c[\![tl]\!]\ (\pi_1(\varepsilon_c[\![b :: c]\!]cbve\ cfve))(\pi_2(\varepsilon_c[\![b :: c]\!]cbve\ cfve))$. It will extract L value in *cbve* from left to right, we show the process of the operation *tl* in Figure 3.3.

$\varepsilon_c[\![x = tl\ a :: b :: c]\!]cbve\ cfve$

$= P_c[\![tl]\!](\pi_1(\varepsilon_c[\![b :: c]\!]cbve\ cfve))(\pi_2(\varepsilon_c[\![b :: c]\!]cbve\ cfve))$

$= P_c[\![tl]\!](\pi_1(P_c[\![::]\!](\pi_1(\varepsilon_c[\![b]\!]cbve\ cfve), \pi_1(\varepsilon_c[\![c]\!]cbve_1\ cfve))(\pi_2(\varepsilon_c[\![c]\!]cbve_1\ cfve)))$

$(\pi_2(P_c[\![::]\!](\pi_1(\varepsilon_c[\![b]\!]cbve\ cfve), \pi_1(\varepsilon_c[\![c]\!]cbve_1\ cfve))(\pi_2(\varepsilon_c[\![c]\!]cbve_1\ cfve))))$

$= P_c[\![tl]\!](\pi_1(P_c[\![::]\!](\pi_1(cbve(b)), \pi_1(cbve_1(c)))(\pi_2(cbve_1(c))))\ (\pi_2(P_c[\![::]\!](\pi_1(cbve(b)),$

$\pi_1(cbve_1(c)))(\pi_2(cbve_1(c))))$

$= P_c[\![tl]\!](\pi_1(P_c[\![::]\!](2, 3)cbve_1))(\pi_2(P_c[\![::]\!](2, 3)cbve_1))$

$= P_c[\![tl]\!](\pi_1(2 :: 3, cbve_1))(\pi_2(2 :: 3, cbve_1))$

$= P_c[\![tl]\!](2 :: 3)cbve_1$

$= \langle 2 :: 3, cbve_1 \rangle$

Figure 3.3: The operation of an primitive function

In a functional language, a function call always returns a value. The value of a function is simply the value of its body, which is itself an expression. The actual paramters of a function are mapped to the formal parameters of the function at the time that the function call occurs. If the return value of a function can be recognized as a constant at compile time, then we can evaluate it at compile time and reduce the run time cost. In the analysis of constant propagation for a function, we compute the expressions of actual parameters according to information in the current *cbve*. And we will pass the computed values on to thoses formal parameters in the function body and create a new *cbve* for the function. The new *cbve* only contains the information of formal parameters at the program point. We do computation of function body based on the new cbve.

Theorem 1 $\forall_{e \in E}$, $\varepsilon_c[\![e]\!] cbve\ cfve$ *can correctly extract information which variables are bound to constants in the expression* e.

Proof: We prove it by structural and fixed point induction.

case 1. $\varepsilon_c[\![c]\!]\ cbve\ cfve = \langle c, cbve \rangle$

No variable is in the expression. No change is needed for *cbve*, and we will return a concrete value.

case 2. $\varepsilon_c[\![x]\!]\ cbve\ cfve = cbve(x)$

A variable x appears in the expression, which means that x is used in the expression. No change is needed for *cbve*. We will lookup the variable to current cbve and extract information from the current *cbve*.

case 3. $\varepsilon_c[\![read\ x]\!]\ cbve\ cfve = \langle \perp, cbve[\perp/x] \rangle$

For a binding expression, the exprssion defines a variable, and its value result by system input. So we will return a value $\perp$ and refresh or newly add value $\perp$ of the variable to *cbve*.

case 4. $\varepsilon_c[\![e_1\ pf_2\ e_2]\!]\ cbve\ cfve = P_c[\![pf_2]\!](\pi_1(\varepsilon_c[\![e_1]\!]cbve\ cfve), \pi_1(\varepsilon_c[\![e_2]\!]cbve_1\ cfve))(\pi_2(\varepsilon_c[\![e_2]\!]cbve_1\ cfve))$ where $cbve_1 = \pi_2(\varepsilon_c[\![e_1]\!]cbve\ cfve)$

For the binary primitive expression, it updates no variables. We will compute constant information of subexpressions based on relative environment cbve from left to right. By case 1 ,case 2 and case 3, we know that we can correct extract information. Finally, we will compute constant informaion based on pf_2. We compute concrete value if every variables is constant, otherwise return value $\perp$.

case 5. $\varepsilon_c[\![pf_1\ e]\!]\ cbve\ cfve = P_c[\![pf_1]\!](\pi_1(\varepsilon_c[\![e]\!]\ cbve\ cfve))\ (\pi_2(\varepsilon_c[\![e]\!]cbve\ cfve))$

For the unary expression, it updates no variables. We will compute constant information of subexpressions based on relative environment cbve from left to right. By case1 ,case2, and case 3, we know that we can correct extract information. Finally, we will compute concrete value based on pf_1. We compute concrete value if the variable that all need is constants, otherwise return value $\perp$.

