Using the Succinct Solver to Implement Flow Logic Specifications of Classical Data Flow Analyses

Han Gao

IMM

Informatics and Mathematics Modelling
Technical University of Denmark

IMM-THESIS-2004-53

July 22, 2004

Preface

This report constitutes my Master of Science Thesis, written during the period from January 26th, 2004 to July 26th, 2004, at the Informatics and Mathematical Modelling department of the Technical University of Denmark.

I would like to thank my supervisor, Professor **Hanne Riis Nielson** who was always the true sources of inspiration, knowledge, guidance and help to myself throughout the period of my master thesis. I would also like to thank PhD students **Terkel Tolstrup** and **Henrik Pilegaard**, with whom I had many informative conversations.

Finally, thanks to my parents for their unfailing support and patient during my studies.

Lyngby, July 26th, 2004

Han Gao

Abstract

Program analysis is a long-history-studied field, which is used to determine properties of programs at compile time without actually executing them. Traditionally, program analysis has been developed and used to optimize compilers and programs in order to get better performance, but recent development shows a promising and much wider application area including validating security properties of Java programs and communication protocols.

A relatively new term in program analysis field is Flow Logic, which concerns about specifying which criteria an analysis estimate should satisfy in order to be acceptable for a program. More important, it separates the specification of an acceptable estimate from the actual computation of the analysis information.

In this thesis, we concern about using flow logic to solve some data flow questions. In particular, we show that data flow analysis can be expressed in term of flow logic. In order to be thoroughly, both bit-vector analyses and monotone analyses are considered. As a comparison, we implement bit-vector analyses in data flow approach, thus paving the way to evaluate the relative advantage of both approaches. This evaluation is based on the execution times of Succinct Solver running on constraints generated for certain scalable benchmarks. The result is encouraging that, flow logic approach is degrees of complexity polynomial better than data flow approach, as far as the analyses presented in this thesis are concerned. As a conclusion, we give some strategies for specifying flow logic specifications for classical data flow analyses. Furthermore, The correctness and expressiveness of Succinct Solver are examined as well.

Keywords

Program analysis, Succinct Solver, Alternation-free Least Fixpoint Logic, Flow Logic, Data Flow Analysis, Framework

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Program analysis is a long-history-studied field, which is used to determine properties of programs at compile time without actually executing them. Traditionally, program analysis has been developed and used to optimize compilers and programs in order to get better performance, for instance the information of variables usage can be used to optimize storage allocation, but recent development shows a promising and much wider application area including validating security properties of Java programs and communication protocols.

Unfortunately, in order to remain computable, analyses can only provide approximate answers to certain questions, which may look like "What is known about ... in program point ...?". There are two different kinds of statements: first, when a property must hold for all possible execution paths, and second, when a property may hold for some particular execution (there is no guarantee that it does not hold). Sometimes the *must*-analysis(under approximation) finds less properties guaranteed than there actually exist, similarly the *may*-analysis(over approximation) sometimes finds more properties than these that actually might hold. The selection of the kind of approximation depends on the analysis to be performed but no matter which one is chosen, it must be safely and conservatively to guarantee an accurate analysis result. The nature of program analysis is illustrated in Figure 1.1.

In general, many approaches exist for building program analyzers, including Data Flow Analysis, Constraint Based Analysis, Abstract Interpretation and, Type and Effect Systems. A good and detailed introduction to program analysis is made in book [2].

Data flow analysis is a mature research area, where frameworks have been developed to classify and solve different classes of data flow problems. These problems can be de-

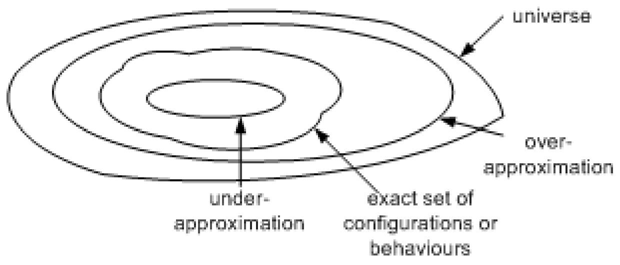


Figure 1.1: The Nature of Approximation

scribed using the formalisms of the different frameworks and a solution algorithm is then selected.

The most popular framework for data flow analysis is the monotone framework. In this framework, data flow problems are defined through lattices of values with a meet(join) operator, often called the *property space*, and a family of *transfer functions* defined over those lattices. Data flow problems are often represented as equations, which encode the information at each node in terms of the information at relevant nodes and the transfer functions. The solution to such a problem is then obtained by computing a fixed point of the equation system. Classical data flow analyses within monotone framework include Reaching Definition Analysis, Very Busy Expression Analysis, Available Expression Analysis and Live Variable Analysis, etc.

A relatively new term in program analysis field is Flow Logic [1], which is actually rooted upon existing technologies and insights, especially from the classical areas of data flow analysis, constraint base analysis and abstract interpretation. Flow logic concerns about specifying which criteria an analysis estimate should satisfy in order to be acceptable for a program. It separates the specification of an acceptable estimate from the actual computation of the analysis information.

A flow logic specification defines: a) the universe of discourse for the analysis estimates, which is usually given by complete lattices; b) the format of the judgement and c) the defining clauses. The main benefit from this is that specification and implementation become independently. Thus one can concentrate on specifying the analysis without considering about the implementation.

One important character of flow logic specifications is, they enjoy the Moore family property, which ensures a best or most precise analysis result exists.

Flow logic can be applied to a variety of programming paradigms, including imperative, functional, object-oriented and concurrent constructs, and even a combination of different paradigms. This is facilitated by its high abstraction: there is a clear separation between specification and implementation for analyses.

Flow logic has been mostly used to specify constraint based analysis([15], [14], [13], etc). In this thesis, we will continue to exploit flow logic by applying it to specify classical data flow analyses.

From flow logic's point of view, an analysis is divided two phases:

1. define a judgement, usually in clausal form, expressing the acceptability of an analysis estimate for a program.
2. compute the clauses to get a solution.

One advantage of such a division is that the second phase is independent of the kinds of analyses as well as programming languages, therefore many state-of-the-art technologies may be employed for computing the analysis information.

A lot of methods could serve for the computing task, like classical iterative-base worklist algorithm and XSB prolog system. Here we shall use Succinct Solver 2.0 [3] as the back-end of the analyses.

Succinct solver, by Nielson and Seidl, is a highly specialized tool calculating the *least* solution for the given clause. A benefit of constructing such a constraint solver is that solver technology may be shared among applications in a variety of programming languages and among many classes of problems.

A wide variety of applications have been used to validate the robustness of the specification language and to suggest the many other features to be provided by the Succinct Solver in order to be a useful tool also for the non-expert-user. These applications include security properties of Java Card bytecode [14], access control features of Mobile and Discretionary Ambients [15] and validations of protocol narrations formalized in a suitable process algebra [13]. Additionally by now the Succinct Solver also incorporates some mechanisms for transforming input ALFP clauses in order to assist in clause tuning aiming at increasing performance and ease of use[6].

Finally, the performance of Succinct Solver against XSB Prolog has been validated. The result is quite encouraging - not only Succinct Solver outperforms XSB Prolog with tabled resolution [12] in optimum cases by exhibiting a lower asymptotic complexity, on a few cases Succinct Solver also has been able to deal with specification for which XSB Prolog could not produce a solution.

1.1 Objectives

The thesis focuses on applying the flow logic approach to the specification of classical data flow analyses within monotone framework for iterative programs and, in particular, exploiting the use of succinct solver for their implementation.

As part of the research we conduct a number of experiments applying both flow logic and data flow specifications to some scalable benchmarks. The expressiveness of ALFP is also investigated.

1.2 Contributions

The contributions of the thesis can be divided into two parts,

1. Besides of the traditional data flow specification, in this thesis, we have developed flow logic specifications for bit-vector framework analyses. By running on our scalable benchmarks, it shows the latter one can be solved more efficiently.
    - Specifications are presented with respect to both data flow and flow logic approach
    - The comparison of the efficiency of these two approaches are made.
    - A general strategy about rewriting data flow specifications of bit-vector framework analyses into flow logic specifications is developed.
2. Flow logic specification for some monotone framework analyses are specified. Included are detection of sign analysis, constant propagation analysis, interval analysis and array bound analysis.

1.3 Organization

This thesis consists of five chapters of which this introduction is the first.

An overview of the remaining five chapters is given as the followings:

In chapter 2 we clarify the syntax and semantics of WHILE language, in which the programs to be analyzed are written. We also give a brief introduction to ALFP in terms of the syntax and the semantics. The solver technologies are explained.

In chapter 3 we specify four classical analyses belonging to bit-vector framework in two different ways, one is in the traditional data flow approach, the other is in flow logic approach. A comparison of these two approaches is made, which shows that the latter one outperforms the former one. Relationship between these two kinds of specifications are analyzed.

In chapter 4 we extend the analyses to monotone framework. Detection of signs analysis, constant propagation analysis and array bounds analysis are presented. A general strategy of writing flow logic specifications for monotone analysis is given.

In chapter 5 gives a conclusion for the thesis and proposes directions for future work

Appendix A contains timing strategy and how the results are presented.

Appendix B contains experiments result conducted on benchmark, b-m

Appendix C contains the guide for how to use the analyzers

Chapter 2

Setting the Scene

One of the main characters of Flow logic is the separation of specification of analyses and implementation of analyses. The analyses will be specified for a simple WHILE language presented below; the actual analyses will be presented in the next chapter. The implementation of the analyses will also be discussed later; however in this chapter we shall present the generic tool the Succinct Solver, to be used for that.

2.1 WHILE Language

In the first part of this chapter, we shall concentrate on a small imperative language WHILE. Before starting the analyses, we give the syntax and grammar of WHILE language, in which programs to be analyzed are written. WHILE is a small Pascal-like programming language. It is only composed by variable declaration and statement. Due to the time limitation and our emphasis on intra-procedure analysis, neither procedures nor functions can be defined.

2.1.1 Syntax

A program in WHILE is a series of variable declarations followed by a sequence of statements.

Suppose we have a fixed countable set Var of simple variables[1], Arr of array names and a countable set of Num of natural numbers. The set of programs p as well as variables

[1]in contrast to array elements

v, expressions e, statements s and declarations dec are generated by the syntax shown in Figure 2.1, where $x \in Var$, $A \in Arr$, $n, n_1, n_2 \in Num$.

$$
\begin{array}{lcl}
v & ::= & x \mid A[e] \\
e & ::= & true \mid false \mid n \mid v \mid e_1 \; op \; e_2 \mid \neg e \\
op & ::= & + \mid - \mid \times \mid \div \mid \leq \mid > \mid = \mid and \mid or \\
s & ::= & [v := e]^l \mid [Skip]^l \mid s_1 \,;\, s_2 \mid \text{If } [e]^l \text{ Then } s_1 \text{ Else } s_2 \\
 & \mid & \text{While } [e]^l \text{ Do } s \mid \text{Begin } dec \; s \text{ End} \\
dec & ::= & \text{Var } x \mid \text{Array } A \text{ of } [n_1..n_2] \mid dec_1 \,;\, dec_2 \\
p & ::= & \text{INPUT } dec_1 \text{ OUTPUT } dec_2 \; s;
\end{array}
$$

Figure 2.1: Abstract Syntax of WHILE Language

The primitive elements in WHILE are numbers as well as boolean values ($true$ and $false$), which are then used to construct complex variables and expressions. Here variables include simple variables and arrays indexed by an expression.

According to the nature of operators, expressions can be divide into three categories: arithmetic expressions, boolean expressions and relational expressions but as we shall see later we shall not need to distinguish between these expressions.

It is required that all variables in a program are declared before used. Array's declaration must indicate the name of the array as well as it's lower and upper bounds. The lower bound can be chosen freely and not necessary to be 0[2]; but it is programmer's responsibility to make sure the lower bound should always less or equal than the upper bound. Each statement may have its own variable declarations, these declared variables are only locally to the statement, meaning that they are not valid outside the statement.

The declaration part of a program is divided into to two parts, which follow key words INPUT and OUTPUT, respectively. Variables declared in both of them are global to the whole program. The difference is the OUTPUT part contains variables, whose values are going to be used after the very end point of the program. This is similar to the *return variable* of a function. By now, these *return variables* are only used in the Live Variable

[2]not like in C or Java

Analysis. We will come back to that later in Chapter 3.5.

If not explicitly stated, the notation in Figure 2.2 will be used throughout this thesis.

x, y	$\in$	Var	$SimpleVariable$
m, n	$\in$	Num	$Numberal$
e	$\in$	Exp	$Expression$
s	$\in$	$Stmt$	$Statements$
l	$\in$	Lab	$Labels$
B	$\in$	$Block$	$Elementary\ Blocks$
dec	$\in$	Dec	$Declarations$
p	$\in$	$Prog$	$Program$
A, C	$\in$	Arr	$Array\ Names$

Figure 2.2: Notation of Thesis

To aid the analyses, we shall associate data flow information with single assignment statements, skip statements and the tests that appear in conditionals and loops, each of which is regarded as an *elementary block*. We also assume that distinct elementary blocks are initially assigned distinct labels.

Example 2.1 One example WHILE program, calculating the sum of elements of array A, is,

```
INPUT     Array $A$ of [2..4]
OUTPUT    Var $sum$
Begin     Var $x$
          $[x := 2]^1$;
          While $[x <= 4]^2$ Do
           (  $[sum := sum + A[x]]^3$;
              $[x := x + 1]^4$)
End
```

2.1.2 Semantics

The Semantics allows us to pinpoint the precise meaning of programs. In order to avoid possible ambiguity in the future, the semantics of WHILE is clarified in this section. Many kinds of semantics would be appropriate but among the operational semantics we shall now prefer natural semantics(big-step operational semantics) with environment and state.

Since a program can be viewed as a combination of declaration and statement, we then give the semantic of them separately. Before going into detail, some preparations have to be done.

Firstly, we introduce a *State*, which associates each variable and array element with its current value. It is defined as,

$$\sigma \in State = (Var \cup Arr \times \mathbb{Z}) \hookrightarrow \mathbb{Z} \cup \mathbb{T}$$

where $\mathbb{Z}$ represents integer number and $\mathbb{T}$ contains the truth value ($true$ and $false$).
Secondly, we define an environment $\rho \in Env$, which maps an array name to its upper and lower bounds, represented as,

$$\rho \in Env = Arr \to \mathbb{Z} \times \mathbb{Z}$$

which is a total function since arrays are required to be declared before used.
Finally, an auxiliary total function $\mathcal{E}$ is used to determine the exactly value of an expression. The functionality of $\mathcal{E}$ is,

$$\mathcal{E} : Exp \to Env \times State \hookrightarrow \mathbb{Z} \cup \mathbb{T}$$

and the definition is given in below table.

$\mathcal{E}[n](\rho, \sigma) = n$		
$\mathcal{E}[x](\rho, \sigma) = \sigma(x)$		
$\mathcal{E}[A[e]](\rho, \sigma) = \sigma(A, i)$	if	$i = \mathcal{E}[e](\rho, \sigma) \in \mathbb{Z}$
		$\rho\ (A) = (i_1, i_2)$
	and	$i_1 \leqslant i \leqslant i_2$
$\mathcal{E}[e_1\ op\ e_2](\rho, \sigma) = w$	if	$\mathcal{E}[e_1](\rho, \sigma) = w_1$
		$\mathcal{E}[e_2](\rho, \sigma) = w_2$
	and	$w = w_1\ op\ w_2$ is defined.
$\mathcal{E}[\neg e](\rho, \sigma) = \neg w$	if	$\mathcal{E}[e](\rho, \sigma) = w \in \mathbb{T}$

Any constant, say n, has the value n and the values of simple variables is determined by looking up ρ. In the case, the expression is of the form $A[e]$, e is evaluated followed by a check on whether the result is an integer that falls within the bounds of array A, which are obtained by looking up ρ. This makes sure no out of bound error may occur here. In the case $[e_1\ op\ e_2]$, both e_1 and e_2 are evaluated followed by applying operator *op* on their values. Expression $\neg e$ has the negated value of e, providing that e has a boolean value.

Semantics of Program

We shall begin with the semantics of program and then statements and declarations. When come to the semantics of program, it's initial *environment* ρ_0 and *state* σ_0 are empty and therefore the semantic meaning of that program is computed wrt. its own declarations.

$$[\text{prog}_{NS}] \quad \frac{< dec_1, \rho_0, \sigma_0 > \rightarrow (\rho', \sigma') \;\; < dec_2, \rho', \sigma' > \rightarrow (\rho'', \sigma'') \;\; \rho'' \vdash < s, \sigma'' > \rightarrow \sigma'''}{\rho_0 \vdash < \text{INPUT } dec_1 \text{ OUTPUT } dec_2 \ s, \ \sigma_0 > \rightarrow \sigma'''}$$

Symbol "$\rightarrow$" defines a transition system between two configurations. The idea of $\rho \vdash < s, \sigma > \rightarrow \sigma'$ is the execution of s from σ based on environment ρ will result in a new state σ'. Similarly $< dec, \rho, \sigma > \rightarrow (\rho', \sigma')$ means declaration *dec* will update the current ρ and σ to new environment ρ' and new state σ'. Both transition systems with be specified below.

Semantics of Statements

The semantic meaning of a statement in WHILE always returns a state. The environment remains static unless we are dealing with a statement when environment is updated by the corresponding program statement's declaration. In this case, the statement will be executed on the updated environment.

In Table 2.1, it is stated that in case of assignment statement, an array bound checking is performed, which is the same as when performing $\mathcal{E}[A[e]]$, in order to ensure index is always within the array bounds. In the case of composite statement, the semantic meaning of the first statement is computed. The resulting *state* σ is then used in the computation of the semantic meaning of the second statement. Thus ensure the second statement is computed wrt. an updated *state*.

In case of a block statement[3] the current environment ρ and state σ is updated by the statement's declarations as presented by $< dec, \rho, \sigma > \rightarrow (\rho', \sigma')$; all global declarations in the current environment ρ and state σ are overwritten by the local declarations stated in the statement. Statement inside the block is then executed based on the new ρ and σ. When the block is left, local variables are restored to the values held before entering the block. This ensures that local variables and global variables with the same name won't be interfered.

This value restoration is done in two steps; firstly, all the local variables are collected and secondly update the *state* σ by assigning the original values to those variables, which

[3]a statement has local variable declarations

are corresponding to the following two operations.
Operation DV($Dec; Dec$) collects variable names, which are locally to the block, from the declaration part. In the case the declaration is for an array, all the elements within the array's bounds will be collected.

$$\begin{array}{lcl} \mathrm{DV}(Var\ x;\ Dec) & = & \{x\}\ \cup\ \mathrm{DV}(Dec) \\ \mathrm{DV}(Array\ A\ of\ [n1..n2];\ Dec) & = & \{(A,i)\ \mid\ n1 \leqslant i \leqslant n2 \text{ if } n1 \leq n2\}\ \cup\ \mathrm{DV}(Dec) \\ \mathrm{DV}(\varepsilon) & = & \emptyset \end{array}$$

ε appears in the third rule of above operation denotes empty list, which ensures the termination of the operation. The resulted set is used to indicate which variables should be restored to the original value when a block is leaving.