case 6.$\varepsilon_c[\![x = e_0]\!]\ cbve\ cfve = \langle v_1, cbve_1[v_1/x]\rangle\ where\ \langle v_1, cbve_1\rangle = \varepsilon_c[\![e]\!]cbve\ cfve$

For a binding expression, we will update constant information of defined variable to cbve from computing subexpression e_0. So we first compute expression e_0. By repetitively, applying the above cases, we can gather a value in subexpression e_0 and refresh or newly add gathered value to *cbve* based on the variable,.

case 7. $\varepsilon_c[\![$ if e_0 then e_1 else $e_2]\!]\ cbve\ cfve = ...$

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, refresh or newly add informations to these *cbve*s from two branch . After computing braches, we execute operation $\wedge$ at merge node. In the method, we gather a new environment to propagate downward.

case 8. $\varepsilon_c[\![f_i(e_1, \ldots, e_n)]\!]\ cbve\ cfve = cfve[\![f_i]\!](v_1, \ldots, v_n)$ where

$v_1 = \pi_1(\varepsilon_c[\![e_1]\!]cbve\ cfve)$,. . .,$v_n = \pi_1(\varepsilon_c[\![e_n]\!]cbve_{n-1}\ cfve)$

For a funtion expression, we compute actual parameters based on current cbve in turn and bind to formal parameters. Then we repetitively apply the above cases to computation of the function body based on these values of formal parameters. The value of a function body is constant, then it must be terminated. If value of a function is non-constant, their parameter values do not change at next scope and we return a value $\perp$. If the values bound to foramls have non-constant and they are operands on relation expression of function body, we return $\perp$.

Hence, we know that $\forall_{e \in E}$, $\varepsilon_c[\![e]\!]cbve\ cfve$ can correctly extract informations for each expression. $\square$

We show an example in Figure 3.5. We explain our approach by the example. For a given program we compute constant the information for the expressions. We show the process how update *cbve* for each expression in Figure 3.6. Initially, *cbve* and $cfve$ are empty. We start from the first expression of the program and compute with constant propagation. The first expression is $read(x)$. We add a new entry to *cbve* and assign a $\perp$ value to x if the variable x is not in *cbve*. The others can be computed by the method in Figure 3.4. And we will show the result of analysis in Figure 3.5.

Semantic Domains

$$
\begin{array}{rcl}
D & = & \text{Int} + \text{Bool} \\
L & = & \text{D} + \{\bot\} + \{\top\} \\
Fun & = & L^n \to L \\
Cbve & = & Bv \to L \\
Cfve & = & Fv \to Fun
\end{array}
$$

Semantic Function

$$
\begin{array}{rcl}
\varepsilon_c & : & Exp \to Cbve \to Cfve \to (L \times Cbve) \\
P_c & : & Pf \to L^n \to (L \times Cbve) \\
\varepsilon_{cf} & : & Exp \to Cfve
\end{array}
$$

$$
\begin{array}{rcl}
\varepsilon_c[\![c]\!]\ cbve\ cfve & = & \langle c, cbve\rangle \\
\varepsilon_c[\![x]\!]\ cbve\ cfve & = & \langle cbve(x), cbve\rangle \\
\varepsilon_c[\![read\ x]\!]\ cbve\ cfve & = & \langle \bot, cbve[\bot/x]\rangle \\
\varepsilon_c[\![pf_1\ e]\!]\ cbve\ cfve & = & P_c[\![pf_1]\!](\pi_1(\varepsilon_c[\![e]\!]\ cbve\ cfve))\ (\pi_2(\varepsilon_c[\![e]\!]cbve\ cfve)) \\
\varepsilon_c[\![e_1\ pf_2\ e_2]\!]\ cbve\ cfve & = & P_c[\![pf_2]\!](\pi_1(\varepsilon_c[\![e_1]\!]cbve\ cfve),\ \pi_1(\varepsilon_c[\![e_2]\!]cbve_1\ cfve)) \\
 & & (\pi_2(\varepsilon_c[\![e_2]\!]cbve_1\ cfve)) \text{ where } cbve_1 = \pi_2(\varepsilon_c[\![e_1]\!]cbve\ cfve) \\
\varepsilon_c[\![x = e]\!]\ cbve\ cfve & = & \langle v_1, cbve_1[v_1/x]\rangle \\
 & & \text{where } \langle v_1, cbve_1\rangle = \varepsilon_c[\![e]\!]cbve\ cfve \\
\varepsilon_c[\![\text{ if } e_0 \text{ then } e_1 \text{ else } e_2]\!]\ cbve\ cfve & = & \left\{\begin{array}{ll}
\varepsilon_c[\![e_1]\!]cbve_0\ cfve & if\ \varepsilon_c[\![e_0]\!]cbve\ cfve \\
 & \quad = \langle True, cbve_0\rangle \\
\varepsilon_c[\![e_2]\!]cbve_0\ cfve & if\ \varepsilon_c[\![e_0]\!]cbve\ cfve \\
 & \quad = \langle False, cbve_0\rangle \\
\langle v_1, cbve_1\rangle \wedge \langle v_2, cbve_2\rangle & if \varepsilon_c[\![e_0]\!]cbve\ cfve = \langle \bot, cbve_0\rangle \\
 & where \\
 & \quad \varepsilon_c[\![e_1]\!]cbve_0\ cfve = \langle v_1, cbve_1\rangle \\
 & \quad \varepsilon_c[\![e_2]\!]cbve_0\ cfve = \langle v_2, cbve_2\rangle
\end{array}\right. \\
\varepsilon_c[\![f_i(e_1, \ldots, e_n)]\!]\ cbve\ cfve & = & \langle cfve[\![f_i]\!](v_1, \ldots, v_n), cbve\rangle \text{ where} \\
 & & v_1 = \pi_1(\varepsilon_c[\![e_1]\!]cbve\ cfve), \ldots, v_n = \pi_1(\varepsilon_c[\![e_n]\!]cbve_{n-1}\ cfve) \\
\varepsilon_{cf}[\![\text{fun } f_i(x_1, \ldots, x_n) = e_i]\!] & = & cfve\ where \\
 & & cfve = [(\lambda(y_1, \ldots, y_n).\varepsilon_c[\![e_i]\!][y_k/x_k]_{k=1}^n)/f_i]
\end{array}
$$