Operation ($\sigma'[X \mapsto \sigma]$) defines a transition from initial state to final state. The idea of it is the *state* σ' is the same as σ except for variables in set X, whose values are specified by *state* σ.

$$\begin{array}{lcl} (\sigma'[X \mapsto \sigma])(x) & = & \begin{cases} \sigma(x) & \text{if } x\ \in\ X \cap Var \\ \sigma'(x) & \text{if } x\ \in\ Var \text{ and } x\ \notin\ X \end{cases} \\ (\sigma'[X \mapsto \sigma])(A,i) & = & \begin{cases} \sigma(A,i) & \text{if } (A,i)\ \in\ X \cap (Arr \times \mathbb{Z}) \\ \sigma'(A,i) & \text{if } (A,i)\ \in\ (Arr \times \mathbb{Z}) \text{ and } (A,i)\ \notin\ X \end{cases} \end{array}$$

$[\text{ass}_{NS}^{Var}]$	$\rho \vdash < x := e,\ \sigma > \rightarrow \sigma[x \mapsto \mathcal{E}[e](\rho, \sigma)]$
$[\text{ass}_{NS}^{Arr}]$	$\rho \vdash < A[e] := e', \sigma > \rightarrow \sigma[(A, i) \mapsto \mathcal{E}[e'](\rho, \sigma)]$ where $i = \mathcal{E}[\text{e}](\rho, \sigma) \in \mathbb{Z}$, $\rho(A) = (i_1, i_2)$ and $i_1 \leqslant i \leqslant i_2$
$[\text{skip}_{NS}]$	$\rho \vdash < \text{Skip}, \sigma > \rightarrow \sigma$
$[\text{if}_{NS}^{true}]$	$\dfrac{\rho \vdash < s_1,\ \sigma\ > \rightarrow\ \sigma'}{\rho \vdash < \text{If } e \text{ Then } s_1 \text{ Else } s_2,\ \sigma\ > \rightarrow\ \sigma'}$ if $\mathcal{E}[e](\rho, \sigma) = \text{true}$
$[\text{if}_{NS}^{false}]$	$\dfrac{\rho \vdash < s_2,\ \sigma\ > \rightarrow\ \sigma'}{\rho \vdash < \text{If } e \text{ Then } s_1 \text{ Else } s_2,\ \sigma\ > \rightarrow\ \sigma'}$ if $\mathcal{E}[e](\rho, \sigma) = \text{false}$
$[\text{while}_{NS}^{true}]$	$\dfrac{\rho \vdash < s, \sigma > \rightarrow \sigma' \quad \rho \vdash < \text{While } e \text{ Do } s,\ \sigma' > \rightarrow\ \sigma''}{\rho \vdash < \text{While } e \text{ Do } s\ ,\ \sigma > \rightarrow\ \sigma''}$ if $\mathcal{E}[e](\rho, \sigma) = \text{true}$
$[\text{while}_{NS}^{false}]$	$\rho \vdash < \text{While } e \text{ Do } s\ ,\ \sigma > \rightarrow\ \sigma$ if $\mathcal{E}[e](\rho, \sigma) = \text{false}$
$[\text{comp}_{NS}]$	$\dfrac{\rho \vdash < s_1, \sigma > \rightarrow \sigma' \ \rho \vdash < s_2, \sigma' > \rightarrow \sigma''}{\rho \vdash < s_1; s_2, \sigma > \rightarrow \sigma''}$
$[\text{block}_{NS}]$	$\dfrac{< dec, \rho, \sigma > \rightarrow (\rho', \sigma') \quad \rho' \vdash < s, \sigma' > \rightarrow \sigma''}{\rho \vdash < \text{Begin } dec\ s \text{ End}, \sigma > \rightarrow \sigma''[DV(dec) \rightarrow \sigma]}$

Table 2.1: Nature Semantics for Statement

Semantics of Declaration

This section explains how σ and ρ are updated according to a set of declarations.

$$[dec_{NS}^{Var}] \quad \vdash < Var\ x, \rho, \sigma > \to (\rho \mid_x, \sigma \mid_X)$$
$$\text{where } X \in \{x\} \cup \{(x,i) \mid i \in \mathbb{Z}\}$$

$$[dec_{NS}^{Arr}] \quad \vdash < \text{Array } A \text{ of } [n1..n2], \rho, \sigma > \to (\rho[A \mapsto (n1, n2)], \sigma \mid_X)$$
$$\text{where } X \in \{A\} \cup \{(A,i) \mid i \in \mathbb{Z}\}$$

$$[dec_{NS}^{Comp}] \quad \frac{\vdash < dec_1, \rho, \sigma > \to (\rho', \sigma') \ \ \vdash < dec_2, \rho', \sigma' > \to (\rho'', \sigma'')}{\vdash < dec_1; dec_2, \rho, \sigma > \to (\rho'', \sigma'')}$$

$$[dec_{NS}^{Empty}] \quad < \varepsilon, \rho, \sigma > \to (\rho, \sigma)$$

$\mid$ is a restriction operator, which maps an element in ρ or σ(if there is such one) to *undef*(undefined), corresponding to removing that element from the declaration.

$\rho \mid_A = \rho[A \to undef]$
$\sigma \mid_x = \sigma[x \to undef] \qquad \sigma \mid_{(A,i)} = \sigma[(A,i) \to undef]$

In the case the declaration is for $Var\ x$, the bounds information for array named x will be removed from ρ. σ is updated by removing the values associated with $Var\ x$ as well as with array x.
In the case the declaration is for array A, bounds of A in ρ are then updated to the declared values. Values associated with $Var\ A$ and array A in σ are removed.

Example 2.2 Consider the program,

```
INPUT var $x$ OUTPUT var $y$
Begin $x$:= 1;
    Begin var $x$
        $x$:= 5
    End;
    y:= $x$ + 1
End
```

The x in x:= 5 relates to the local variable x, whereas the x in y:= x + 1 relates to the global variable x that is also used in the statement x:= 1. Therefore, the statement y:= x + 1 assigns y the value 2 rather than 6.

Example 2.3 Suppose $\rho(A) = (1{,}2)$ $\sigma(x) = 5$ $\sigma(A, 1) = 1$ $\sigma(A, 2) = 2$
Declaration $(Var\ A;\ Array\ x\ of\ [5, 6])$ will result in

- after evaluating $Var\ A$
  $\rho(A) = undef$
  $\sigma(x) = 5$ $\sigma(A, 1) = undef$ $\sigma(A, 2) = undef$

- after evaluating $Array\ x\ of\ [5, 6]$
  $\rho(A) = undef$ $\rho(x) = (5{,}6)$
  $\sigma(x) = undef$ $\sigma(x, 5) = undef$ $\sigma(x, 6) = undef$
  $\sigma(A, 1) = undef$ $\sigma(A, 2) = undef$

2.2 Succinct Solver 2.0

Succinct Solve, implemented in Standard ML, is a highly specialized tool which incorporates state-of-the-art approaches to constraints solving. One advantage of such a tool is that it separates implementation considerations from specification and therefore increases the likelihood that a correct and useful analysis can be developed with only a limited effort. Furthermore, solver technology may be shared among applications in a variety of programming languages and among many classes of problems.
To obtain easily readable formulae, Succinct Solver accept a subset of first order predicate logic that allows the appropriate theoretical results to be established, which is the so called Alternation-free Least Fixed Point Logic (ALFP for short). ALFP only disallows those features of first order predicate logic that would make it impossible to ensure the existence of a least solution.
Succinct Solver operates over finite domains and computes the *least* solution of the given clause according to the *universe*. User can choose to input ALFP clauses to the solver in either SML data structures, text file[4], stdIN(standard input) or string and the output can be one of the four forms as well. We hope the following example could give reader a preliminary idea about what's the least solution and how does ALFP looks like.

Example 2.4
The solving result by Succinct Solver for clause

$$R(a, a) \wedge R(b, a) \wedge R(c, a) \wedge R(d, a) \wedge (\forall x\ (R(b, x) \wedge R(x, a)) \vee R(y, x) \Rightarrow Q(x))$$

[4]parsed by solver's built-in parser

is

The Universe:	(a, b, c, d, y)
Relation $R/2$:	$(d, a), (c, a), (b, a), (a, a),$
Relation $Q/1$:	$(a),$

The Universe contains ground terms[5]. Note that free variable occurring in a formula is also regarded as ground term, therefore y is also included in universe. The number followed each name of relation indicates the aries of that relation (i.e. R is a two-ary relation). The working flow of Succinct Solver is that it iterative through all the atoms in the universe to search for an atom, say n, such that either both $R(b, n)$ and $R(n, a)$ exists or $R(y, n)$ exists. If there is such an n, then n is known to be an interpretation of predicate Q.

In this chapter, we shall first give a brief introduction to ALFP, which could be able to enable reader begin to use Succinct Solver, and then come a bit deeper into solver's internal structure aiming at showing reader some clause transformations, which may result in better solver performance.

2.2.1 ALFP in Brief

The Alternation-free fragment of Least Fixed Point Logic(ALFP) extends Horn clauses by allowing both existential and universal quantifications in pre-conditions, negative queries (subject to the notion of stratification), disjunctions of pre-conditions and conjunctions of conclusion. Also, ALFP formulas are subject to a strict syntactical left-to-right stratification requirement (as known from Datalog) that allows negation to be dealt within a convenient manner. Such stratified Least Fixed Point formulas are alternation-free.
The introduction of ALFP given in this section is in terms of syntax and the semantics.

Syntax

Given a fixed countable set $\mathcal{X}$ of variables, a countable set $\mathcal{C}$ of constant symbols, a finite ranked alphabet $\mathcal{R}$ of predicate symbols and a finite set $\mathcal{F}$ of function symbols, the set of ALFP clauses, *cl*, together with pre-conditions, *pre* and terms t, are generated by the grammar showed in Figure 2.3.

In the syntax, R is a k-ary predicate symbol for $k \geqslant 1$, t_1, t_2, $\ldots$ denote arbitrary variables. $R(\ldots)$ and $\neg R(\ldots)$ occurred in pre-conditions are called queries and negative queries, respectively, whereas the others are called assertions of predicate R.
In order to deal with negations conveniently, a notion of stratification is introduced, which

[5]a term not containing variables

$$
\begin{array}{lcl}
t & ::= & c \ \mid\ x \ \mid\ f(t_1,\ldots,t_k) \\
\\
pre & ::= & R(t_1,\ldots,t_k) \ \mid\ \neg R(t_1,\ldots,t_k) \ \mid\ pre_1 \wedge pre_2 \\
 & \mid & pre_1 \vee pre_2 \ \mid\ \exists x : pre \ \mid\ t_1 = t_2 \ \mid\ t_1 \neq t_2 \\
 & \mid & 1 \ \mid\ 0 \\
\\
cl & ::= & R(t_1,\ldots,t_k) \ \mid\ 1 \ \mid\ cl_1 \wedge cl_2 \\
 & \mid & pre \Rightarrow cl \ \mid\ \forall x : cl
\end{array}
$$

Figure 2.3: Abstract Syntax of ALFP clauses

could be expressed by making use of a mapping *rank*: $\mathcal{R} \rightarrow \mathbb{N}$ that maps predicate symbols to ranks in $\mathbb{N}$(natural numbers). A clause *cl* is an *alternation-free-Least Fixpoint formula* if it has the form $cl = cl_1 \wedge \ldots \wedge cl_s$ and there is a function *rank*: $\mathcal{R} \rightarrow \mathbb{N}$ such that for all j=1,. . .,s, the following properties hold:

1. all predicates of assertions in cl_j have rank j;
2. all predicates of queries in cl_j have ranks at most j; and
3. all predicates of negated queries in cl_j have ranks strictly less than j.

Intuitively, stratification ensures that a negative query is not performed until the relation queried has been fully evaluated.

Example 2.5
We define an equality predicate **eq** and a non-equality predicate **neq** by the clause:

$$(\forall x : \mathbf{eq}(x,x)) \wedge (\forall x : \forall y : \neg\mathbf{eq}(x,y) \Rightarrow \mathbf{neq}(x,y))$$

The clause has the form $cl = cl_1 \wedge cl_2$. By applying function *rank* on the clause, we have *rank*(**eq**)= 1 and *rank*(**neq**) = 2 since they are assertions in cl_1 and cl_2, respectively. Property 3 requires that *rank*(**eq**) should less than 2, which clearly holds. Therefore the conditions are fulfilled.

Example 2.6
The following formula is ruled out by the function *rank*:

$$(\forall x : \neg P(x) \Rightarrow Q(x)) \wedge (\forall x : Q(x) \Rightarrow P(x))$$

This is because it is impossible to have *rank*(P) < *rank*(Q) and *rank*(Q)≤*rank*(P)(these inequations are resulted from properties 3 and 2).

In later chapters, we shall use t for representing a term, pre for pre-condition, p for predicate symbol and cl for a clause.

Semantics

Given a non-empty and countable universe $\mathcal{U}$ of atomic values together with interpretations ρ and σ for predicate symbols and free variables, respectively, we define the satisfaction relations

$$(\rho, \sigma) \models pre \quad \text{and} \quad (\rho, \sigma) \models cl$$

for pre-conditions and clauses as in Table 2.2. Functionality of $\models$ is defined as,

$$\models \quad (\rho \times \sigma \times (pre \cup cl)) \rightarrow \{true, false\}$$

Here we write $\rho(R)$ for the set of k-tuples$(a_1, \ldots, a_k)$ from $\mathcal{U}$ associated with the k-ary

$(\rho, \sigma) \models R(x_1, \ldots, x_k)$	iff	$(\sigma(x_1), \ldots, \sigma(x_k)) \in \rho(R)$
$(\rho, \sigma) \models \neg R(x_1, \ldots, x_k)$	iff	$(\sigma(x_1), \ldots, \sigma(x_k)) \notin \rho(R)$
$(\rho, \sigma) \models pre_1 \wedge pre_2$	iff	$(\rho, \sigma) \models pre_1$ and $(\rho, \sigma) \models pre_2$
$(\rho, \sigma) \models pre_1 \vee pre_2$	iff	$(\rho, \sigma) \models pre_1$ or $(\rho, \sigma) \models pre_2$
$(\rho, \sigma) \models \exists x : pre$	iff	$(\rho, \sigma[x \mapsto a]) \models pre$ for some a $\in \mathcal{U}$
$(\rho, \sigma) \models \forall x : pre$	iff	$(\rho, \sigma[x \mapsto a]) \models pre$ for all a $\in \mathcal{U}$
$(\rho, \sigma) \models R(x_1, \ldots, x_k)$	iff	$(\sigma(x_1), \ldots, \sigma(x_k)) \in \rho(R)$
$(\rho, \sigma) \models 1$	iff	always
$(\rho, \sigma) \models cl_1 \wedge cl_2$	iff	$(\rho, \sigma) \models cl_1$ and $(\rho, \sigma) \models cl_2$
$(\rho, \sigma) \models pre \Rightarrow cl$	iff	$(\rho, \sigma) \models cl$ whenever $(\rho, \sigma) \models pre$
$(\rho, \sigma) \models \forall x : cl$	iff	$(\rho, \sigma[x \mapsto a]) \models cl$ for all a $\in \mathcal{U}$

Table 2.2: Semantics of pre-conditions and clauses

predicate symbol R, we write $\sigma(x)$ for the atom of $\mathcal{U}$ bound to x and finally $\sigma[x \mapsto a]$ stands for the mapping that is as σ except that x is mapped to a.
The upper part of the Table 2.2 lists the rules for pre-conditions. In case of query $R(args)$, atoms in $\mathcal{U}$, which are bound to $x_1, \ldots, x_k$, are examined whether their combination is a possible interpretation of R stored in ρ. In case of conjunction of queries, it is satisfiable iff both queries are satisfiable. In case of existential quantification, it is satisfiable iff there exists at least one atom in universe such that after adding it as a bound value of x to σ, the pre-condition is satisfiable. In case of universal quantification, the only difference with

existence quantification is the all the atoms in universe should satisfy the above condition. Rules for clauses are listed in lower part of the table. 1 here represents the truth value, therefore it always satisfiable. In case clause has the form $pre \Rightarrow cl$, satisfiability of cl is checked only when pre is satisfiable.

2.2.2 Overview of the Succinct Solver

Succinct Solver aims at non-expert-user; understanding syntax and semantic are sufficient for the user to write ALFP clauses and begin to use Succinct Solver. But things could be done better. As mentioned in the beginning of this chapter Succinct Solver has some internal mechanisms to assist in clause tuning in order to increase it's performance, these mechanism are general. User may design his own specifical clause transformation to solve the clause more efficiently. In this section, we give an overview of the solver by sketching the program structure and explain briefly functionalities of the main functions. It serves to help reader understand the strategies about clause tuning given at the end of the section. The Succinct Solver has been coded using a combination of recursion, continuations, prefix trees and memoisation [4]. It incorporates most of the technology of differential worklist solvers. However, rather than using an explicit worklist, it is based on the top-down solver of Le Charlier and van Hentenryck [8]. This gives a rather robust solver with good time and space performance.

Program Structure

The program structure of the solver is sketched in Figure 2.4, where three main modules are shown.

Module *Translate* is responsible for parsing the ALFP clause from the text file and transforming it into an internal representation to facilitate processing, where atoms and predicate symbols are finally represented as integers. At the same time, it extracts the useful static information, i.e. *universe*, etc, into the internal data structures.
The information in *universe* is used by the *Output* module to produce the final output. The information of parsed clause is used by the *Solve* module to compute the solution to the clause.
Module *Solve* then processes the clause and computes the solution by manipulating two main data structures,

- *env*, which is a *partial environment* mapping variables to atoms of the universe. The environment is constructed in a lazy fashion meaning that variables may not have

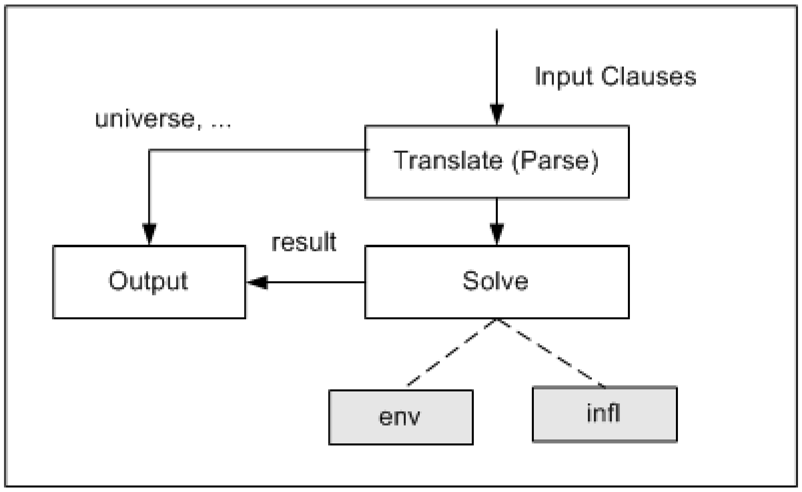


Figure 2.4: Program Structure of Succinct Solver

been given their values when introduced by the quantifiers, hence the use of partial environment;

- *infl*, which takes care of *consumer* registration. As mentioned before, the environment employed by the solver is a partial one, it may happen that some query $R(x_1, \ldots, x_k)$ inside a pre-condition fails to be satisfied at the given point in time, but may hold in the future when a new tuple $(a_1, \ldots, a_k)$ has been added to the interpretations of R. In this case, the current computation is suspended by constructing a *consumer* for R and recording it in *infl*.