Figure 3.4: Semantic-Based Analysis of Interprocedural Constant Propagation

Expression
read(x);
y = 2;
y = y-1;
if x = y then z=y else z=1;
w=z;

(a) An Example Program

Expression	*Cbve*
read(x);	$\{x \mapsto \bot\}$
y=2;	$\{x \mapsto \bot\},\{y \mapsto 2\}$
y=y-1;	$\{x \mapsto \bot\},\{y \mapsto 1\}$
if x=y	$\{x \mapsto \bot\},\{y \mapsto 1\}$
then z=y	$\{x \mapsto \bot\},\{y \mapsto 1\},\{z \mapsto 1\}$
else z=1;	$\{x \mapsto \bot\},\{y \mapsto 1\},\{z \mapsto 1\}$
w=z;	$\{x \mapsto \bot\},\{y \mapsto 1\},\{z \mapsto 1\},\{w \mapsto 1\}$

(b)Computing Cbve of an example program

Expression
read(x);
y = 2;
y = 1;
if x = 1 then z=1 else z=1;
w=1;

(c) Optimized program

Figure 3.5: An example of constant propagation

Let cbve and cfve be empty.
1. $\varepsilon_c[\![read(x)]\!]cbve\ cfve = \langle \bot, cbve[\bot/x] \rangle$

2. $\varepsilon_c[\![y = 1]\!]cbve\ cfve = \langle v_1, cbve_1[v_1/y] \rangle$ where $\langle v_1, cbve_1 \rangle = \varepsilon_c[\![1]\!]cbve\ cfve$
$\Longrightarrow \varepsilon_c[\![y = 1]\!]cbve\ cfve = \langle 1, cbve_1[1/y] \rangle$ where $\langle 1, cbve_1 \rangle = \varepsilon_c[\![1]\!]cbve\ cfve$

3. $\varepsilon_c[\![y = y - 1]\!]cbve\ cfve = \langle v_1, cbve_1[v_1/y] \rangle$ where $\langle v_1, cbve_1 \rangle = \varepsilon_c[\![y - 1]\!]cbve\ cfve$
$\Longrightarrow \varepsilon_c[\![y = y - 1]\!]cbve\ cfve = \langle v_1, cbve_1[v_1/y] \rangle$ where
$\langle v_1, cbve_1 \rangle = P_c[\![-]\!](\pi_1(\varepsilon_c[\![y]\!]cbve\ cfve), \pi_1(\varepsilon_c[\![1]\!]cbve_1\ cfve))(\pi_2(\varepsilon_c[\![1]\!]cbve_1\ cfve))$ where
$cbve_1 = \pi_2(\varepsilon_c[\![y]\!]cbve \quad cfve)$
$\Longrightarrow \varepsilon_c[\![y = y - 1]\!]cbve\ cfve = \langle v_1, cbve_1[v_1/y] \rangle$ where
$\langle v_1, cbve_1 \rangle = P_c[\![-]\!](\pi_1(cbve(y)), \pi_1\langle 1, cbve_1 \rangle)(\pi_2\langle 1, cbve_1 \rangle)$ where $cbve_1 = \pi_2(cbve(y))$
$\Longrightarrow \varepsilon_c[\![y = y - 1]\!]cbve\ cfve = \langle v_1, cbve_1[v_1/y] \rangle$ where
$\langle v_1, cbve_1 \rangle = P_c[\![-]\!](\pi_1\langle 2, cbve \rangle), \pi_1\langle 1, cbve_1 \rangle)(\pi_2\langle 1, cbve_1 \rangle)$ where $cbve_1 = \pi_2(\langle 2, cbve \rangle)$
$\Longrightarrow \varepsilon_c[\![y = y - 1]\!]cbve\ cfve = \langle 1, cbve_1[1/y] \rangle$

4. $\varepsilon_c[\![if\ x = y\ then\ z = y\ else\ z = 1]\!]cbve\ cfve$
$\Longrightarrow \varepsilon_c[\![if\ x = y\ then\ z = y\ else\ z = 1]\!]cbve\ cfve = \varepsilon_c[\![z = y]\!]cbve_0\ cfve \wedge \varepsilon_c[\![z = 1]\!]cbve_0\ cfve$ where
$\varepsilon_c[\![x = y]\!]cbve\ cfve = \langle \bot, cbve_0 \rangle$
$\Longrightarrow \varepsilon_c[\![if\ x = y\ then\ z = y\ else\ z = 1]\!]cbve\ cfve = \langle 1, cbve_1 \rangle \wedge \langle 1, cbve_2 \rangle$ where
$\varepsilon_c[\![x = y]\!]cbve\ cfve = \langle \bot, cbve_0 \quad \rangle$

5. $\varepsilon_c[\![w = z]\!]cbve\ cfve = \langle v_1, cbve_1[v_1/w] \rangle$ where $\langle v_1, cbve_1 \rangle = \varepsilon_c[\![z]\!]cbve\ cfve$
$\Longrightarrow \varepsilon_c[\![w = z]\!]cbve\ cfve = \langle v_1, cbve_1[v_1/w] \rangle$ where $\langle v_1, cbve_1 \rangle \quad = cbve(z)$
$\Longrightarrow \varepsilon_c[\![w = z]\!]cbve\ cfve = \langle v_1, cbve_1[v_1/w] \rangle$ where $\langle v_1, cbve_1 \rangle \quad = \langle 1, cbve \rangle$
$\Longrightarrow \varepsilon_c[\![w = z]\!]cbve\ cfve = \langle 1, cbve_1[1/w] \rangle$

Figure 3.6: Work of updating *Cbve* for each expression

3.3 Discussion

Notice that the constant is usually only right for integer value, because expression of the real number is generally architectural and interdependent, they are not well-defined on the constant propagation. The techniques of constant propagation is supported for several purposes in the compilers:

- Expressions evaluated at compile time need not be evaluated at execution time.
- Code that is never executed can be deleted. Unreachable code (a form of dead code) is discovered by identifying conditional braches that always take one of the possible branch paths.
- Detection of the paths never taken simplifies the control flow of a program. The simplified control structure can aid the transformation of the program into a form which is suitable for vector or parallel processing.
- Since many of the parameters to procedures are constants, using interprocedual constant propagation can avoid the expansion of code (inline) that often results from native implementations of procedure integration.

With the above benefits, constant propagation is indispensable to modern optimizing compilers.

Chapter 4

Implementation and Experiment

This section presents the layout of our implementation in Figure 4.1 and experimental result of the algorithm that are proposed in previous section. Our implementation of algorithm is in a MS Windows 2k and Linux simulator Cygwin (version 1.5.9) 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 analysis of interprocedural constant propagation have following phrases.

1. The parses of our program analyzer is generated by Flex and Bison that read the source program and construct its *abstract syntax tree*(AST). We attach attribute *cbve* to each binding expression. The version of Flex and GUN Bison are 2.5.4 and 1.875b respectively.

2. We apply AnalConst algorithm to do Constant propagation analysis.

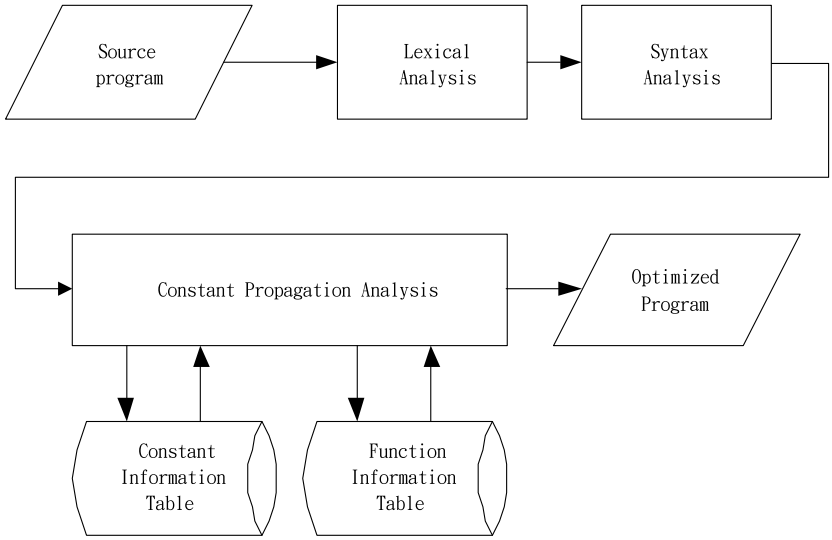

Figure 4.1: The Implementation Layout of Constant Propagation Analysis

4.1 Constant Propagation Information Table

Constant propagation information table is the data structure that we construct during the process of constant propagation analysis. It keep the information that we need for compute constant propagtion. In the table, there are constant information for each reachable variable . It is a hash table. Our approach is to analyze program by theses information. We show it as follows.

Constatn information	table structure
Lattice cell	the lattice value of reachable variables
Type Field	the type value
Value Field	the concrete value provides with type *Constant*
List Field	the concrete list provides with type *List*

The following are basic operations on cbve information table.