Main Functions

In this section, we take a closer look at the core module *Solve* and briefly explain its working procedure.
The *Solve* module is primarily composed of two function, *execute* and *check*. The *check* function deals with preconditions, while the *execute* function deals with clauses.

Check Given a clause and a set of partial environments, *check* will determine whether ro not the clause will evaluate to true for all interpretations σ that are unifiable with a partial environment. This is done by recursively processing the clause and by updating the partial environments by collecting the binding of the instantiated variables. New bindings for variables are obtained at the queries $R(x_1, \ldots, x_k)$.

Execute The intention of *execute* is to update the global data structure corresponding to ρ (see Table 2.2). The function is executed in a recursive way that upon the termination, ρ contains all the computed interpretations for all predicate symbols. Furthermore, whenever the function adds a new tuple $(a_1, \ldots, a_k)$ to the interpretations of R, the list of consumers associated with R is activated, and thereby the corresponding computations are resumed.

Strategies for Getting Better Performance

It is known that the actual formulation of the analysis has a great impact on the execution time of the solver. In this section we provide two strategies to fasten the computation, both of them are done by tuning clauses, which amounts to transforming the clause given as input.
Experiments with the analysis of Discretionary Ambient [5] have also shown that both of these strategies may significantly improve the efficiency of solving clauses.

- the Order of the Parameters of Relations
  Succinct Solver uses prefix trees as an internal representation of relations[4]. Therefore preferences are given to certain query patterns and imposing a potential large penalty on other query patterns. A generalized strategy is to try to put the bound parameters ahead of unbound parameters.

  Example 2.7
  Assume that a relation $R(x_1, x_2)$ appears in a precondition where x_2 is always bound and x_1 is always unbound. In this case, query the inverse relation $R'(x_2, x_1)$ preferable over querying $R(x_1, x_2)$.

- the Order of Conjuncts in Preconditions
  Pre-conditions are evaluated from left to right and in the context of an environment, which describes successful bindings of variables. When checking a query to a relation R, the evaluation of remaining preconditions is performed for the new environments, which are made by unifying the current environment with elements in R. In this way, environments are propagated. If the unification fails, not further work will be done. Therefore, the earlier the unification fails, the less environments are propagated. One strategy is that putting queries, which restrict the variable binding most, at the front of preconditions will increase efficiency.

  Example 2.8

$$\begin{array}{ll} \forall x\ (R(x,a) \wedge R(b,x) \Rightarrow Q(x)) & \ldots\ (1) \\ \forall x\ (R(b,x) \wedge R(x,a) \Rightarrow Q(x)) & \ldots\ (2) \end{array}$$

  where a and b are constants and suppose relation R contains few elements with b

as the first component but more elements with a as the second component, like in example 2.1.
In this case, clause(2) will be more efficient than clause (1) as fewer environments are propagated from the first query to R.

Chapter 3

Bit-vector Framework

Program analysis concerns about some certain properties of programs. A lot of analyses is defined according to different requirements. Despite a great difference between each analysis, there are sufficient similarities to be abstracted in order to make an underlying framework possible. One advantage to construct such a framework is that people is able to concentrate on the general property of different analyses and design generic algorithms to solve the problems of the same family, i.e. one of such an algorithm is iterative-based worklist algorithm.

Generally speaking, there are two important ingredients in a framework,

Property Space L, which is used to represent the data flow information as well as the combination operator, $\sqcup$: $\mathcal{P}(L) \rightarrow L$, which combines information from different paths. This property space is actually a complete lattice. Furthermore, when implementing analyses, it is always requires that L satisfies the Ascending (or Descending) Chain Condition.

Transfer Function $f(l)$: $L \rightarrow L$ for l is a possible label of the program in question, which maps the information from one end of a block to the other end (i.e. from entry to exit or the reverse). We defined a set $\mathcal{F}$ over L containing all the $f(l)$, which has two basic properties:

- has an identity function: there exists an f such that $f(x) = x$ for all x
- closed under composition: if $f_1, f_2 \in \mathcal{F}$, then $f_1 \bullet f_2 \in \mathcal{F}$

Figure 3.1 shows the relationship between different frameworks. In this chapter, we shall focus on bit-vector framework and come to monotone framework in Chapter 4. One of the most important similarity between analyses within bit-vector framework is that they

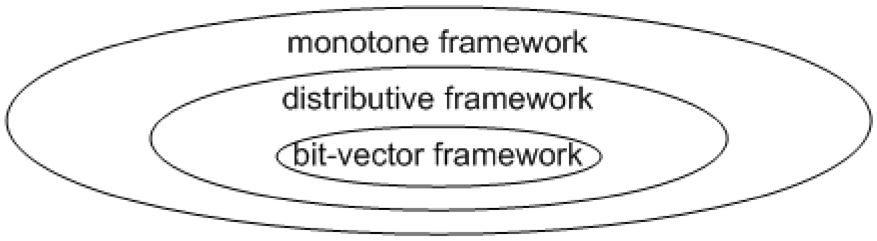


Figure 3.1: Relationship between different frameworks

share the same data flow equations:

$$Analysis_\circ(l) = \begin{cases} \imath & \text{if } l \in E \\ \sqcup\{Analysis_\bullet(l') \mid (l', l) \in F\} & \text{otherwise} \end{cases} \tag{3.1}$$

$$Analysis_\bullet(l) = (Analysis_\circ(l) \cap kill(l)) \cup gen(l) \tag{3.2}$$

where

- $\sqcup$ is $\cap$ or $\cup$
- F is either flow or reversed flow of a program
- E is the initial block or one of the final block of program
- $\imath$ specifies the initial or final analysis information
- $kill(l)$ and $gen(l)$ specify which information becomes invalid and valid after executing the block labeled l

The notions $_\circ$ and $_\bullet$ used above are defined as:

> *Analysis* $_\circ(l)$ contains as its elements the collected data-flow information immediately before executing elementary block l, while *Analysis* $_\bullet(l)$ contains the information immediately after executing l, where *Analysis* is the abbreviation of each analysis's name.

Having data flow equations, the remaining question is how to solve these equations to obtain an analysis result. This is actually done by Succinct Solver. Our task here is to develop program analyzers, which take WHILE programs as input and specify the equations in forms of ALFP clauses accepted by Succinct Solver. Therefore they serve as front-ends of the Succinct Solver.

Here we concern about four classical analyses within bit-vector framework: reaching definition analysis, very busy expression analysis, available expression analysis and live variable analysis. We will go into details of the first two analyses, which are regarded as complement, since one is *forward may* analysis whereas the other is *backward must* analysis.

To each analysis, both data flow approach and flow logic approach are applied, therefore two sets of ALFP clauses are generated. Experiments show that flow logic specifications could be solved more efficiently than the other one.

3.1 General Program Information

Although program analyzers vary from one to another according different specifications(e.g. different kill and gen set), they do share some common operations, which are used to extract basic information of the program to be analyzed. In this section, prior to defining specification and implementation of individual analyzer, we introduce a bit about these commonly used operations, all of which work in a recursive way. We present them in SML-style notation below.

3.1.1 Blocks

As mentioned in chapter 2, elementary blocks are either assignments, conditionals in branch statements (e.g. b in If $[b]^l$ Then ... Else ... and While $[b]^l$ Do ...), or Skip statements. Function *blocks*, applied to a statement, produces a set of elementary blocks.

blocks: $Stmt \rightarrow \mathcal{P}(Block)$

$$\begin{array}{lcl}
\text{blocks}([\text{Skip}]^l) & = & \{[\text{Skip}]^l\} \\
\text{blocks}([x := \ e]^l) & = & \{[x := e]^l\} \\
\text{blocks}(\text{If } [e]^l \text{ Then } s1 \text{ Else } s2) & = & \{[e]^l\} \ \cup \ \text{blocks}(s1) \ \cup \ \text{blocks}(s2) \\
\text{blocks}(\text{While } [e]^l \text{ Do } s) & = & \{[e]^l\} \ \cup \ \text{blocks}(s) \\
\text{blocks}(s1; \ s2) & = & \text{block}(s1) \ \cup \ \text{blocks}(s2) \\
\text{blocks}(\text{Begin } dec \ s \text{ End}) & = & \text{blocks}(s)
\end{array}$$

3.1.2 Initial label

Function *init* is used to acquire the initial label for a statement.

init: $Stmt \rightarrow Lab$

$$\begin{array}{lcl}
\text{init}([\text{Skip}]^l) & = & l \\
\text{init}([x := e]^l) & = & l \\
\text{init}(\text{If } [e]^l \text{ Then } s1 \text{ Else } s2) & = & l \\
\text{init}(\text{While } [e]^l \text{ Do } s) & = & l \\
\text{init}(s1;\ s2) & = & \text{init}(s1) \\
\text{init}(\text{Begin } dec\ s \text{ End}) & = & \text{init}(s)
\end{array}$$

Note that *init* returns a single element, which corresponding to the only entry point of a program.

3.1.3 Final labels

Function *final* acquires final labels for a statement.

final: $Stmt \rightarrow \mathcal{P}(Lab)$

$$\begin{array}{lcl}
\text{final}([\text{Skip}]^l) & = & \{l\} \\
\text{final}([x := e]^l) & = & \{l\} \\
\text{final}(\text{If } [e]^l \text{ Then } s1 \text{ Else } s2) & = & \text{final}(s1) \cup \text{final}(s2) \\
\text{final}(\text{While } [e]^l \text{ Do } s) & = & \{l\} \\
\text{final}(s1;\ s2) & = & \text{final}(s2) \\
\text{final}(\text{Begin } dec\ s \text{ End}) & = & \text{final}(s)
\end{array}$$

Whereas a sequence of statements can only has one single entry, it may have multiple exits. Thus, final returns a set of elements.

3.1.4 Labels

Function *labels* returns all the block labels of a given statement.

labels: $Stmt \rightarrow \mathcal{P}(Lab)$

$$\begin{array}{lcl}
\text{labels}([\text{Skip}]^l) & = & \{l\} \\
\text{labels}([x := e]^l) & = & \{l\} \\
\text{labels}(\text{If } [e]^l \text{ Then } s1 \text{ Else } s2) & = & \{l\} \cup \text{labels}(s1) \cup \text{labels}(s2) \\
\text{labels}(\text{While } [e]^l \text{ Do } s) & = & \{l\} \cup \text{labels}(s) \\
\text{labels}(s1;\ s2) & = & \text{labels}(s1) \cup \text{labels}(s2) \\
\text{labels}(\text{Begin } dec\ s \text{ End}) & = & \text{labels}(s)
\end{array}$$

3.1.5 Flows and Reverse Flows

flow, defined as

flow: $Stmt \rightarrow \mathcal{P}(Lab \times Lab)$

is a mapping from statement to sets of flows.

$$\begin{array}{lcl}
\text{flow}([\text{Skip}]^l) & = & \emptyset \\
\text{flow}([x := e]^l) & = & \emptyset \\
\text{flow}(\text{If } [e]^l \text{ Then } s1 \text{ Else } s2) & = & \text{flow}(s1) \,\cup\, \text{flow}(s2) \,\cup\, \{(l,\text{init}(s1)), (l,\text{init}(s2))\} \\
\text{flow}(\text{While } [e]^l \text{ Do } s) & = & \text{flow}(s) \,\cup\, \{(l,\text{init}(s))\} \,\cup\, \{(l',l) \mid l' \in \text{final}(s)\} \\
\text{flow}(s1;\ s2) & = & \text{flow}(s1) \,\cup\, \text{flow}(s2) \,\cup\, \{(l,\text{init}(s2)) \mid l \in \text{final}(s1)\} \\
\text{flow}(\text{Begin } dec\ s \text{ End}) & = & \text{flow}(s)
\end{array}$$

Reverse flows are required when formulating backward analyses. Function

flow^R: $Stmt \rightarrow \mathcal{P}(Lab \times Lab)$

is defined by:

$$\text{flow}^R(s) = \{(l,l') \mid (l',l) \in \text{flow}(s)\}$$

3.1.6 Variable List

var is an operation over statement to collect all the variables declared inside it.

var: $Stmt \rightarrow \mathcal{P}(Var)$

It is defined as,

$$\begin{array}{lcl}
\text{var}(\text{Begin } dec\ s \text{ End}) & = & \text{varD}(dec) \,\cup\, \text{var}(s) \\
\text{var}([\text{Skip}]^l) & = & \emptyset \\
\text{var}([x := e]^l) & = & \emptyset \\
\text{var}(\text{If } [e]^l \text{ Then } s1 \text{ Else } s2) & = & \text{var}(s1) \,\cup\, \text{var}(s2) \\
\text{var}(\text{While } [e]^l \text{ Do } s) & = & \text{var}(s) \\
\text{var}(s1;\ s2) & = & \text{var}(s1) \,\cup\, \text{var}(s2)
\end{array}$$

where varD is an auxiliary function operates on declarations.

varD: $Dec \rightarrow \mathcal{P}(Var)$

$$\begin{array}{lcl}
\text{varD}(\text{Var } x) & = & \{x\} \\
\text{varD}(\text{Array } A \text{ of } [n_1..n_2]) & = & \{(A,i) \mid n_1 \leqslant\ i\ \leqslant n_2\}
\end{array}$$

Example 3.1 Consider the following statement, *Sum*.

```
Begin  Var $x$
    $[x := 1]^1$;  While $[x <= 10]^2$ Do $[x := x + 1]^3$
End
```

We have init(*Sum*)=1, final(*Sum*)={2}, var(*Sum*)={x} and labels(*Sum*)={1,2,3}. The function flow produces the set

$$\{(1,2),(2,3),(3,2)\}$$

The flow^R is the set

$$\{(2,1),(3,2),(2,3)\}$$

3.1.7 Expressions and Sub-expressions

Function *exp* operates over statement to collect all the non-trivial expressions appeared in it. "Trivial" here is defined as an expression is trivial if it is a constant or a single variable[1]. In this thesis, we shall not be concerned with trivial expression: whenever we talk about some expressions we will always be talking about the non-trivial expressions.

The functionality of exp is exp: $Stmt \to \mathcal{P}(Exp)$ and it is defined as,

$$\begin{array}{lcl}
\text{exp(Begin } dec\ s \text{ End)} & = & \text{exp}(s) \\
\text{exp}([\text{Skip}]^l) & = & \emptyset \\
\text{exp}([x := e]^l) & = & \text{getExp}(e) \\
\text{exp(If } [e]^l \text{ Then } s1 \text{ Else } s2) & = & \text{getExp}(e) \ \cup\ \text{exp}(s1) \ \cup\ \text{exp}(s2) \\
\text{exp(While } [e]^l \text{ Do } s) & = & \text{getExp}(e) \ \cup\ \text{exp}(s) \\
\text{exp}(s1;\ s2) & = & \text{exp}(s1) \ \cup\ \text{exp}(s2)
\end{array}$$

where getExp(e) is used to determine whether e is an expression, namely whether it contains more than one variables connected by operators.

$$\begin{array}{lcl}
\text{getExp}(n) & = & \emptyset \\
\text{getExp}(x) & = & \emptyset \\
\text{getExp}(A[e]) & = & \emptyset \\
\text{getExp}(e_1\ op\ e_2) & = & \{e_1\ op\ e_2\}
\end{array}$$

Function subExp: $Exp \to \mathcal{P}(Exp)$ defined as,

$$\begin{array}{lcl}
\text{subExp}(n) & = & \emptyset \\
\text{subExp}(x) & = & \emptyset \\
\text{subExp}(A[e]) & = & \text{subExp}(e) \\
\text{subExp}(e_1\ op\ e_2) & = & \text{subExp}(e_1) \ \cup\ \text{subExp}(e_2) \ \cup\ \{e_1\ op\ e_2\} \ \cup\ \{e_2\ op\ e_1\}
\end{array}$$

is used to collect all the sub-expressions of a given expression. Note that, for simplicity's sake, we define sub-expression in a strict way. For example, $a + b \notin \text{subExp}(a + c + b)$.

[1]either a simple variable or an array element

3.1.8 Free Simple Variable and Free Array Name

Function *Fv* is a mapping from an expression to the free simple variables contained in that expression. It is defined as:
Fv: $Exp \rightarrow \mathcal{P}(Var)$

$$\begin{array}{lcl} \mathrm{Fv}(n) & = & \emptyset \\ \mathrm{Fv}(x) & = & \{x\} \\ \mathrm{Fv}(A[e]) & = & \mathrm{Fv}(e) \\ \mathrm{Fv}(e_1 \ op \ e_2) & = & \mathrm{Fv}(e_1) \ \cup \ \mathrm{Fv}(e_2) \end{array}$$

Note that in the case a simple variable appears in the index of an array element, it is also regarded as a free simple variable.

Function *FAv* is a mapping from an expression to the free array names[2] contained in the expression, which is defined as:

FAv: $Exp \rightarrow \mathcal{P}(Arr)$

$$\begin{array}{lcl} \mathrm{FAv}(n) & = & \emptyset \\ \mathrm{FAv}(x) & = & \emptyset \\ \mathrm{FAv}(A[e]) & = & \{A\} \ \cup \ \mathrm{FAv}(e) \\ \mathrm{FAv}(e_1 \ op \ e_2) & = & \mathrm{FAv}(e_1) \ \cup \ \mathrm{FAv}(e_2) \end{array}$$

Example 3.2 Consider an expression $e = y + A[3] + A[x]$, where x and y are variables.
We have
$\mathrm{subExp}(e) = \{y + A[3] + A[x], y + A[3], A[3] + A[x], \}$
$\mathrm{Fv}(e) = \{y, x\}$
$\mathrm{FAv}(e) = \{A\}$

3.1.9 Notation for Program

In the later sections of this chapter and the consequent chapters, we shall use the notation S_* to represent the statement of the program that we are analyzing, Block_* to represent the elementary blocks in S_*, Lab_* to represent the labels appearing in S_*, Var_* to represent the variables appearing in S_* and Exp_* to represent the set of non-trivial expressions in

[2]names of arrays appear in an expression

S_*, i.e.

$$\begin{array}{lcl} \text{Block}_* & = & \text{blocks}(S_*) \\ \text{Lab}_* & = & \text{labels}(S_*) \\ \text{Var}_* & = & \text{var}(S_*) \\ \text{Exp}_* & = & \exp(S_*) \end{array}$$

3.2 Reaching Definition Analysis

A definition of a variable x is a statement that assigns, or may assign, a value to x. A definition d reaches point p if there is a path from the point immediately following d to p, such that d is not redefined along that path. Intuitively, if a definition d of variable x reaches point p, then d might be the place at which the value of x used at p might last have been defined.