- **Table lookup:** we use the variable name to lookup cbve information table.

- **Table update:** we update the table when we meet the variables that is a defined variable in a expression.

In addition, we also need a function information table is the data structure that we construct during the process of analysis. It keep the information that we need for computing constant propagation. In the table, there are information formal parameters and function body for each function definition. It is also a hash table. We gather information from the table when meeting function application.

Function information	table structure
args field	the pointer points to argument list in AST
body Field	the pointer points to function body in AST

The following are basic operations on cfve information table.

- **Table lookup:** we use the function name to lookup cfve information table.
- **Table update:** we update the table when we meet the fuunction definition.

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. **Constant Propagation Analysis:** This function is fist part of analysis. It uses recursive way to extract expressions of program by calling

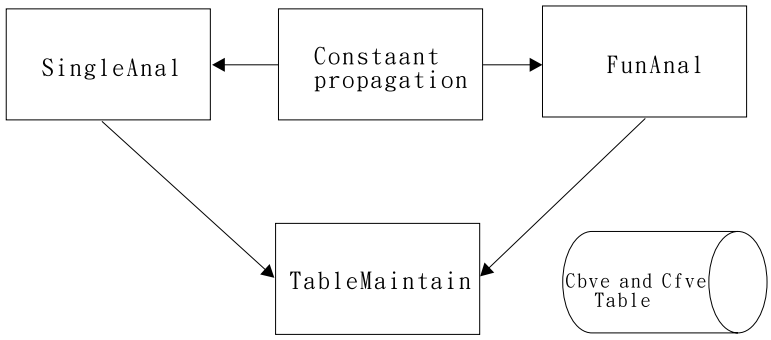

Figure 4.2: Call Graph of Constant Propagation Analysis

function AnalConst. Then we select one functions which are SingleAnal and FunAnal according to expression form.

2. **SingleAnal:** The function is to compute constant information of variable for each expression. And store these information in the cbve table by Call TablesMaintain.

3. **FunAnal:** The function is to compute constant information of returning result for each functional call. Before entering one funtion, we will create a new cbve table and we analyze constant propagation according to new table cbve in function body.

4. **TablesMaintain:** The function is to maintain cbve and cfve table. We access the constant information table by the function. It is used to store or load the data that we need.

We have carried out several experiments to determine the feasibility of our proposed method. We implement the program from the semantic method,and

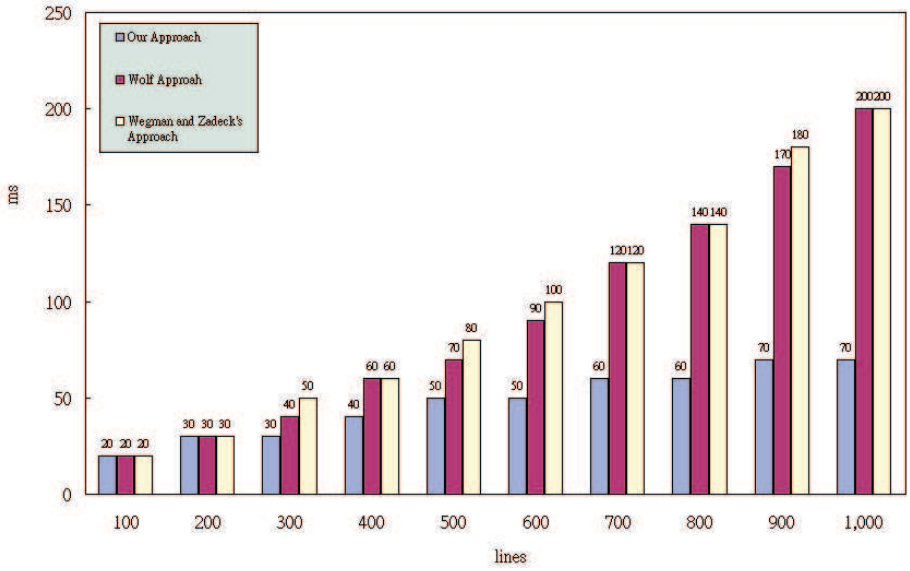

Figure 4.3: Time Cost of Constant Propagation for All Constant Conditions

get the time cost from every 100 lines of program belongs to single procedure. We divide into three cases. One is that all operands in the relation expression are constants. The other is that not all operands in the relation expression are constants. In our approach, we observe one fact that the first case is faster to their methods. We show the experiment result in Figure 4.3. In the second case, we observe that our approach in the some cases is not always slower than their methods. We show the experiment result in Figure 4.4. Without graph representation, we can not identify which variables of merge node in the worst case. So we will compute all elements in Cbve, it will increase time cost at the position. We show the result in Figure 4.5.

In some interprocedure constant propagation, they mainly discuss which formal parameters in a procedure are bound to constants for whole execution.