Reaching Definition analysis(RD for short) is used to indicate:

at each program point, in which elementary block is each variable defined.

The information gained from this analysis is used, say, in calculating ud and du chains as well as in Constant Folding transformation to determine which variable has a constant value.

Example 3.3 Consider the following program:

$$[x := 1]^1;\ [y := 2]^2;\ \text{If } [x > 0]^3 \text{ Then } [x := x + 1]^4 \text{ Else } [y := y * x]^5$$

Here assignments labeled 1 and 2 reach the entry of 4 and 5.

3.2.1 Data Flow Approach

The analysis itself is defined by the pair of functions $\text{RD}_\circ$ and $\text{RD}_\bullet$ mapping labels to sets of pairs of variables and labels:

$$\text{RD}_\circ,\text{RD}_\bullet : \quad \text{Lab}_* \rightarrow \mathcal{P}(\text{Var}_* \times \text{Lab}^0_*)$$

In order to deal with uninitialized variables, we use a special label "Lab0" (at the very beginning point, each variable is defined at lab0) and set $\text{Lab}^0_* = \text{Lab}_* \cup \{\text{lab0}\}$.
It's property space is a complete lattice

$$(\mathcal{P}(\text{Var}_* \times \text{Lab}^0_*), \ \subseteq)$$

and the transfer function is

$$f^{RD}_l(\text{RD}_\circ(l)) \ = \ (\text{RD}_\circ(l) \setminus kill(l)) \ \cup \ gen(l)$$

Kill Component

The kill component are used to indicate at certain point of the program, which definition for a variable is not valid any more (e.g. the variable has been re-defined).

Table 3.1 shows for each kind of elementary block what the kill components are.

1	$[x := e]^l$	$\{(x, lab0)\} \cup$ $\{(x, l') \mid B^{l'}$ is an assignment to x in $\mathrm{S}_*\}$
2	$[A[n] := e]^l$	$\{(A[n], lab0)\} \cup$ $\{(A[n], l') \mid B^{l'}$ is an assignment to $A[n]$ or $A[e]$ in $\mathrm{S}_*\}$
3	$[A[e] := e']^l$	$\emptyset$
4	$[Skip]^l$	$\emptyset$
5	$[e]^l$	$\emptyset$

Table 3.1: Kill Component of RD Analysis

1. $[x := e]^l$
   In case, the current block is an assignment to a variable, x, we
   - kill $(x, lab0)$, since we know x is no longer defined outside the program.
   - kill $(x,\ l')$, when $B^{l'}$ has the form $[x := e]^{l'}$. Because now x is only defined in block l

2. $[A[n] := e]^l$
   In case the current block is an assignment to an array, A, with a constant index value, n, we
   - kill $(A[n], lab0)$ and $(A[n],\ l')$ If $\mathrm{B}^{l'}$ has the form $[A[n] := e]^{l'}$. Since at this point, $A[n]$ is only defined in elementary block l.
   - kill $(A[n],\ l')$ If $B^{l'}$ has the form $[A[e] := e']^{l'}$. Because now no matter what the value does x have, $A[n]$ must be defined in block l, thus we can safely kill $(A[n],\ l')$.

3. $[A[e] := e']^l$
   In case the current block is an assignment to an array, A, with an index, e, we kill nothing. The reason is the value of e is unknown, therefore it is unsafe to kill anything.

4. $[Skip]^l$ and $[e]^l$
   Not necessary kill any thing, since no variable is defined or redefined here.

Gen component

Gen component is used to indicate which definition of certain variable is generated at an elementary block l.

1	$[x := e]^l$	$\{(x, l)\}$
2	$[A[n] := e]^l$	$\{(A[n], l)\}$
3	$[A[e] := e']^l$	$\{(A[i], l) \mid (A, i) \in Var_*\}$
4	$[Skip]^l$	$\emptyset$
5	$[e]^l$	$\emptyset$

Table 3.2: Gen Component of RD Analysis

1. $[x := e]^l$
   In case, the current block is an assignment to a variable, x, we generate (x, l), since now x is defined in block l.
2. $[A[n] := e]^l$
   In case the current block is an assignment to an array, A, with a constant index value, n, we generate ($A[n]$, l), since now $A[n]$ is defined in block l.
3. $[A[e] := e']^l$
   In case the current block is an assignment to an array, A, with a variable index, e, we generate ($A[i]$,l), for $(A, i) \in$ Var$_*$. Because at this point, all these array elements may be defined in block l.
4. $[Skip]^l$ and $[e]^l$
   Not necessary gen any thing, since no variable is defined here.

Data Flow Equations

This analysis is a *forward may* analysis, calculating the smallest set satisfying RD$_\circ$. The data flow equations of reaching definition are listed in figure 3.2.
Equation (1) takes care the initialization of entry information: all the variables are set to defined outside program (here we use lab0 for that purpose) at the initial block. Equation (2) says that, for non-initial blocks, information come from different paths are combined together at the entry point. $\cup$ is used here, since reaching definition is a *may* analysis(over approximation). Transfer function f^{RD} is specified in equation (3).

$$\begin{array}{lcl} \mathrm{RD}_\circ(l) & = & \left\{ \begin{array}{ll} \{(x,\ lab0) \mid x \in \mathrm{Var}_*\} & \text{if } l = \mathit{init}(\mathrm{S}_*) \quad (1) \\ \bigcup\{\mathrm{RD}_\bullet(l') \mid (l', l) \in \mathit{flow}(\mathrm{S}_*)\} & \text{otherwise} \qquad (2) \end{array} \right. \\ \mathrm{RD}_\bullet(l) & = & (\mathrm{RD}_\circ(l) \setminus \mathit{kill}_{RD}(B^l)) \cup \mathit{gen}_{RD}(B^l) \quad \text{where } B^l \in \mathrm{Block}_* \quad (3) \end{array}$$

Figure 3.2: Data Flow Equations of RD Analysis

Implementation

By now, we have the specification the RD analysis. The remaining question is to "translate" them into ALFP clauses. When implementing this RD analyzer, we employ the following predicates,

Predicate	**Meaning**
INIT(l)	l is the initial label of program
FLOW(l_1,l_2)	there exists a flow from l_1 to l$_2$
VAR(x)	x is a variable in program
RDGEN(l_1, x, l_2)	at block l_1, (x, l_2) is generated
RDKILL(l_1, x, l_2)	at block l_1, (x, l_2) is killed
RDIN(l_1, x, l_2)	at the entry of l_1, x is defined at l_2, viz. $(x, l_2) \in \mathrm{RD}_\circ(l_1)$
RDOUT(l_1, x, l_2)	at the exit of l_1, x is defined at l_2, viz. $(x, l_2) \in \mathrm{RD}_\bullet(l_1)$

Generate Clauses
Clauses are generated using functions listed in chapter 3.1 as well as the specifications defined above.

INIT(l) holds if init(S$_*$)= l

FLOW($l1, l2$) holds if $(l1, l2)$ ∈flow(S$_*$)

VAR(x) holds if x ∈Var$_*$
VAR($A'i$) holds if (A, i) ∈Var$_*$
In case of array, all its elements whose indexes fall within the array bounds are generate as variables. Additionally, we use $A'i$ to represent $A[i]$ in ALFP clause because symbol "[" and "]" are not recognized by Succinct Solver.

RDGEN(l, x, l') ⇔ $(x, l') \in gen_{RD}(B^l)$ where B^l ∈Block$_*$
RDGEN($l, A'i, l'$) ⇔ $(A[i], l') \in gen_{RD}(B^l)$ where B^l ∈Block$_*$

RDKILL$(l, x, l') \Leftrightarrow (x, l') \in kill_{RD}(B^l)$ where $B^l \in$Block$_*$
RDKILL$(l, A'i, l') \Leftrightarrow (A[i], l') \in kill_{RD}(B^l)$ where $B^l \in$Block$_*$
RDGEN and RDKILL are generated according to the specifications of gen and kill. For block labeled l, RDGEN(l, x, l') holds if (x, l') is generated by block l and RDKILL(l, x, l') holds if (x, l') is killed by block l.

Finally, data flow equations are translated to the following three clauses,
$\forall l\ \forall x$ (INIT$(l)\wedge$VAR$(x)) \Rightarrow$RDIN$(l, x, lab0)$
At the initial point, all the variables are defined at $lab0$.

$\forall l\ \forall x\ \forall l1\ \forall l2$ (RDOUT$(l1, x, l2)\wedge$FLOW$(l1, l) \Rightarrow$RDIN$(l, x, l2))$
If flow$(l1, l)$ exists and at exit of $l1$ x is defined at $l2$ then at entry of l, x is still defined at $l2$.

$\forall l\ \forall x\ \forall l1$ (RDIN$(l, x, l1) \wedge \neg$RDKILL$(l, x, l1))\vee$RDGEN$(l, x, l1) \Rightarrow$RDOUT$(l, x, l1)$
At the exit of block l, variable x is defined at $l1$ providing that either $(x, l1)$ is generated by block l or at the entry of l x is defined at $l1$ and $(x, l1)$ is not killed by block l.

Example 3.4 Consider the program piece,
$[A[1] := 2]^1$; While $[x < 5]^2$ Do (If $[x > 0]^3$ Then $[A[x] := x]^4$ Else $[x := A[2]]^5$)
where array A is declared by $Array\ A\ of$ [1..2]. Data flow specifications give rise to the following clause

INIT(L1)$\wedge$
VAR$(x)\wedge$VAR$(A'1)\wedge$VAR$(A'2)\wedge$
FLOW(L1,L2)$\wedge$FLOW(L2,L3)$\wedge$FLOW(L3,L4)$\wedge$FLOW(L3,L5)$\wedge$FLOW(L4,L2)$\wedge$FLOW(L5,L2)$\wedge$

RDGEN(L1,A'1,L1)$\wedge$RDKILL(L1,A'1,L2)$\wedge$RDKILL(L1,A'1,L4)$\wedge$RDKILL(L1,A'1,Lab0)
RDGEN(L4,A'1,L4)$\wedge$RDGEN(L4,A'2,L4)$\wedge$
RDGEN(L5,x,L5)$\wedge$RDKILL(L5,x,Lab0)

$(\forall l$ INIT$(l)\Rightarrow(\forall x$ VAR$(x)\Rightarrow$RDIN$(l, x,$Lab0$)))\wedge$
$(\forall l\ \forall x\ \forall l1$ (RDIN$(l, x, l1)\wedge\neg$RDKILL$(l, x, l1))\vee$RDGEN$(l, x, l1)\Rightarrow$RDOUT$(l, x, l1))\wedge$
$(\forall l\ \forall l1\ \forall l2\ \forall x$ RDOUT$(l1, x, l)\wedge$FLOW$(l1, l2)\Rightarrow$RDIN$(l2, x, l))$

where Li represents label i. The result could be summarize to the below table.

$\text{RD}_\circ$	x	$A[1]$	$A[2]$	$\text{RD}_\bullet$	x	$A[1]$	$A[2]$
L1	Lab0	Lab0	Lab0	L1	Lab0	L1	Lab0
L2	Lab0 L5	L1 L4	Lab0 L4	L2	Lab0 L5	L1 L4	Lab0 L4
L3	Lab0 L5	L1 L4	Lab0 L4	L3	Lab0 L5	L1 L4	Lab0 L4
L4	Lab0 L5	L1 L4	Lab0 L4	L4	Lab0 L5	L1 L4	Lab0 L4
L5				L5	L5	L1 L4	Lab0 L4

3.2.2 Flow Logic Approach

Determining data flow equations using *kill* and *gen* components is a traditional approach for specifying reaching definition analysis. It is classical, yet not very efficient when implemented using the Succinct Solver, partial because of a large amount of unused elements in *kill* set. Examining the following program piece,

$$[x := \ 1]^1;\ [x := \ 2]^2;\ [x := \ 3]^3;\ [x := \ 4]^4;\ [x := \ 5]^5$$

the *kill* set for x at block 2 is $\{1, 2, 3, 4, 5\}$, while according to the program structure, it is sure that x in block 2 can never be defined at $\{3, 4, 5\}$. Therefore, solving clauses RDKILL(L2,x,L3), RDKILL(L2,x,L4) and RDKILL(L2,x,L5) becomes extra tasks for Succinct Solver.

In this section, we present another approach for handling this analysis, the flow logic approach, which is not only useful for Reaching Definition analysis but also applicable to all the analyses within bit-vector framework. Comparisons about the efficiencies of these two approaches are given at the end of each section.

WHILE language is known to be well-structure shown by its grammar, in the sense that there is no abnormal exit in program(like GOTO statement). Consequently, it is naturally to consider about analyzing programs with respect to their control flow structure.

An example of the control flow structure is shown in Figure 3.3. Note that s, $s1$, $s2$ in the figure maybe compositional statements, hence their structures are induced by applying the same rule.

Recall the data flow equations at the beginning of this chapter. These equations inspire us to split the flowing information into two parts, inter-statement and intra-statement, which takes care of the information flowing within a node and along an edge, respectively. More precisely they specify the relationship between $\text{RD}_\circ(l)$ and $\text{RD}_\bullet(l)$(for elementary

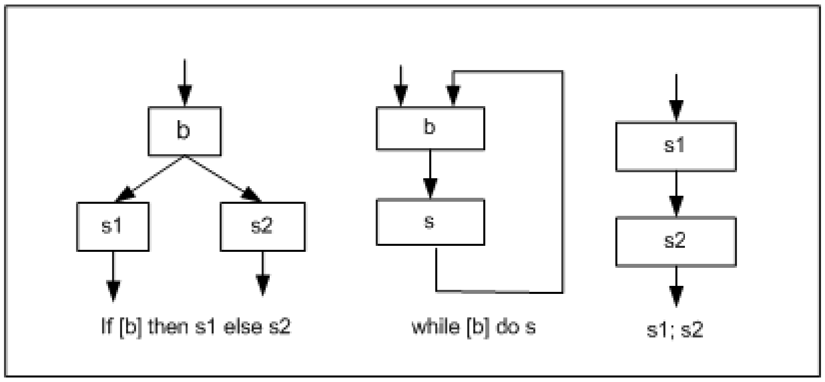


Figure 3.3: Flow Relation of Forward Analysis

block l) as well as between $\mathrm{RD}_\bullet(l)$ and $\mathrm{RD}_\circ(l')$(for $\mathrm{flow}(l, l')$).

Our approach begins by constructing a flow graph for a given program in the way that,

- there is a node for each elementary block
- there is an edge from block l to block l', if $\mathrm{flow}(l, l')$ exists.

The two flowing information categories are handled separately by,

- for each node, figure out which information held at the entry is also going to be held at the exit and which information is newly revealed to be held by analyzing the current block, therefore it holds at the exit.
- for each edge from l to l', specify the relationship between the starting and the ending point. There are two cases according to the character of the performed analysis, either the information held the starting point is contained in the ending point or the other way around, corresponding to *may* analysis or *must* analysis.

Specification

Now we can define the shape of the information that is statically collected from a given program and then used to check certain property of programs. The result of our analysis here is a pair ($\mathrm{RD}_\circ$, $\mathrm{RD}_\bullet$), where they are abstract caches associating each labeled program point and variable with its definition labels.

$$\mathrm{RD}_\circ, \mathrm{RD}_\bullet \in \widehat{Cache} = (\mathrm{Lab}_* \times \mathrm{Var}_*) \to \mathcal{P}(Lab)$$

For the formulation of the new approach for reaching definition analysis, we shall write

$$(\mathrm{RD}_\circ, \mathrm{RD}_\bullet) \models s \text{ and } (\mathrm{RD}_\circ, \mathrm{RD}_\bullet) \models B$$

for when ($\text{RD}_\circ$, $\text{RD}_\bullet$) is an acceptable analysis of statement s or elementary block B. Therefore, the functionality of relation "$\models$" is

$$\models : (\widehat{Cache} \times \widehat{Cache} \times (Stmt \cup Block)) \rightarrow \{\text{true, false}\}$$

The specifications of the analysis for elementary block and statement are listed in Table 3.3, where $\text{RD}_\bullet(l) \subseteq \text{RD}_\circ(l')$ is defined by

$$\forall\ x\ \text{RD}_\bullet(l, x) \subseteq \text{RD}_\circ(l', x)$$

$(\text{RD}_\circ, \text{RD}_\bullet) \models [x := e]^l$	iff	$l \in \text{RD}_\bullet\ (l, \text{x}) \land$ $\forall$ y y$\neq$x $\Rightarrow \text{RD}_\circ(l, \text{y}) \subseteq \text{RD}_\bullet(l, \text{y})$
$(\text{RD}_\circ, \text{RD}_\bullet) \models [A[n] := e]^l$	iff	$l \in \text{RD}_\bullet\ (l, \text{A[n]}) \land$ $\forall$ y y$\neq$A[n] $\Rightarrow \text{RD}_\circ(l, \text{y}) \subseteq \text{RD}_\bullet(l, \text{y})$
$(\text{RD}_\circ, \text{RD}_\bullet) \models [A[e] := e']^l$	iff	$\forall$i (A,i)$\in\text{Var}_*$ $\Rightarrow l \in \text{RD}_\bullet\ (l, \text{A[i]}) \land$ $\text{RD}_\circ(l) \subseteq \text{RD}_\bullet(l)$
$(\text{RD}_\circ, \text{RD}_\bullet) \models [\text{Skip}]^l$	iff	$\text{RD}_\circ(l) \subseteq \text{RD}_\bullet(l)$
$(\text{RD}_\circ, \text{RD}_\bullet) \models [\text{e}]^l$	iff	$\text{RD}_\circ(l) \subseteq \text{RD}_\bullet(l)$
$(\text{RD}_\circ, \text{RD}_\bullet) \models \text{s}_1; \text{s}_2$	iff	$(\text{RD}_\circ, \text{RD}_\bullet) \models \text{s}_1 \land$ $(\text{RD}_\circ, \text{RD}_\bullet) \models \text{s}_2 \land$ $\forall\ l\ l\in\text{final}(\text{s}_1) \Rightarrow \text{RD}_\bullet(l) \subseteq \text{RD}_\circ(\text{init}(\text{s}_2))$
$(\text{RD}_\circ, \text{RD}_\bullet) \models \text{If } [\text{e}]^l \text{ Then s}_1 \text{ Else s}_2$	iff	$(\text{RD}_\circ, \text{RD}_\bullet) \models \text{s}_1 \land$ $(\text{RD}_\circ, \text{RD}_\bullet) \models \text{s}_2 \land$ $(\text{RD}_\circ, \text{RD}_\bullet) \models \text{b} \land$ $\text{RD}_\bullet(l) \subseteq \text{RD}_\circ(\text{init}(\text{s}_1)) \land$ $\text{RD}_\bullet(l) \subseteq \text{RD}_\circ(\text{init}(\text{s}_2))$
$(\text{RD}_\circ, \text{RD}_\bullet) \models \text{While } [\text{e}]^l \text{ Do s}$	iff	$(\text{RD}_\circ, \text{RD}_\bullet) \models \text{s} \land$ $(\text{RD}_\circ, \text{RD}_\bullet) \models \text{b} \land$ $\text{RD}_\bullet(l) \subseteq \text{RD}_\circ(\text{init}(\text{s})) \land$ $\forall l'\ l'\in\text{final(s)} \Rightarrow \text{RD}_\bullet(\text{l'}) \subseteq \text{RD}_\circ(l))$

Table 3.3: Control Flow Analysis of RD for Elementary Blocks and Statements

The first five rules are responsible for linking the current label into the abstract caches. For $[x := e]^l$, we claim that at the exit of l, the definition point of x is l. All the other variable except x preserve their definition information, because they are not touched in block

l. Clauses for block $[A[n] := e]^l$ are defined in the same way as above. For $[A[e] := e']^l$, since the value of e can not be determined, therefore we make all the elements of array A being defined at l, other variables are still defined at the original places.

The last three rules contain "recursive calls" demanding that sub-statements must be analyzed in consistent ways using ($\mathrm{RD}_\circ$, $\mathrm{RD}_\bullet$). Clause $\mathrm{RD}_\bullet(l) \subseteq \mathrm{RD}_\circ(l')$ is defined in the case *flow*(l, l') exists. The idea is reaching definition analysis is a *may* analysis, $\mathrm{RD}_\circ(l')$ equals to the *joining*($\cup$) of the information from all its predecessors. Therefore, each part($\mathrm{RD}_\bullet(l)$) is a subset of the whole($\mathrm{RD}_\circ(l')$).

Implementation

Predicates used in implementing the flow logic specifications are INIT, VAR, RDIN and RDOUT, the meanings of which are exactly the same as they are in the previous approach.

At the very beginning, all the variables are set to be defined at lab0 by the clause

$$\forall l \ \mathrm{INIT}(l) \Rightarrow (\forall x \ \mathrm{VAR}(x) \Rightarrow \mathrm{RDIN}(l, x, Lab0))$$

Clauses for elementary blocks could be translated directly from the specifications. For statements, we specify the below clause for flow(l_1, l_2)

$$\forall x \ \forall l \ \mathrm{RDOUT}(l_1, x, l) \Rightarrow \mathrm{RDIN}(l_2, x, l)$$

Example 3.5 Returning to the program in Example 3.4. Flow logic specifications give rise to the clause

INIT(L1)$\wedge$
VAR(x)$\wedge$VAR($A'1$)$\wedge$VAR($A'2$)$\wedge$
($\forall l$ INIT(l)$\Rightarrow$($\forall x$ VAR(x)$\Rightarrow$RDIN(l, x,Lab0)))$\wedge$

RDOUT(L1,$A'1$,L1)$\wedge$
($\forall x$ $\forall l$ RDIN(L1,x, l)$\wedge x \neq A'1$ $\Rightarrow$RDOUT(L1,x, l))$\wedge$

($\forall x$ $\forall l$ RDIN(L2,x, l)$\Rightarrow$RDOUT(L2,x, l))$\wedge$

($\forall x$ $\forall l$ RDIN(L3,x, l)$\Rightarrow$RDOUT(L3,x, l))$\wedge$

RDOUT(L4,$A'1$,L4)$\wedge$

RDOUT(L4,$A'2$,L4)$\wedge$
($\forall x$ $\forall l$ RDIN(L4,x, l)$\Rightarrow$RDOUT(L4,x, l))$\wedge$

RDOUT(L5,x,L5)$\wedge$
($\forall y$ $\forall l$ RDIN(L5,y, l)$\wedge y \neq x$ $\Rightarrow$RDOUT(L5,y, l))$\wedge$

($\forall x$ $\forall l$ RDOUT(L1,x, l)$\Rightarrow$RDIN(L2,x, l))$\wedge$
($\forall x$ $\forall l$ RDOUT(L2,x, l)$\Rightarrow$RDIN(L3,x, l))$\wedge$
($\forall x$ $\forall l$ RDOUT(L3,x, l)$\Rightarrow$RDIN(L4,x, l))$\wedge$
($\forall x$ $\forall l$ RDOUT(L3,x, l)$\Rightarrow$RDIN(L5,x, l))$\wedge$
($\forall x$ $\forall l$ RDOUT(L4,x, l)$\Rightarrow$RDIN(L2,x, l))$\wedge$
($\forall x$ $\forall l$ RDOUT(L5,x, l)$\Rightarrow$RDIN(L2,x, l))

The result of solving the above clause is exactly the same as the one in Example 3.4.

3.2.3 Comparison of Two Approaches

In order to do the comparison, we create a benchmark, namely a scalable program a-m and develope two sets of clauses according to both data flow and flow logic specifications mentioned before. The function of program a-m is to calculate the sums of counterparts in m arrays, where each array has only two elements.

The measured execution times of Succinct Solver running on both sets of clauses are shown in Figure 3.4. The plot is made using logarithmic scales on both axes, which shows the running times of the Succinct Solver as a function of the size of the programs. Thus a polynomial dependency will result in a straight line. Higher degrees of the polynomial will appear as a steeper gradient. Hence, two polynomial with the same degree will appear as parallel lines. The polynomial with the smallest coefficient will appear as the lowest of the two lines. We refer to Appendix A for detail information about benchmarks as well as the strategies for timing the experiments and presenting the result. Appendix B contains comparisons based on another benchmark.

From Figure 3.4 we can see that Succinct Solver v2.0 solves the ALFP clause generated using flow logic approach much efficiently than the one generated using the classical data flow approach by degrees of the complexity polynomial. This could be explained by the range for selecting the kill components. As mentioned in the beginning of chapter 3.2.2, data flow approach always generates a large amount of kill components and in some cases numbers of them are not relevant to the block under analyzing. These extra kill components waste the final computation time of Succinct Solver. While in flow logic ap-

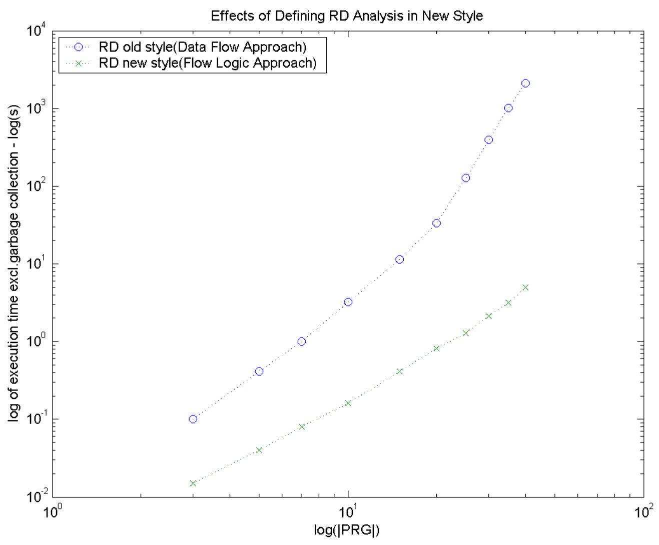


Figure 3.4: Comparison of execution times on a-m

proach, no such kill component exists in the sense that information at exit of each block comes from the information at the entry of the block excluding those killed by the block. An extreme example could be

$$[x := \;1]^1;\; [x := \;2]^2;\; [x := \;3]^3;\; [x := \;4]^4;\; [x := \;5]^5$$

The kill components for block 1 are RDKILL(L1,x,L1), RDKILL(L1,x,L2), RDKILL(L1,x,L3), RDKILL(L1,x,L4) and RDKILL(L1,x,L5). Information held at the exit of block 1 is specified by the clause

$$\forall l\ \forall x\ \forall l1\ (\mathrm{RDIN}(l,x,l1) \wedge \neg\mathrm{RDKILL}(l,x,l1)) \vee \mathrm{RDGEN}(l,x,l1) \Rightarrow \mathrm{RDOUT}(l,x,l1)$$

Thus all the interpretations of predicate RDKILL have to be examined to ensure that tuple $(1, x, l1)$ is not one of them when it is a interpretation of RDIN.

While in flow logic approach information at exit of block 1 is specified by the clause

$$\begin{aligned}&\mathrm{RDOUT}(\mathrm{L1}, x, \mathrm{L1}) \wedge \\ &(\forall y\ \forall l\ \mathrm{RDIN}(\mathrm{L1}, y, l) \wedge y \neq x \Rightarrow \mathrm{RDOUT}(\mathrm{L1}, y, l))\end{aligned}$$

The calculation for the above clause is simpler: the definition point of each the variable is not changed except the variable is x, in which case its definition point is L1 now. Thus the redundant check in solving data flow clause is avoided.

In benchmark a-m, 3/4 statements of the program are used to redefined variables. As program is enlarged, more and more solving time is spend on examining whether one interpretation of RDKILL exists or not, which results in data flow approach is degrees of complexity polynomial worse than flow logic approach, as shown in Figure 3.4.

3.3 Very Busy Expression Analysis

In this section, we present very busy expression analysis (VB for short) using both data flow and flow logic approaches.
An expression is *very busy* at the exit of a block if, no matter what path is taken from the block, the expression must always be used before any of the variables occurring in it are redefined. The aim of very busy expression analysis is to determine:

> For each program point, which expressions *must* be very busy at the exit from the point.

One possible optimization based on the result of very busy expression analysis is code hoisting.

Example 3.6 Consider the program:

$$\text{If } [a > b]^1 \text{ Then } [x := a + b]^2 \text{ Else } [y := a + b]^3$$

Expression $a + b$ is very busy at the start of the conditional. Therefore it's value could be evaluated before block 1 and stored for later use. Optimization code hoisting will do this job.

3.3.1 Data Flow Approach

The analysis is defined by the pair of function $\text{VB}_\circ$ and $\text{VB}_\bullet$ mapping labels to sets of expressions:

$$\text{VB}_\circ, \text{VB}_\bullet : \text{Lab}_* \rightarrow \mathcal{P}(Exp)$$

It's property space is a complete lattice

$$(\mathcal{P}(\text{Exp}_*), \ \supseteq)$$

and the transfer function is

$$f_l^{VB}(\text{VB}_\circ(l)) \ = \ (\text{VB}_\circ(l) \setminus kill(l)) \ \cup \ gen(l)$$

Kill Component

The kill component are used to indicate at certain point of the program, which expression is no longer valid (e.g. its value has been changed).
Table 3.4 shows for each kind of elementary block what the kill components are.

1	$[x := e]^l$	$\{e_1 \in \text{Exp}_* \mid x \in \text{Fv}(e_1)$
2	$[A[n] := e]^l$	$\{e_1 \in \text{Exp}_* \mid A \in \text{FAv}(e_1)$
3	$[A[e] := e']^l$	$\{e_1 \in \text{Exp}_* \mid A \in \text{FAv}(e_1)$
4	$[Skip]^l$	$\emptyset$
5	$[e]^l$	$\emptyset$

Table 3.4: Kill Component of VB Analysis

1. $[x := e]^l$
   In this case, the kill component includes the expressions appear in program, which contain assignee x or any array element indexed by x, since the change of x's value will result in the change of the value of $A[x]$.
2. $[A[n] := e]^l$
   In this case, the kill component includes the expressions which contain any array element indexed by n or an expression. Expressions containing $A[n]$ should be killed as its value has changed and expressions containing $A[e'']$ should be killed because its not sure that whether it is equal to $A[n]$, hence kill it satisfies the requirement of a *must* analysis.
3. $[A[e] := e']^l$
   In this case each expression containing array element with an expression index should be killed, because its value *may* changed.
4. $[Skip]^l$ and $[e]^l$
   Not necessary to kill any thing, since no variable's value has changed here.

Gen Component

Gen component is used to indicate which expression becomes valid, meaning that its value has been computed at an elementary block l. Note that in row 1, 2, 3 and 5, the subexpressions of e become valid, since their values have been computed.

1	$[x := e]^l$	subExp(e)
2	$[A[n] := e]^l$	subExp(e)
3	$[A[e_1] := e]^l$	subExp(e)
4	$[Skip]^l$	$\emptyset$
5	$[e]^l$	subExp(e)

Table 3.5: Gen Component of VB Analysis

Data Flow Equations

Very busy expression analysis is one of the four classical analyses within bit-vector framework, meaning that it shares the standard data flow equations. Since it is a *backward must* analysis, we are interested in the largest sets satisfying the equation for $\mathrm{VB}_\bullet$.

$$\begin{array}{lcl} \mathrm{VB}_\bullet(l) & = & \begin{cases} \emptyset & \text{if } l \in \text{final}(\mathrm{S}_*) \\ \bigcap\{\mathrm{VB}_\circ(l') \mid (l',l) \in \text{flow}^R(\mathrm{S}_*)\} & \text{otherwise} \end{cases} \\ \mathrm{VB}_\circ(l) & = & (\mathrm{VB}_\bullet(l) \setminus kill_{VB}(B^l)) \cup gen_{VB}(B^l) \quad \text{where } B^l \in \text{Block}_* \end{array}$$

Figure 3.5: Data Flow Equations of VB Analysis

Implementation

Specifying *must* analysis is a bit more complicated than *may* analysis, because:
Data flow equations of VB analysis demand that

$$\mathrm{VB}_\bullet(l) = \bigcap\{\mathrm{VB}_\circ(l') \mid (l',l) \in \text{flow}^R(\mathrm{S}_*)\} \quad \text{if } l \notin \text{final}(\mathrm{S}_*)$$

This can not be implemented straight-forwardly when there is a dependency cycle existing among blocks. Assume two reverse flow pairs exist, say (l_1, l_2) and (l_2, l_1), which may happen for a While statement. Information at exit of l_1 depends on the availability of information at entry of l_2 and the same thing for l_2. One of them is needed to become available in order for the other one to be available. Thus a dead lock happens here. Therefore, rather than the above equation, we re-write it to:

$$\mathrm{VB}_\bullet(l) \supseteq \mathrm{VB}_\circ(l') \quad \text{if } (l',l) \in \text{flow}^R(\mathrm{S}_*)$$

which corresponding to the clause

$$\forall e \; e \in \mathrm{VB}_\circ(l') \wedge (l',l) \in \text{flow}^R(\mathrm{S}_*) \Rightarrow e \in \mathrm{VB}_\bullet(l)$$

By doing in this way, the dependency cycle is broken: information could be transferred smoothly between blocks. But this also allows the transference of unwanted information, since the expression e held at the entry of l' may not hold at entry of other predecessors of block l. Thus the second step is to select and discard all these unwanted information.

As VB analysis is known to be a *must* analysis, it satisfies the descending chain condition, meaning that there exists n such that $l'_{n+1} \sqsubseteq l'_n$. This can be understand as information at each program point becomes less and less as the computation progresses. Now the problem arise; it is natural to create interpretations for predicates but impossible to remove certain interpretation. This problem could be solved by employing a flag, say Valid and Invalid to indicate whether an information still holds. Therefore flag is set to Valid when an interpretation is generated and whenever an interpretation should be removed, we set its flag to Invalid.

We shall use the following predicates in the clause generated according to data flow specification. Predicates used in clauses and their meanings are listed in the below table.

Predicate	**Meaning**
EXP(e)	e is an expression
PLUS(x, y)	expression $x + y$
RFLOW(l_1,l_2)	there is a reverse flow from l_1 to l_2
VBGEN(l_1, e)	at block l_1, e is generated
VBKILL(l_1, e)	at block l_1, e is killed
VBIN(l_1, e, $flag$)	at the entry of l_1, e is *busy*, where $flag$ is either Valid or Invalid indicating whether this information is held any longer
VBOUT(l_1, e, $flag$)	at the exit of l_1, e is *busy*, where $flag$ is either Valid or Invalid indicating whether this information is held any longer

After incorporating the flag, data flow equations of VB analysis could be translated to the following four clauses,

$$\forall l\ \forall e\ (\text{VBOUT}(l, e,\text{VALID}) \wedge \neg\text{VBKILL}(l, e)) \vee \text{VBGEN}(l, e) \Rightarrow \text{VBIN}(l, e,\text{VALID}) \quad (1)$$
$$\forall l\ \forall e\ \text{VBOUT}(l, e,\text{INVALID}) \wedge \neg\text{VBGEN}(l, e) \Rightarrow \text{VBIN}(l, e,\text{INVALID}) \quad (2)$$
$$\forall l\ \forall e\ \forall l1\ \forall f\ \text{VBIN}(l1, e, f) \wedge \text{RFLOW}(l1, l) \Rightarrow \text{VBOUT}(l, e, f)) \quad (3)$$
$$\forall l\ \forall e\ \forall l1\ \text{VBOUT}(l1, e,\text{VALID}) \wedge \text{RFLOW}(l, l1) \wedge \neg\text{VBIN}(l, e,\text{VALID}) \Rightarrow \text{VBOUT}(l1, e,\text{INVALID})) \quad (4)$$

The first two clauses take care about the relationship between the information at the exit and entry of a block, while clause (1) states that an expression is valid at the entry of a block when either it is valid at the exit and is not killed in the block or it is generated in the block; clause (2) states that if an expression is invalid at the exit of a block and it is not one of the expressions generated in the block then it is still invalid at the entry of the block. Relationship between two blocks is specified by the last two clauses. The idea is, for each node, we first let it has all the information held by it predecessors, which similar to a *may* analysis, and then select out the expressions not held any longer and set their flags to Invalid. Clause (3) means that if reverse flow($l1, l$) exists, all the expressions held at the entry of $l1$, no matter valid or invalid are going to be held at the exit of l without changed their validity. Clearly these expressions at the exit of l may be valid or invalid, among all the valid expressions, if one is not valid at one predecessors, then it will be marked as invalid. That is to say, an expression is valid at one node if and only if it is not invalid at any of the predecessors. This work is done by clause (4).