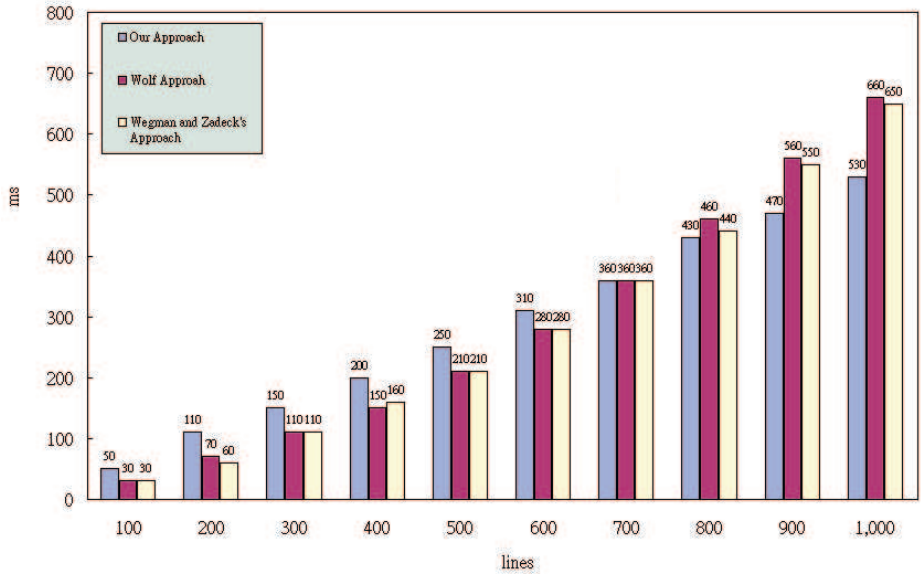

Figure 4.4: Time Cost of Constant Propagation for Some Non-Constant Conditions

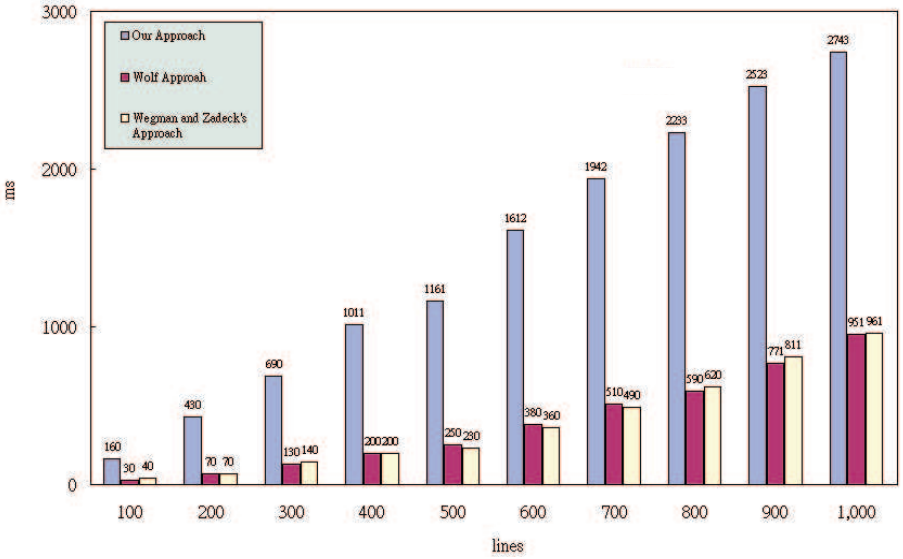

Figure 4.5: Time Cost of Constant Propagation for All Non-Constant Conditions

In function languages, a function often provides with recursive property in general such that the formal parameters often are not bound to constant values. We extend the idea of constants of formal parameters to analyze function result. When we meet a function expression, we bind actual parameters to formal parameters. Before computing the function body, we consider if the values of formal parameters is same as previous ones. If not, we compute function body based on current values of formal parameters, otherwise return $\perp$. After analyzing by our approach, it is decreased time cost on some situations. We are guarantee, in addition, can converge in general semantics of recursive calls to contrast with several previous methods.

Chapter 5

Related Work

In the chapter, we introduce related work about constant propagation. We will introduce constant propagation based on graph representation and interative method of the solution.

5.1 Wegman and Zadeck's Approach

Wegman and Zadeck's[WZ91] presente four algorithms in order of increasing simplest, fastest, and powerful global constant propagation. Those algorithm can be classified in two ways, (1) whole graph or sparse graph representation, and (2) simple or conditional constant detection. There are four sorts of evolution of constant propagation: Simple Constant, Sparse Simple Constant, Conditional Constant, and Sparse Conditional Constant. Every successive algorithm find out at least the constants found by previous algorithm.

The first algorithm, Simple Constant (SC), was developed by Killdall[Kil73]. He was among the first to describe the constant propagation problem and to

give an algorithm solution. The algorithm propagates constants though the program flow graph, and analyzes the program by using only one worklist for CFG nodes. At each node in the program, there are two LatticeCell associated with the value of every variable in the program, one with the value at entry to the node and the other with the exit. The process are visiting a node that involves examination of every LatticeCell at that node. It can be treated the way as functional mapping. Input data is the value of all variables at the entrance of the node, and the output is the set of values for all variables at the exit of the node. The problem of SC is inefficient, since constants are no information assumed about which directional branch will take so that they have been propagated through irrelevant nodes.

The second algorithm, Sparse Simple Constant (SSC),was developed by Reif and Lewis. they transform the program into SSA form so that only one assignment can reach each use. In SSA form, each assignment to a variable is given a unique name and all of the uses reached by that assignment are renamed to match new name of the assignment. At the join nodes, it must add a special form of assignment called a ϕ -function. These elements of ϕ-function are members of control flow predecessors of variable X. The algorithm propagates constants through the SSA graph(Def-Use chain from SSA), and works by keeping one worklist. The worklist is initialized to contain all SSA edges where the definition of an expression is not $\perp$. Although the number of constant which are produced by SSC are the same as SC, it speeds up the running rate.

The third algorithm, Conditional Constant (CC), was developed by Wegbreit. The algorithm takes conditional branches into account, and the framework of that is base on CFG edges which iterate over executable CFG edges instead of over all edges. All program flow edge is initially made as not executable. The Program flow edges are made executable by symbolically executing the program. When an assignment is executed, its out-edge in the flow graph leaving that node is marked as executable and added to the worklist. If a predicate is executed, the result value determine which branch be take, if possible. However, if the conditional expression evaluates to $\perp$, then all out-edges from the node are added to the worklist by conservation assumptions. The algorithm is able to ignore any definition that reach a use from a program flow graph edge that is never executed but it has the same running time as SC.

The final algorithm, Sparse Conditional constant (SCC), was developed by Wegman and Zadeck themselves. The algorithm is the most powerful one than the three algorithms and useful to improves running time which is relative to the previous algorithm CC. The algorithm uses two worklists: one is FlowWorkList which is a worklist of program flow graph edges and the other is SSAWorkList which is a worklist of SSA edges, to go through the program flow graph. Each flow edge will associated with a Executable flag that controls the evaluation of ϕ-function in the target node. In addition, there are two function: one is VisitExpression, and the other is Visit-ϕ function. When the item is a program flow graph edge from the FlowWorkList, then examine

the ExecutableFlag of the edge. If the ExecutableFlag is false, it marks the ExecutableFlag as true, performs Visit-ϕ for all ϕ-function at the target node, if the only one ExecutableFlag associated with the incoming program flow graph edges is true then it performs the VisitExpression for the expression in the node, and if there only is one outgoing flow graph edge in the node then it adds that edge to the FlowWorkList. In SSAWorkList, if the destination of that edge is a ϕ-function then it performs Visit-ϕ and if it is an expression then examine ExecutableFlags on reaching that node only if there are some true then it performs VisitExpression. The algorithm stops when both worklists become empty. The algorithm SCC finds the same class of constants as CC, yet has the same speedup rate over CC as SSC has over SC.

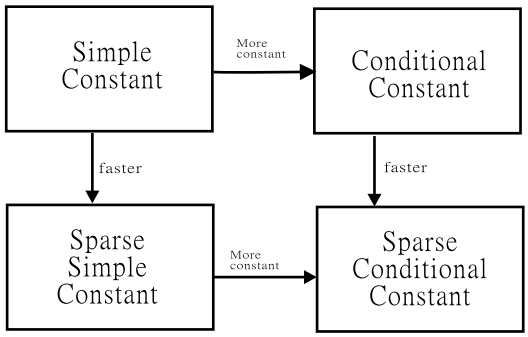

Figure 5.1: The Relationship of Four Constant Propagation Algorithms

In the summary, the sparse graph allows the algorithm could be even more faster than the representation of entire graph. Since to propagate information about each variable to each variable into every node in a program flow graph is inefficient. When we could be avoid evaluating the statement of the program

that are never executed, we can be found more constant by evaluating all predicates which control which branch to take. Because the potential constants could not be killed by the values which are generated in the unreachable areas. We drawn the relationship among four constant propagation algorithms in Figure 5.1.

5.2 Wolf's Approach

Wolf's approach solves constant propagation based on FUD chain. The algorithm is an optimistic demand-driven recursive as opposed to traditional iterative solves. The FUD chain have two important properties. The first is that each use of a variable is reached by a single definition: the use-def chain for any one use is a single link to the unique reaching definition. The second property handles control flow merge nodes, when multiple reaching definitions exist in the original program. Although it is a demand-driven recursive technique, they only solve the problem when the graph is acyclic[**?**]. For the cases with cycles in the FUD chain graph, wolf extends the solution from SSA form to gated single assignment form (GSA) introduced by Ballance. GSA is an extension of SSA form which incorporates control flow information into value merges. they claim that their algorithm is more efficient than Wegman and Zadeck's Sparse Conditional Constant Propagation. Their demand-driven approach associate each ϕ-function with a predicate that decide which branch will be taken. Unlike traditional iterative solvers for constant propagation that require many recomputation until reach a fixed point. They take advantage

of the solver which use the process of classify linear and non-linear induction variables to find constant. But no empirical results was brought out.