Example 3.8 Assume a program contains the following statements
$[A[1] := x + A[2]]^1$; $[x := A[x] + y]^2$; If $[x < 10]^3$ Then $[A[1] := x + y]^4$ Else $[x := x + y]^5$
and $A[1]$ and $A[2]$ are the only elements of array A
Clause generated according to data flow specifications is,

RFLOW(L2,L1)$\wedge$RFLOW(L3,L2)$\wedge$RFLOW(L4,L3)$\wedge$RFLOW(L5,L3)$\wedge$

VBGEN(L1,EXP(PLUS($x, A'2$)))$\wedge$VBKILL(L1,EXP(PLUS($A'x, y$)))$\wedge$
VBGEN(L2,EXP(PLUS($A'x, y$)))$\wedge$VBKILL(L2,EXP(PLUS($x, A'2$)))$\wedge$
VBKILL(L2,EXP(PLUS(x, y)))$\wedge$VBKILL(L5,EXP(PLUS($A'x, y$)))
VBGEN(L4,EXP(PLUS(x, y)))$\wedge$VBKILL(L4,EXP(PLUS($A'x, y$)))$\wedge$
VBGEN(L5,EXP(PLUS(x, y)))$\wedge$VBKILL(L5,EXP(PLUS($x, A'2$)))$\wedge$
VBKILL(L5,EXP(PLUS($A'x, y$)))$\wedge$VBKILL(L5,EXP(PLUS(x, y)))$\wedge$

($\forall l$ $\forall e$ (VBOUT(l, e,VALID)$\wedge\neg$VBKILL(l, e))$\vee$VBGEN(l, e)$\Rightarrow$VBIN(l, e,VALID))$\wedge$
($\forall l$ $\forall e$ VBOUT(l, e,INVALID)$\wedge\neg$ VBGEN(l, e)$\Rightarrow$VBIN(l, e,INVALID))$\wedge$
($\forall l$ $\forall e$ $\forall l1$ $\forall f$ VBIN($l1, e, f$) $\wedge$RFLOW($l1, l$)$\Rightarrow$VBOUT(l, e, f))$\wedge$
($\forall l$ $\forall e$ $\forall l1$ VBOUT($l1, e$,VALID)$\wedge$ RFLOW($l, l1$)$\wedge\neg$VBIN(l, e,VALID)$\Rightarrow$VBOUT($l1, e$,INVALID))

The summarized result is presented below.

VB$_\circ$	Valid expressions	VB$_\bullet$	Valid expressions
L1	$x + A[2]$	L1	$A[x] + y$
L2	$A[x] + y$	L2	$x + y$
L3	$x + y$	L3	$x + y$
L4	$x + y$	L4	
L5	$x + y$	L5	

3.3.2 Flow Logic Approach

Generating ALFP Clauses

The flow graph of backward analysis is different with that of forward analysis in the sense that there is an edge from block l to block l' if $\mathit{flow}^R(l, l')$ exists. One example is shown in Table 3.6.

Detail specifications for VB analysis are listed in Table 3.6. Note that we handle the

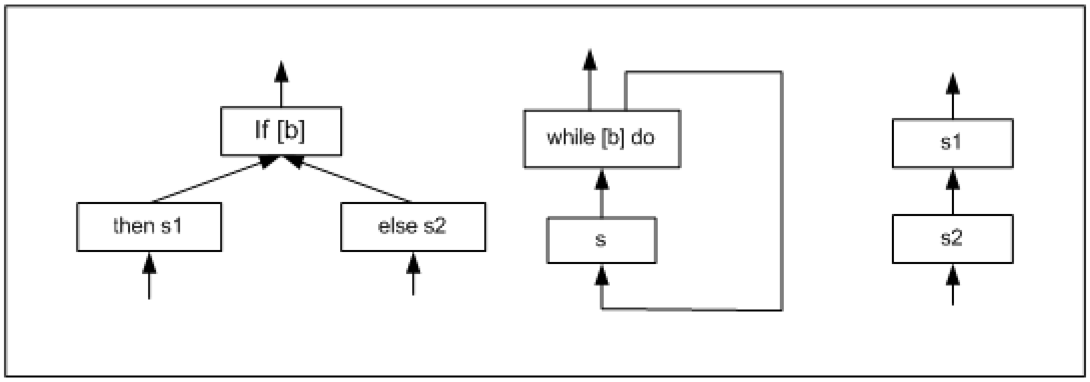


Figure 3.6: Flow Relation of Backward Analysis

inter- and intra-statement flowing information in the same fashion as for reaching definition analysis, meaning that we write

$$(\mathrm{VB}_\circ,\ \mathrm{VB}_\bullet) \models s \text{ and } (\mathrm{VB}_\circ,\ \mathrm{VB}_\bullet) \models B$$

for when $(\mathrm{VB}_\circ,\ \mathrm{VB}_\bullet)$ is an acceptable analysis of statement s or elementary block B, where

$$\mathrm{VB}_\circ,\ \mathrm{VB}_\bullet \in \widehat{Cache} = \mathrm{Lab}_* \to \mathcal{P}(Exp)$$

In Table 3.6, it states that if the elementary block is an assignment or a test, all the sub-expressions of the right-hand-side of that assignment or the test become valid at the entry of that block. Furthermore, in case of $[x := e]^l$ valid expressions at exit of l are still valid at entry of l provided that x is not a free variable of it. In case of $[A[n] := e]^l$ and $[A[e] := e']$, expression at exit of l can go through the block to the entry if it does not have A as a free array name of it. The lower part of the table mainly states that for a reverse flow (l_1, l_2), it is always the case that $\mathrm{VB}_\bullet(l_2) \Subset \mathrm{VB}_\circ(l_1)$, where symbol $\Subset$ represents the operator $\sqcap$ and indicates the relationship between a flow pair. $\mathrm{VB}_\circ(l) \Subset \mathrm{VB}_\bullet(l')$ for $\mathrm{flow}^R(l, l')$ is defined as,

$\forall$ e (e,Valid) $\in \mathrm{VB}_\circ(l) \Rightarrow$ (e,Valid) $\in \mathrm{VB}_\bullet(l')$
$\forall$ e (e,Invalid) $\in \mathrm{VB}_\circ(l) \Rightarrow$ (e,Invalid) $\in \mathrm{VB}_\bullet(l')$
$\forall$ e (e,Valid) $\in \mathrm{VB}_\bullet(l') \wedge$ (e,Valid) $\notin \mathrm{VB}_\circ(l) \Rightarrow$ (e,Invalid) $\in \mathrm{VB}_\bullet(l')$

The first two clauses let the information, no matter it is valid or invalid, at entry of l flows into the exit of l'. The third clause says that, information at the exit of l' is valid only when it holds at the entry of l. When there are more than one predecessors of l', information should be held at entry points of all the predecessors in order for it to be valid at exit of l'.

$(VB_\bullet, VB_\circ) \models [x{:=}\ e]^l$	iff $\forall\ e' \in Exp(e)\ (e',valid) \in VB_\circ(l) \wedge$ $\forall\ e'\ (e',Valid) \in VB_\bullet(l) \wedge x \notin Fv(e')$ $\Rightarrow (e',valid) \in VB_\circ(l)$
$(VB_\bullet, VB_\circ) \models [A[n]{:=}\ e]^l$	iff $\forall\ e' \in Exp(e)\ (e',valid) \in VB_\circ(l) \wedge$ $\forall e_1\ (e_1,\ Valid) \in VB_\bullet(l) \wedge A \notin FAv(e_1)$ $\Rightarrow (e_1,valid) \in VB_\circ(l)$
$(VB_\bullet, VB_\circ) \models [A[e]{:=}\ e']^l$	iff $\forall\ e' \in Exp(e)\ (e',valid) \in VB_\circ(l) \wedge$ $\forall e_1\ (e_1,\ Valid) \in VB_\bullet(l) \wedge A \notin FAv(e_1)$ $\Rightarrow (e_1,valid) \in VB_\circ(l)$
$(VB_\bullet, VB_\circ) \models [skip]^l$	iff $VB_\bullet(l) \subseteq VB_\circ(l)$
$(VB_\bullet, VB_\circ) \models [e]^l$	iff $\forall e'\ e' \in Exp(e) \Rightarrow (e',valid) \in VB_\circ(l) \wedge$ $VB_\bullet(l) \subseteq VB_\circ(l)$
$(VB_\bullet, VB_\circ) \models s_1;\ s_2$	iff $(VB_\bullet, VB_\circ) \models s_1 \wedge$ $(VB_\bullet, VB_\circ) \models s_2 \wedge$ $\forall l \in final(s_1)\ VB_\circ(init(s_2)) \Subset VB_\bullet(l)$
$(VB_\bullet, VB_\circ) \models$ if $[e]^l$ then s_1 else s_2	iff $(VB_\bullet, VB_\circ) \models [e]^l \wedge$ $(VB_\bullet, VB_\circ) \models s_1 \wedge$ $(VB_\bullet, VB_\circ) \models s_2 \wedge$ $VB_\circ(init(s_1)) \Subset VB_\bullet(1) \wedge$ $VB_\circ(init(s_2)) \Subset VB_\bullet(1)$
$(VB_\bullet, VB_\circ) \models$ while $[e]^l$ do s	iff $(VB_\bullet, VB_\circ) \models [e]^l \wedge$ $(VB_\bullet, VB_\circ) \models s \wedge$ $VB_\circ(init(s)) \Subset VB_\bullet(l) \wedge$ $\forall l' \in final(s)\ VB_\circ(l) \Subset VB_\bullet(l')$

Table 3.6: Control Flow Analysis of VB for Elementary Blocks and Statements

Implementation

Besides the predicates explained in the data flow approach part, the following predicates are used additionally.

- FAV(A, e), which indicates that element of array A has appeared in the expression e. This equals to say A ∈FAv(e)
- FV(x, e), meaning that x appears in e, which equals to say x ∈Fv(e)

VB analyzer goes through the programs and collects all the expressions contained in each statement. These expressions are then examined to check which array names and simple variables have appeared in it, thus interpretations for predicates FAV and FV are established.

$\text{VB}_\circ(l) \Supset \text{VB}_\bullet(l')$ for $\text{flow}^R(l, l')$ is translated to the clauses,

$$\forall e\ \forall f\ \text{VBIN}(l, e, f) \Rightarrow \text{VBOUT}(l', e, f)$$
$$\forall e\ \text{VBOUT}(l', e, \text{Valid}) \wedge \neg\text{VBIN}(l, e, \text{Valid}) \Rightarrow \text{VBOUT}(l', e, \text{Invalid})$$

These are the same as the clauses specify the inter-statement relationship in data flow approach and we refer to that part for explanation.

Order of Clauses

Recall that mentioned in Chapter 2.1, *alternation-free* formulae are employed by Succinct Solver. A clause *cl* is an *alternation-free-Least Fixpoint formula* if it could satisfy the three properties of the function *Rank*. In order to deal with negated queries, we have to decide the order of clauses carefully. Clauses used for VB analysis mainly have the followed three forms. We skip both the arguments and other irrelevant predicates and only list concerned predicates names here.

$$\text{VBIN} \Rightarrow \text{VBOUT}$$
$$\text{VBOUT} \Rightarrow \text{VBIN}$$
$$\neg\text{VBIN} \Rightarrow \text{VBOUT}$$

Before feeding clauses into succinct solver, we have to re-order them such that clauses of the third form are always placed at the end. We claim that the three properties, mentioned above, hold by doing that. Assume there are totally n clauses and m of them have the negated queries(¬VBIN), which are the last ones. All the other clauses then occupy the first $n - m$ places. The whole picture looks like,

$$n \left\{ \begin{array}{c} \left. \begin{array}{cccc} \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{array} \right\} n-m \\ \left. \begin{array}{c} \neg\text{VBIN}\Rightarrow\text{VBOUT} \\ \neg\text{VBIN}\Rightarrow\text{VBOUT} \\ \neg\text{VBIN}\Rightarrow\text{VBOUT} \end{array} \right\} m \end{array} \right.$$

From the upper part of the picture, it can be seen that $Rank(\text{VBOUT}) \leq n-m$ and $Rank(\text{VBIN}) \leq n-m$, since they act as queries and the lower part tells that $Rank(\text{VBIN}) < n-m+1$, the same as $Rank(\text{VBIN}) \leq n-m$, because its in negated queries. Therefore, the properties hold in the sense that there is a non-empty solution space for the above three inequations.

Example 3.9 By analyzing the program in example 3.8, the following clause is generated according to flow logic specifications.

FV(x,EXP(PLUS($x, A'2$)))∧FV(y,EXP(PLUS($A'x, y$)))∧
FV(x,EXP(PLUS(x, y)))∧FV(y,EXP(PLUS(x, y)))∧
FAV(A,EXP(PLUS($x, A'2$)))∧FAV(A,EXP(PLUS($A'x, y$)))∧

VBIN(L5,EXP(PLUS(x, y)),VALID)∧
(∀e VBOUT(L5,e,VALID)∧¬FV(x,e)⇒VBIN(L5,e,VALID))∧

VBIN(L4,EXP(PLUS(x, y)),VALID)∧
(∀e VBOUT(L4,e,VALID)∧¬FAV(A,e)⇒VBIN(L4,e,VALID))∧

(∀e VBOUT(L3,e,VALID)⇒VBIN(L3,e,VALID))∧

VBIN(L2,EXP(PLUS($A'x, y$)),VALID)∧
(∀e VBOUT(L2,e,VALID)∧¬FV(x,e)⇒VBIN(L2,e,VALID))∧

VBIN(L1,EXP(PLUS($x, A'2$)),VALID)∧
(∀e VBOUT(L1,e,VALID)∧¬FAV(A,e)⇒VBIN(L1,e,VALID))∧

(∀e ∀f VBIN(L5,e, f)⇒VBOUT(L3, e, f))∧
(∀e VBOUT(L3,e,VALID)∧¬VBIN(L5,e,VALID) ⇒ VBOUT(L3,e,INVALID))∧

(∀e ∀f VBIN(L4,e, f)⇒VBOUT(L3, e, f))∧
(∀e VBOUT(L3,e,VALID)∧¬VBIN(L4,e,VALID) ⇒ VBOUT(L3,e,INVALID))∧

$(\forall e\ \forall f\ \text{VBIN}(\text{L3},e,f) \Rightarrow \text{VBOUT}(\text{L2},e,f)) \wedge$
$(\forall e\ \forall f\ \text{VBIN}(\text{L2},e,f) \Rightarrow \text{VBOUT}(\text{L1},e,f))$
The solving result is the same as one in example 3.8.

Comparison of Two Approaches

Figure 3.12 shows the execution time of Succinct Solver running on two sets of specifications. As it can be seen from Figure 3.12, the complexities of solve on the two sets

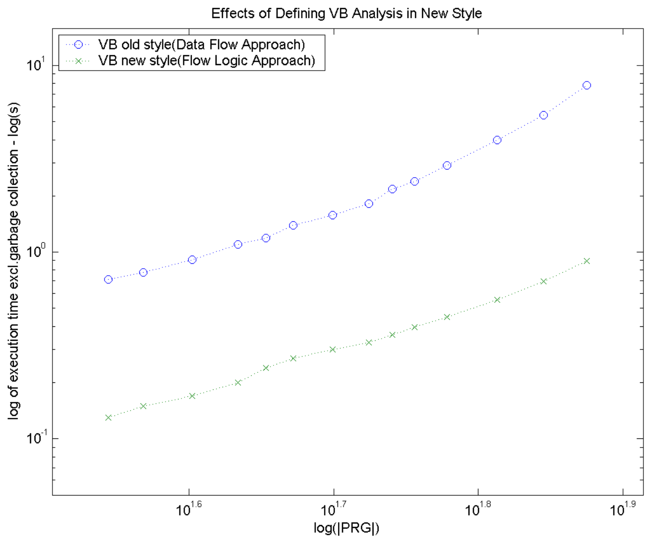


Figure 3.7: Comparison of execution times on a-*m*

are equal but for flow logic specification, it has a constant factor in its favor. The small indication on the complexity polynomial is too small to be significant.

3.4 Available Expression Analysis

Available expression analysis (AE for short) will determine:

> For each program point, which expressions *must* have already been computed, and not later modified, on all paths to the program point.

This information can be used to avoid the re-computation of an expression. For clarity, we will concentrate on arithmetic expressions.

Example 3.10 In the following statement

$$[x := a + b]^1;\ \text{if } [x > 1]^2 \text{ Then } [y := a + b]^3 \text{ Else } [y := x]^4$$

expression $a + b$ is available at block 3 since its value has been computed in block 1.

3.4.1 Data Flow Approach

An expression is killed in a block if any of the variables used in the expression is modified in the block; we use the function

$$kill_{AE} : \text{Block}_* \to \mathcal{P}(\text{Exp})$$

to produce the set of non-trivial expressions killed in the block. An expression is generated

1	$[x := e]^l$	$\{e' \in \text{Exp}_* \mid x \in \text{Fv}(e')$
2	$[A[n] := e]^l$	$\{e' \in \text{Exp}_* \mid A \in \text{FAv}(e')$
3	$[A[e] := e']^l$	$\{e' \in \text{Exp}_* \mid A \in \text{FAv}(e')$
4	$[Skip]^l$	$\emptyset$
5	$[e]^l$	$\emptyset$

Table 3.7: Kill Component of AE Analysis

when it is evaluated in the block and none of the variables used in the expression is modified in the block. The set of generated non-trivial expressions is produced by the function:

$$gen_{AE} : \text{Block}_* \to \mathcal{P}(\text{Exp})$$

Available expression analysis is a *forward must* analysis calculating the largest sets sat-

1	$[x := e]^l$	$\{e' \in \text{subExp}(e) \mid x \notin \text{Fv}(e')$
2	$[A[n] := e]^l$	$\{e' \in \text{subExp}(e) \mid A \notin \text{FAv}(e')$
3	$[A[e] := e_1]^l$	$\{e' \in \text{subExp}(e_1) \mid A \notin \text{FAv}(e')$
4	$[Skip]^l$	$\emptyset$
5	$[e]^l$	$\text{subExp}(b)$

Table 3.8: Gen Component of AE Analysis

isfying the equation for $\text{AE}_\circ$ showed in Figure 3.8.

Predicates used by AE analysis generator are,

$$
\begin{array}{rcll}
\text{AE}_\circ(l) & = & \begin{cases} \emptyset & \text{if } l = \text{init}(\text{S}_*) \\ \bigcap\{\text{AE}_\bullet(l') \mid (l',l) \in \text{flow}(\text{S}_*)\} & \text{otherwise} \end{cases} \\
\text{AE}_\bullet(l) & = & (\text{AE}_\circ(l) \setminus kill_{AE}(B^l)) \cup gen_{AE}(B^l) & \text{where } B^l \in \text{Block}_*
\end{array}
$$

Figure 3.8: Data Flow Equations of AE Analysis

Predicate	**Meaning**
EXP(e)	e is an expression
PLUS(x, y)	expression $x + y$
FLOW(l_1,l_2)	there is a flow from l_1 to l_2
VAR(x)	x is a variable in program
AEGEN(l_1, e)	at block l_1, e is generated
AEKILL(l_1, e)	at block l_1, e is killed
AEIN(l_1, e, $flag$)	at the entry of l_1, e is *available*, where $flag$ is either Valid or Invalid indicating whether this information is held any longer
AEOUT(l_1, e, $flag$)	at the exit of l_1, e is *available*, where $flag$ is either Valid or Invalid indicating whether this information is held any longer

Clauses for statements could be generated directly according to the specifications. AE analysis is a *must* analysis, thus a *flag* is employ to indicate whether an information is still valid or not at each block. Similar to VB analysis, clauses specifying the data flow equations are generated as,

$$\forall l\ \forall e\ (\text{AEIN}(l, e,\text{VALID}) \wedge \neg\text{AEKILL}(l, e)) \vee \text{AEGEN}(l, e) \Rightarrow \text{AEOUT}(l, e,\text{VALID}) \quad (1)$$
$$\forall l\ \forall e\ \text{AEIN}(l, e,\text{INVALID}) \wedge \neg\text{AEGEN}(l, e) \Rightarrow \text{AEOUT}(l, e,\text{INVALID}) \quad (2)$$
$$\forall l\ \forall e\ \forall l1\ \forall f\ \text{AEOUT}(l1, e, f) \wedge \text{FLOW}(l1, l) \Rightarrow \text{AEIN}(l, e, f)) \quad (3)$$
$$\forall l\ \forall e\ \forall l1\ \text{AEIN}(l1, e,\text{VALID}) \wedge \text{FLOW}(l, l1) \wedge \neg\text{AEOUT}(l, e,\text{VALID}) \Rightarrow \text{AEIN}(l1, e,\text{INVALID})) \quad (4)$$

Example 3.11 Consider the program in example 3.8. The clause generated for the program is,

FLOW(L1,L2)∧FLOW(L2,L3)∧FLOW(L3,L4)∧FLOW(L3,L5)∧

AEGEN(L1,EXP(PLUS($x, A'2$)))∧AEKILL(L1,EXP(PLUS($A'x, y$)))∧
AEKILL(L2,EXP(PLUS($x, A'2$)))∧AEKILL(L2,EXP(PLUS(x, y)))∧
AEKILL(L2,EXP(PLUS($A'x, y$)))∧
AEGEN(L4,EXP(PLUS(x, y)))∧AEKILL(L4,EXP(PLUS($A'x, y$)))∧

AEKILL(L5,EXP(PLUS($x, A'2$)))∧AEKILL(L5,EXP(PLUS($A'x, y$)))∧
AEKILL(L5,EXP(PLUS(x, y)))∧

(∀l ∀e (AEIN(l, e,VALID)∧¬AEKILL(l, e))∨AEGEN(l, e)⇒ AEOUT(l, e,VALID)))∧
(∀l ∀e AEIN(l, e,INVALID)∧¬ AEGEN(l, e)⇒AEOUT(l, e,INVALID))∧
(∀l ∀e ∀$l1$ ∀f AEOUT($l1, e, f$)∧FLOW($l1, l$)⇒AEIN(l, e, f))∧
(∀l ∀e ∀$l1$ AEIN($l1, e$,VALID)∧ FLOW($l, l1$)∧¬AEOUT(l, e,VALID)⇒AEIN($l1, e$,INVALID))

The summarized result is listed in the blow table.

$\text{AE}_\circ$	available expressions	$\text{AE}_\bullet$	available expressions
L1		L1	$x + A[2]$
L2	$x + A[2]$	L2	
L3		L3	
L4		L4	$x + y$
L5		L5	

3.4.2 Flow Logic Approach

The result of available expression analysis is a pair ($\text{AE}_\circ$, $\text{AE}_\bullet$), where

$$\text{AE}_\circ,\ \text{AE}_\bullet \in \widehat{Cache} = \text{Lab}_* \to \mathcal{P}(\text{Exp})$$

where elementary blocks and statements have to satisfy the result in order to be acceptable.

$$(\text{AE}_\circ,\text{LV}_\bullet) \models B \textit{ and } (\text{AE}_\circ,\text{LV}_\bullet) \models s$$

Rules for satisfiability of elementary blocks are listed in the upper part of Table 3.4.2. It states that, in case of $[x := e]^l$, all the sub-expressions of e, which do not have x as free variable are valid at the exit of l. All the sub-expressions valid at the entry of l are still valid at exit if x is not a free variable of it. In case of $[A[m] := e]^l$, all the sub-expressions of e of e, which do not contain element of array A, are valid at exit of l. Sub-expressions valid at entry of l are still valid at exit provided that not element of array A are contained in it. This condition is also applicable to $[A[e] := e']^l$. In case of test, sub-expressions of the test become valid at the exit and all expressions valid at entry are valid at exit.

The lower part of the table shows the satisfiability rules of statements. Similar to VB analysis, since available expression analysis is a *must* analysis, operator $\Subset$ is used to specify the relationship between a flow pair(i.e. $\text{AE}_\bullet(l_1) \Subset \text{AE}_\circ(l_2)$ when flow(l_1, l_2) exists), the corresponding clause is,

∀e ∀f AEOUT(l_1, e, f) ⇒ AEIN(l_2, e, f)
∀e AEIN(l_2, e,Valid) ∧ ¬AEOUT(l_1, e,Valid) ⇒ AEIN(l_2, e,Invalid)

$(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models [x := e]^l$	iff $\forall e'(e' \in \mathrm{subExp}(e) \wedge x \notin \mathrm{Fv}(e') \Rightarrow (e',\mathrm{Valid}) \in \mathrm{AE}_\bullet(l)) \wedge$ $\forall e'\ ((e',\mathrm{Valid}) \in \mathrm{AE}_\circ(l) \wedge x \notin \mathrm{Fv}(e') \Rightarrow (e',\mathrm{Valid}) \in \mathrm{AE}_\bullet(l))$
$(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models [A[n] := e]^l$	iff $\forall e' \in \mathrm{subExp}(e) \wedge A \notin \mathrm{FAv}(e') \Rightarrow (e',\mathrm{Valid}) \in \mathrm{AE}_\bullet(l) \wedge$ $\forall e'\ ((e',\mathrm{Valid}) \in \mathrm{AE}_\circ(l) \wedge A \notin \mathrm{FAv}(e') \Rightarrow (e',\mathrm{Valid}) \in \mathrm{AE}_\bullet(l))$
$(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models [A[e] := e']^l$	iff $\forall e' \in \mathrm{subExp}(e) \wedge A \notin \mathrm{FAv}(e') \Rightarrow (e',\mathrm{Valid}) \in \mathrm{AE}_\bullet(l) \wedge$ $\forall e'\ ((e',\mathrm{Valid}) \in \mathrm{AE}_\circ(l) \wedge A \notin \mathrm{FAv}(e') \Rightarrow (e',\mathrm{Valid}) \in \mathrm{AE}_\bullet(l))$
$(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models [Skip]^l$	iff $\mathrm{AE}_\circ(l) = \mathrm{AE}_\bullet(l)$
$(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models [e]^l$	iff $\forall e' \in \mathrm{subExp}(e) \Rightarrow (e',\mathrm{Valid}) \in \mathrm{AE}_\bullet(l) \wedge$ $\mathrm{AE}_\circ(l) \subseteq \mathrm{AE}_\bullet(l)$
$(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models s_1; s_2$	iff $(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models s_1 \wedge$ $(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models s_2 \wedge$ $\forall l\ (l \in \mathrm{final}(s_1) \Rightarrow \mathrm{AE}_\circ(\mathrm{init}(s_2)) \Subset \mathrm{AE}_\bullet(l))$
$(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models$ If $[e]^l$ Then s_1 Else s_2	iff $(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models [e]^l \wedge$ $(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models s_1 \wedge$ $(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models s_2 \wedge$ $\mathrm{AE}_\circ(\mathrm{init}(s_1)) \Subset \mathrm{AE}_\bullet(l) \wedge$ $\mathrm{AE}_\circ(\mathrm{init}(s_2)) \Subset \mathrm{AE}_\bullet(l)$
$(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models$ While $[e]^l$ Do s	iff $(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models [e]^l \wedge$ $(\mathrm{AE}_\circ,\mathrm{AE}_\bullet) \models s \wedge$ $\mathrm{AE}_\circ(\mathrm{init}(s)) \Subset \mathrm{AE}_\bullet(l) \wedge$ $\forall l'(l' \in \mathrm{final}(s) \Rightarrow \mathrm{AE}_\circ(l) \Subset \mathrm{AE}_\bullet(l'))$

Table 3.9: Control Flow Analysis of AE for Elementary Blocks and Statements

Example 3.12 Returning to the program in example 3.8. Flow logic analysis generator will create the following clause

FV(x,EXP(PLUS($x, A'2$)))∧FV(x,EXP(PLUS($A'x, y$)))∧
FV(y,EXP(PLUS($A'x, y$)))∧FV(x,EXP(PLUS(x, y)))∧
FV(y,EXP(PLUS(x, y)))∧
FAV(A,EXP(PLUS($x, A'2$)))∧FAV(A,EXP(PLUS($A'x, y$)))∧

AEOUT(L1,EXP(PLUS($x, A'2$)),VALID)∧
(∀e AEIN(L1,e,VALID)∧!FAV(A, e)⇒AEOUT(L1,e,VALID))∧
(∀e AEIN(L2,e,VALID)∧!FV(x, e)⇒AEOUT(L2,e,VALID))∧
(∀e AEIN(L3,e,VALID)⇒AEOUT(L3,e,VALID))∧
AEOUT(L4,EXP(PLUS(x, y)),VALID)∧
(∀e AEIN(L4,e,VALID)∧!FAV(A, e)⇒AEOUT(L4,e,VALID))∧
(∀e AEIN(L5,e,VALID)∧!FV(x, e)⇒AEOUT(L5,e,VALID))∧

(∀e AEOUT(L1,e)⇒AEIN(L2,e))∧
(∀e AEOUT(L2,e)⇒AEIN(L3,e))∧
(∀e AEOUT(L3,e)⇒AEIN(L4,e))∧
(∀e AEOUT(L3,e)⇒AEIN(L5,e))

The result given by Succinct Solve is exactly the same as the one in data flow approach example.

Figure 3.9 does not show a great degree difference of the complexity polynomial between data flow specification and flow logic specification, rather the two lines are almost in parallel and the degree difference is quite small.

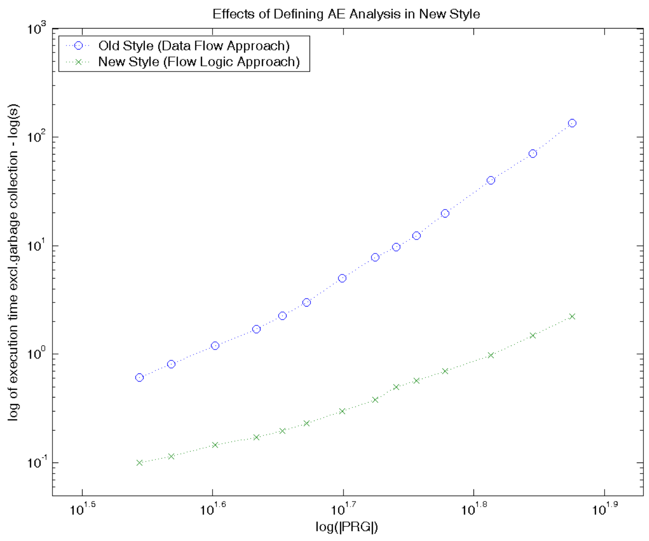


Figure 3.9: Effect of Defining AE Analysis in Flow Logic Approach

3.5 Live Variable Analysis

A variable is *live* at the exit of an elementary block if there exists a path from the block to a use of the variable that does not redefine the variable. The live variable analysis (LV for short) will determine:

For each program point, which variable *may* be live at the exit from the point.

This analysis might be used as the basis for dead code elimination. If the variable is not live at the exit form a block then, if the block is an assignment to the variable, the block can be eliminated.

Example 3.13 In the program below

$$[x := 1]^1;\ [x := 2]^2\ [y := y + x]^3$$

variable x is not live at the exit of block 1, since its value is change in block 2 before used in block 3.

3.5.1 Data Flow Approach

Live variable analysis is a *backward may* analysis, we are interested in the smallest set satisfying the equation for $\mathrm{LV}_\bullet$, which is specified in Figure 3.10.
By analogy with the previous analyses, we also need to define when an expression is killed and when a block generates additional very busy expressions. For this, we use functions:

$$\begin{aligned} kill_{vb}: &\quad \mathrm{Block}_* \rightarrow \mathcal{P}(Var) \\ gen_{vb}: &\quad \mathrm{Block}_* \rightarrow \mathcal{P}(Var) \end{aligned}$$

These functions are defined as in Table 3.10 and Table 3.11. Variable appearing as an

1	$[x := e]^l$	$\{x\}$
2	$[A[n] := e]^l$	$\{A[n]\}$
3	$[A[e] := e']^l$	$\emptyset$
4	$[Skip]^l$	$\emptyset$
5	$[e]^l$	$\emptyset$

Table 3.10: Kill Component of LV Analysis

assignee of an assignment statement is going to be killed, except it is an array element without a constant index. *Skip* statement and conditional do not kill variables. All the free variables appear on the right-hand-side of an assignment statement or a conditional are generated. Predicates used in the generated clause are

1	$[x := \ e]^l$	Fv(e)
2	$[A[n] := \ e]^l$	Fv(e)
3	$[A[e] := \ e']^l$	Fv(e')
4	$[Skip]^l$	$\emptyset$
5	$[e]^l$	Fv(e)

Table 3.11: Gen Component of LV Analysis

$$\begin{aligned} \mathrm{LV}_\bullet(l) &= \begin{cases} \emptyset & \text{if } l \in \mathrm{final(S_*)} \\ \bigcup\{\mathrm{LV}_\circ(l') \mid (l',l) \in \mathrm{flow}^R(\mathrm{S_*})\} & \text{otherwise} \end{cases} \\ \mathrm{LV}_\bullet(l) &= (\mathrm{LV}_\bullet(l) \setminus kill_{LV}(B^l)) \cup gen_{LV}(B^l) \quad \text{where } B^l \in \mathrm{Block}_* \end{aligned}$$

Figure 3.10: Data Flow Equations of LV Analysis

Predicate	**Meaning**
FINAL(l)	l is the label of a final block
RFLOW(l_1,l_2)	there is a reverse flow from l_1 to l_2
LVGEN(l_1, x)	at block l_1, variable x is generated
LVKILL(l_1, x)	at block l_1, variable x is killed
LVIN(l_1, x)	at the entry of l_1, x is *live*,
LVOUT(l_1, x)	at the exit of l_1, x is *live*,

As mentioned in chapter 2.1, variables declared followed the keyword OUTPUT are the return variables, meaning that their values will be held when the execution of programs finishes. The idea is to have these variables live at the exit of final block. This is specified by the clause

$$\forall l \ \mathrm{FINAL}(l) \Rightarrow (\mathrm{LVOUT}(l, x))$$

where x is a return variable.
Data flow equations of LV analysis, similar to RD analysis, are specified by the clause

$(\forall l \ \mathrm{FINAL}(l) \Rightarrow \mathrm{LVOUT}(l, y)) \wedge$
$(\forall l1 \ \forall l2 \ \forall x \ \mathrm{LVIN}(l1, x) \wedge \mathrm{RFLOW}(l1, l2) \Rightarrow \mathrm{LVOUT}(l2, x)) \wedge$
$(\forall l \ \forall x \ (\mathrm{LVOUT}(l, x) \wedge \neg \mathrm{LVKILL}(l, x)) \vee \mathrm{LVGEN}(l, x) \Rightarrow \mathrm{LVIN}(l, x))$

Example 3.14 Data flow specifications give rise to the following clause for the program piece in example 3.8 (Suppose y is declared as a return variable)

FINAL(L4)∧FINAL(L5)∧
RFLOW(L2,L1)∧RFLOW(L3,L2)∧RFLOW(L4,L3)∧RFLOW(L5,L3)∧

LVGEN(L5,x)$\wedge$LVGEN(L5,y)$\wedge$LVKILL(L5,x)$\wedge$
LVGEN(L4,x)$\wedge$LVGEN(L4,y)$\wedge$LVKILL(L4,$A'1$)$\wedge$
LVGEN(L2,$A'1$)$\wedge$LVGEN(L2,$A'2$)$\wedge$LVGEN(L2,y)$\wedge$LVKILL(L2,x)$\wedge$
LVGEN(L1,x)$\wedge$LVGEN(L1,$A'2$)$\wedge$LVKILL(L1,$A'1$)$\wedge$

($\forall l$ FINAL(l)$\Rightarrow$LVOUT(l, y))$\wedge$
($\forall l1$ $\forall l2$ $\forall x$ LVIN($l1, x$)$\wedge$RFLOW($l1, l2$)$\Rightarrow$LVOUT($l2, x$))$\wedge$
($\forall l$ $\forall x$ (LVOUT(l, x)$\wedge\neg$LVKILL(l, x))$\vee$LVGEN(l, x)$\Rightarrow$LVIN(l, x))

Below table shows the summarized result

LV$_\circ$	live variables	LV$_\bullet$	live variables
L1	$x, A[2], y$	L1	$A[1], A[2], y$
L2	$A[1], A[2], y$	L2	x, y
L3	x, y	L3	x, y
L4	x, y	L4	y
L5	x, y	L5	y

3.5.2 Flow Logic Approach

The result of live variable analysis is a pair (LV$_\bullet$, LV$_\circ$), where

$\text{LV}_\bullet, \text{LV}_\circ \in \widehat{Cache} = \text{Lab}_* \to \mathcal{P}(Var)$

where elementary blocks and statements have to satisfy the result in order to be acceptable.

$$(\text{LV}_\bullet,\text{LV}_\circ) \models B \textit{ and } (\text{LV}_\bullet,\text{LV}_\circ) \models s$$

Live variable analysis is a *forward may* analysis, the relationship between LV$_\circ(l_1)$ and LV$_\bullet(l_2)$ (for flow^R) is

$\text{LV}_\circ(l_1) \subseteq \text{LV}_\bullet(l_2)$

Whether an elementary block is satisfiable is specified in Table 3.5.2. In case of $[x := e]^l$ or $[A[n] := e]^l$, free variables contained in e belong to LV$_\circ(l)$ and furthermore, those variables contained in the information at entry of l and not equal to the assignee of the statement (i.e. x or $A[n]$) are contained in the exit information.

$\subseteq$ is used because information at LV$_\bullet(l_2)$ comes from all the blocks where there exists a reverse flow from those blocks to l_2. Thus, information of each block is a subset of the whole.