5.3 Yung and Wang's Approach

Yung and Wang's Approach solves constant propagation for loop based on FUD chain[YW02]. The FUD chain have two important properties. The first is that each use of a variable is reached by a single definition; the use-def chain for one use is a single link to the the unique reaching definition. The second properties handles control-flow merge points, when multiple reaching definitions exist in the original program. The FUD chain representation is thus more sparse than full use-def chains, since the uses after the merge point have only a single FUD chain link, comparsed to serveral links at each node[**?**]. The investigation was originally motivated by the observation that many constants after loops could not be found by Wagman and Zadeck's approach. The algorithm is also an approach of optimistic demand-driven constant propagation analysis. They combine the idea from Wegman and Zadeck's SCC algorithm of finding conditional constants and the idea from the method by Wolfe et al. of finding linear induction variables. They may classify cycles in the data flow graph and take predicate of branch node into account.

Chapter 6

Conclusions

In this thesis, we introduce the related techniques about our analysis and solve the problem of interprocedure constant propagation. We translated into semantics method wihch used on PLUS language. One of the motivations of our work is to preserve the constant values of function expression, hence such constant values can be propagated the expressions after the function expression. Wegaman and Zadeck's method identifies conditional constant based on SSA and control flow graph(CFG) in single procedure. W.H. Her's algorithms find constant on loop based on FUD chian and CFG in single procedure. Their method neeed the graph representation of a program. Instead of using graph representation, we propse a semantic method of constant propagation using abstract interpretation. We can compute constant information without graph representation of the program. The experiments show that our approach without graph representation is suitable for functional language.

Bibliography

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

[Blo94] A. Bloss, "Path Analysis and the Optimization of Nonstrict Functional Languages," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 16 Issue 3, May 1994.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon, "Interprocedural Constant Propagation," *Proceeding of the 1986 SIGPLAN Conference on Compiler Construction*, pp.152-161 July 1986.

[CH95] Paul R. Carini, Michael Hind, "Flow-Sensitive Interprocedural Constant Propagation," *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* , 1995.

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

[GT93] Dan Grove, Linda Torczon, "Interprocedural constant propagation:

a study of jump function implementation," *ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, Vol. 28 Issue 6, pp.213-232, June 1993.

[Hud86] P. Hudak, "A Semantic Model of Reference Counting and its Abstraction (Detailed Summary)," *Proceedings of the 1986 ACM conference on LISP and functional programming* , August 1986.

[Kil73] G. Kildall. "A Unified Approach to Global Program Optimization," *Proceedings of the First Annual ACM Symposium on Principles of Programming Languages*, 1973.

[KS] B. L. Kurtz and K. Slonneger, "Denotational Semantics," *Formal Syntax and Semantics of Programming Languages*, Chapter9, pp.271-340.

[KU77] J. B. Kam, J. D. Ullman. "Monotone data flow analysis frameworks," *Acta Informatica*, 7:305-317, 1977.

[Ros95] M. Rosendahl, "Introduction to Abstract Interpretation," *DIKU, Computer Science University of Copenhagen*, 1995.

[RS93] R. Metzger, S. Stroud "Interprocedural constant propagation: an empirical study," *ACM Letters on Programming Languages and Systems*, Volume 2 Issue 1-4, March 1993.

[Ten76] R. D. Tennent , "The Denotational Semantics of Programming Lan-

guages," *Communications of the ACM*, Auguest 1976 Volume 19 Number 8.

[Sak94] Alexander Sakharov, "Propagation of Constants and Assertions," *ACM SIGPLAN Notices*, Vol. 29, No.3, March 1994.

[Sco96] Michael L. Scott, "Programming Language Pragmatics," *Morgan Kaufmann Press*, 2000.

[SGW93] Eric Stoltz, Michael P. Gerlek, and Michael Wolfe, "Constant Propagation : A Fresh Demand-Driven Look," *Proceedings of the 1994 ACM symposium on Applied computing*, pp.400-404, Phoenix, Arizona, United States, March 1994

[Ven89] G. A. Venkatesh , "A framework for construction and evaluation of high-level specifications for program analysis techniques," *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, Volume 24 Issue 7, June 1989 .

[Wol96] Michael Wolfe, "High Performance Compilers for Parallel Computing," *Addison Wesley Press*, 1996.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck, "Constant Propagation with Conditional Branches," *ACM Transactions on Programming Languages and Systems*, Vol.13, No.2, pp.181-210, April 1991.

[Yun00] 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.

[Yun01] 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.

[Yun03] C. Yung, "The Investigation and Development of a Compiler for a Program Language using Sets," *National Science Council Project Report*, to appear on Auguest 2003.

[Yun97] C. Yung, "Extending Lambda-Calculus to Sets," *Proceedings on 1997 MASPLAS (in cooperation with ACM SIGPLAN)*, East Stroudsburg University, Pennsylvania, April 1997.

[Yun98] C. Yung, "EAS: An Experimental Applicative Language With Sets," *Proceedings on 1998 MASPLAS (in cooperation with ACM SIGPLAN)*, Rutgers University, New Jersey, April 1998.

[Yun99] C. Yung, "Destructive Effect Analysis and Finite Differencing for Strict Functional Languages", *Ph.D. dissertation*, Computer Science Department, New York University, August 1999.

[YW02] C. Yung and H.H. Wang, "Constant Propagation for Loops with factored Use-Def Chains," *Proceedings on the Eighth Workshop on Compiler Techniques for High-Performance Computing*, National Dong Hwa University, Hualien, Taiwan, March 2002.