$(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models [x := e]^l$	iff $\forall x \in \mathrm{Fv}(e) \Rightarrow x \in \mathrm{LV}_\circ(l) \wedge$ $\forall y\ (y \in \mathrm{LV}_\bullet(l) \wedge y \neq x \Rightarrow y \in \mathrm{LV}_\circ(l)) \wedge$
$(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models [A[n] := e]^l$	iff $\forall x \in \mathrm{Fv}(e) \Rightarrow x \in \mathrm{LV}_\circ(l) \wedge$ $\forall y\ (y \in \mathrm{LV}_\bullet(l) \wedge y \neq A[n] \Rightarrow y \in \mathrm{LV}_\circ(l)) \wedge$
$(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models [A[e] := e']^l$	iff $\forall x \in \mathrm{Fv}(e') \Rightarrow x \in \mathrm{LV}_\circ(l) \wedge$ $\mathrm{LV}_\bullet(l) \subseteq \mathrm{LV}_\circ(l)$
$(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models [Skip]^l$	iff $\mathrm{LV}_\bullet(l) = \mathrm{LV}_\circ(l)$
$(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models [e]^l$	iff $\forall x \in \mathrm{Fv}(e) \Rightarrow x \in \mathrm{LV}_\circ(l) \wedge$ $\mathrm{LV}_\bullet(l) \subseteq \mathrm{LV}_\circ(l)$
$(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models s_1; s_2$	iff $(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models s_1 \wedge$ $(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models s_2 \wedge$ $\forall l \in \mathrm{final}(s_1) \Rightarrow \mathrm{LV}_\circ(\mathrm{init}(s_1) \subseteq \mathrm{LV}_\bullet(l))$
$(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models$ If $[e]^l$ Then s_1 Else s_2	iff $(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models [e]^l \wedge$ $(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models s_1 \wedge$ $(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models s_2 \wedge$ $\mathrm{LV}_\circ(\mathrm{init}(s_1)) \subseteq \mathrm{LV}_\bullet(l) \wedge$ $\mathrm{LV}_\circ(\mathrm{init}(s_2)) \subseteq \mathrm{LV}_\bullet(l)$
$(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models$ While $[e]^l$ Do s	iff $(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models [e]^l \wedge$ $(\mathrm{LV}_\bullet,\mathrm{LV}_\circ) \models s \wedge$ $\mathrm{LV}_\circ(\mathrm{init}(s)) \subseteq \mathrm{LV}_\bullet(l) \wedge$ $\forall l' \in \mathrm{final}(s) \Rightarrow \mathrm{LV}_\circ(l) \subseteq \mathrm{LV}_\bullet(l')$

Table 3.12: Control Flow Analysis of LV for Elementary Blocks and Statements

Example 3.15 According the flow logic specifications, LV analyzer will generate the following clause for the program piece in example 3.8 (again, assuming y is a return variable)

FINAL(L4)$\wedge$FINAL(L5)$\wedge$
($\forall l$ FINAL(l)$\Rightarrow$LVOUT(l, y))$\wedge$
LVIN(L5,x)$\wedge$LVIN(L5,y)$\wedge$
($\forall y$ LVOUT(L5,y)$\wedge y \neq x \Rightarrow$LVIN(L5,$y$))$\wedge$

LVIN(L4,x)$\wedge$LVIN(L4,y)$\wedge$
($\forall y$ LVOUT(L4,y)$\wedge y \neq A'1 \Rightarrow$LVIN(L4,$y$))$\wedge$

($\forall y$ LVOUT(L3,y)$\Rightarrow$LVIN(L3,y))$\wedge$

LVIN(L2,$A'1$)$\wedge$LVIN(L2,$A'2$)$\wedge$LVIN(L2,y)$\wedge$
($\forall y$ LVOUT(L2,y)$\wedge y \neq x \Rightarrow$LVIN(L2,$y$))$\wedge$

LVIN(L1,x)$\wedge$LVIN(L1,$A'2$)$\wedge$
($\forall y$ LVOUT(L1,y)$\wedge y \neq A'1 \Rightarrow$LVIN(L1,$y$))$\wedge$

($\forall x$LVIN(L5,x)$\Rightarrow$LVOUT(L3,x))$\wedge$
($\forall x$LVIN(L4,x)$\Rightarrow$LVOUT(L3,x))$\wedge$
($\forall x$LVIN(L3,x)$\Rightarrow$LVOUT(L2,x))$\wedge$
($\forall x$LVIN(L2,x)$\Rightarrow$LVOUT(L1,x))

The result of solving the above clause is exactly the same as the one in the example for data flow approach.

Figure 3.11 shows the execution time of Succinct Solver running on both data flow clause and flow logic clause for LV analysis, which indicates that data flow clause is worse by degrees of the complexity polynomial than the other one.

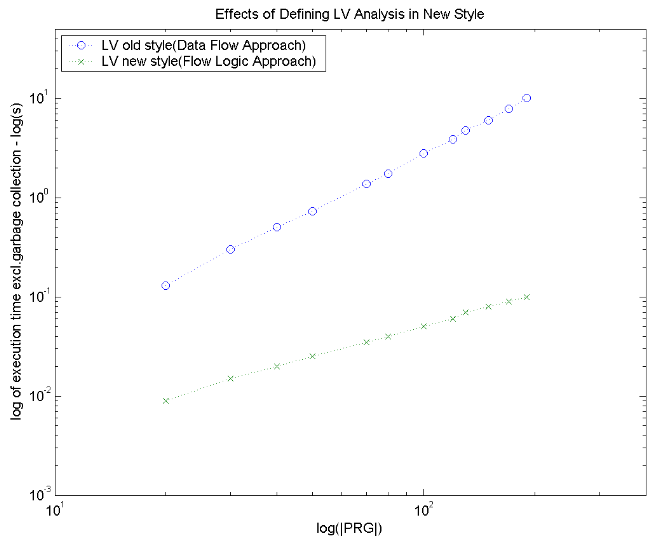


Figure 3.11: Effect of Defining LV analysis in Flow Logic Approach

3.6 Summary

Compared with data flow approach, applying flow logic approach to bit-vector analyses may increase the solving efficiency. For *may* analyses, flow logic specifications clearly exhibit a lower solving complexity but in the cases of *must* analyses, they are proved to be a small constant factor better but fails to show any significant indication on the complexity polynomials. This difference could be explained by considering the characters of analyses: *must* analyses satisfy descending chain condition at each program point, meaning that information about each program point becomes less gradually as solving progresses. In order to remove the un-useful information we invent a flag *Valid*. Consequently, information are categorized to *Valid* and *Invalid*, which make both the specification and implementation of *must* analyses more complicated than *may* analyses. It is somehow un-satisfiable as we try to show the advantages of flow logic specifications compared to data flow specifications. Some efforts have been put into achieving higher solving performance, as it is shown in the below section. During the progress of implementing the four classical analyses, some insights have been got, which enable us to conclude a general strategy for re-writing data flow specifications to flow logic specifications. We present it in the last section of this part.

3.6.1 Optimize Flow Logic Specification

The optimization mentioned in this section is especially for very busy expression analysis, but it may be applicable to other analyses if necessary.

- use the optimizing strategy given in chapter two
  - re-order the parameters of relations
    In VB specifications, we have the following clause querying whether a variable x appears in expression e as a free variable,
    $\forall\, e \ldots \wedge \mathrm{Fv}(e, x) \wedge \Rightarrow \ldots$
    where x is always bound to certain value. To facilitate the unification, we use the following clause instead
    $\forall\, e \ldots \wedge \mathrm{Fv}(x, e) \wedge \Rightarrow \ldots$
    The same optimization happens on querying $\mathrm{FAv}(e, A)$, which is changed to $\mathrm{FAv}(A, e)$
- apply different policies to different statements
  One advantage of flow logic approach compared to data flow approach is that constraints are not specified generally, therefore it enable us to deal with flow information differently according it's property. For example, in VB analysis, from an entry point's view, information flowing from different exit points can be divided into two parts,

- information comes from single exit point
  In this case, constraint can be defined as (for reverse flow(l_1, l_2)),

$$\mathrm{VB}_\bullet(l_2) \subseteq \mathrm{VB}_\circ(l_1)$$

- information comes from multiple exit points
  In this case, constraint can be defined as (for reverse flow(l_1, l_2)),

$$\mathrm{VB}_\bullet(l_2) \Subset \mathrm{VB}_\circ(l_1)$$

  where $\Subset$ is the same as in above section.

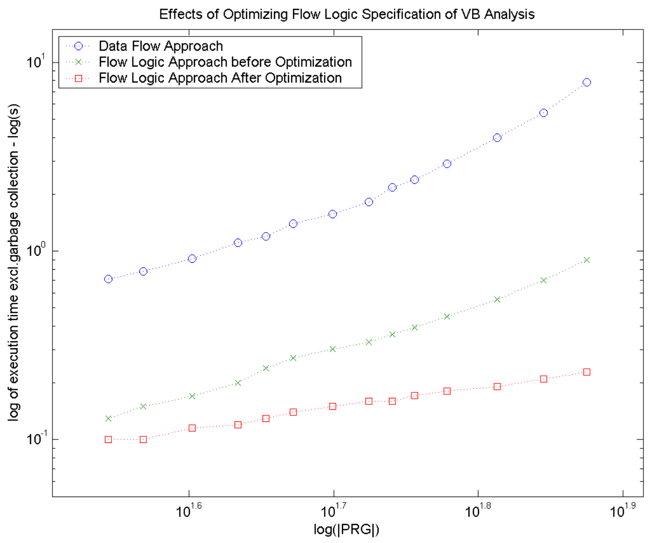
It can easily be seen that calculating $\Subset$ requires more time and spaces than that of $\subseteq$, therefore use $\Subset$ only when necessary will save work. Here we formalize the cases whether $\Subset$ or $\subseteq$ should be used, respectively.

- $\Subset$ should be used for $\text{flow}^R(l_1, l_2)$ where there exists label l_3 such that $\text{flow}^R(l_3, l_2)$ exists
- $\subseteq$ should be used for $\text{flow}^R(l_1, l_2)$ where there *doesn't* exists label l_3 such that $\text{flow}^R(l_3, l_2)$ exists

According to above criterions, two $\Subset$ operators used in flow logic specification for VB analysis described in chapter 3.3, can be conditionally optimized to $\subseteq$, which are listed in Table 3.13. For $s_1; s_2$, the only case when $\Subset$ must be used is the last statement of s_1 is a While $[e]^l$ Do s statement, if we regard s_1 as composed by a series of statements. In this case, information going into s_1 comes from both s and s_2. For While $[e]^l$ Do s statement, information going into the final block of s only originates from l, when the last statement of s is a While $[e]^l$ Do s statement, thus using $\subseteq$ saves work.

Statement	Before optimized	After optimized
$s_1; s_2$	$\forall l \in \text{final}(s_1) \Rightarrow$ $\text{VB}_\bullet(l) \Subset \text{VB}_\circ(\text{init}(s_2))$	$\forall l \in \text{final}(s_1) \Rightarrow \text{VB}_\bullet(l) \Subset \text{VB}_\circ(\text{init}(s_2))$ if the last statement of s_1 is a While...Do statement
		$\forall l \in \text{final}(s_1) \Rightarrow \text{VB}_\bullet(l) \subseteq \text{VB}_\circ(\text{init}(s_2))$ otherwise
While $[e]^l$ Do s	$\forall l' \in \text{final}(s) \Rightarrow$ $\text{VB}_\bullet(l') \Subset \text{VB}_\circ(l)$	$\forall l' \in \text{final}(s) \Rightarrow \text{VB}_\bullet(l') \Subset \text{VB}_\circ(l)$ if the last statement of s is a While...Do statement
		$\forall l' \in \text{final}(s) \Rightarrow \text{VB}_\bullet(l') \subseteq \text{VB}_\circ(l)$ otherwise

Table 3.13: Optimize Flow Logic Specification for VB analysis



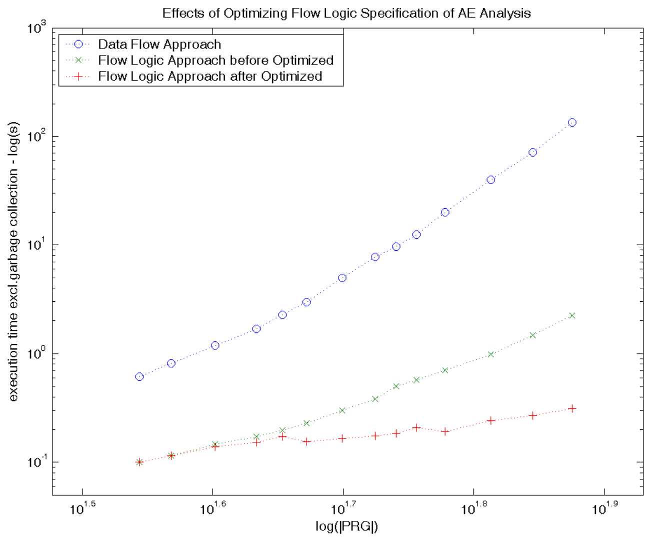


Figure 3.12: Effect of Optimizing Flow Logic Specification

These optimization strategies can also be applied to AE analysis since it is a *must* analysis, too. Figure 3.12 shows the execution time of Succinct Solve running on data flow specification and optimized flow logic specification of VB analysis and AE analysis. It can be seen that the optimization of flow logic clause will increase the efficiency. This increase is by degrees of the complexity polynomial.

3.6.2 Strategy to Write Flow Logic Specification from Data Flow Specification

In this section, we briefly explain how to write flow logic specification for bit-vector framework analyses. Actually, kill and gen components in data flow analysis can be used to the re-writing.

Again, we split the information into two parts, inter-statement, which is related to kill and gen components and intra-statement, which is related to the flow structure of the program.

inter-statement

For each statement, we do it in two steps and below forward analysis is used as example. For backward analysis, just change $\text{Analysis}_\bullet(l)$ to $\text{Analysis}_\circ(l)$.

- information becomes valid after executing the statement, which corresponding to gen components.
  $\forall\ x \in \text{gen}(l) \Rightarrow x \in\text{Analysis}_\bullet(l)$
- which information originally held at entry (exit) point becomes un-held when comes to exit (entry) point. This is roughly corresponds to kill component, except that these kill components are select from the entry (exit) point.
  $\forall\ x \in \text{Analysis}_\circ(l) \land x \notin \text{kill}(l) \Rightarrow x \in\text{Analysis}_\bullet(l)$

In practical, $x \in \text{gen}(l)$ should be detailized to x satisfied the condition of gen and $x \notin \text{kill}(l)$ to x does not satisfied the condition of kill.

intra-statement

How to specify the information relationship depends on whether it is a forward or backward analysis. In this part, we shall use the word $\textit{pair}(l_1, l_2)$ to represent $\text{flow}(l_1, l_2)$ for forward analysis and represent $\text{flow}^R(l_1, l_2)$ for backward analysis. We also distinguish between *may* and *must* analysis. The reason is that operator of *may* analysis is $\sqcup$, which corresponding to $\subseteq$ in practical, while $\sqcap$ is the operator of *must* analysis corresponding to $\Subset$. The following table shows the relationship between l_1 and l_2 for $pair(l_1, l_2)$.

	$forward$	$backward$
may	$\text{Analysis}_\bullet(l_1) \subseteq \text{Analysis}_\circ(l_2)$	$\text{Analysis}_\circ(l_1) \subseteq \text{Analysis}_\bullet(l_2)$
must	$\text{Analysis}_\circ(l_2) \Subset \text{Analysis}_\bullet(l_1)$	$\text{Analysis}_\bullet(l_2) \Subset \text{Analysis}_\circ(l_1)$

In case of *forward may* analysis, all the information at exit of l_1 becomes a part of information at entry of l_2. In case of *backward may* analysis, information at the entry of l_1 is a subset of information held at exit of l_2. This relationship is similar for *must* analysis, except that symbol $\Subset$ is used instead of $\subseteq$ because of the nature of the $\sqcap$ operator.

Chapter 4

Monotone Framework

In last chapter, we have presented some classical analyses belonging to bit-vector framework. Now let's go further from bit-vector framework to a wider range - monotone framework.
Monotone framework is a more general concept than bit-vector framework in the sense that

Property Space Property space of bit-vector framework is $\mathcal{P}(D)$ for a finite set D, while for monotone framework it is a complete lattice $(L, \sqsubseteq)$

Transfer Function Bit-vector analyses require that the transfer functions have the form

$$Analysis_\bullet(l) \quad = \quad (Analysis_\circ(l) \backslash \text{kill}(l)) \cup \text{gen}(l)$$

while in monotone framework, it is only required that each transfer function is monotone, meaning that $l \sqsubseteq l'$ implies $f(l) \sqsubseteq f(l')$. The data flow equations for analyses within monotone framework are,

$$Analysis_\circ(l) \quad = \quad \begin{cases} \imath & \text{if } l \in E \\ \sqcup\{Analysis_\bullet(l) \mid (l', l) \in F\} & \text{otherwise} \end{cases} \tag{4.1}$$

$$Analysis_\bullet(l) \quad = \quad f(Analysis_\circ(l)) \tag{4.2}$$

Some classical monotone analyses include detection of signs analysis, constant propagation analysis and array bound analysis, etc. The data flow specifications of these analyses have been well-developed. In this chapter, we present flow logic specifications of these analyses. Specification of detection of signs analysis is presented first and the other two analyses are developed based on it.

4.1 Detection of Signs Analysis

Detection of Sign Analysis (DS for short) is used to determine:

> At each program point, whether a variable is negative, zero or positive (whether it's sign is -, 0 or +)

The sign information of a variable may be further used for, say array bound analysis.

Example 4.1 Consider the program piece:

$$[x := 1]^1;\ [y := 2]^2;\ [z := x + y]^3;\ [x := -z]^4$$

At block 3, z has the sign "$+$" and x has the sign "$-$" at block 4.

4.1.1 Specification

Detection of signs analysis is a *forward may* analysis.
The flow logic analysis for detection of signs analysis comprises judgments for elementary block B and statement s, which are of the form

$$(\mathrm{DS}_\circ, \mathrm{DS}_\bullet) \models B \text{ and } (\mathrm{DS}_\circ, \mathrm{DS}_\bullet) \models s$$

Like before, $\mathrm{DS}_\circ$ and $\mathrm{DS}_\bullet$ are abstract caches, which map variables to their signs at each block.

$$\mathrm{DS}_\circ, \mathrm{DS}_\bullet \in \widehat{Cache} = \mathrm{Lab}_* \to (\mathrm{Var}_* \to \mathcal{P}(\mathrm{Sign}))$$

Sign $= \{-, 0, +\}$ includes all the possible signs of variables.

Specifications for elementary blocks and statements are presented in Table 4.1.1 and explained below. In the case a block is an assignment statement, all the variables except assignee preserve their signs information held at the entry of block; the assignee, if it is not of the form $A[e]$, then has the signs of the expression on the right-hand side of the statement. In the case the assignee is $A[e]$, all the elements of array A will have the three possible signs. In the case of skip statements and conditionals, the information at the exit of the block is exactly the same as at the entry, since no sign of variables has been changed.
The signs of expressions are determined by function $\mathcal{A}_{DS}$, the functionality of which is,

$$\mathcal{A}_{DS}: \quad \mathrm{Exp} \to (\widehat{State}_{DS} \to \mathrm{Sign})$$

$\mathcal{A}_{DS}$ is defined as,

$$\boxed{\begin{array}{lcl}
\mathcal{A}_{DS}[n]\hat{\sigma} & = & \begin{cases} \{-\} & \text{if } n < 0 \\ \{0\} & \text{if } n = 0 \\ \{+\} & \text{if } n > 0 \end{cases} \\
\mathcal{A}_{DS}[x]\hat{\sigma} & = & \hat{\sigma}(x) \\
\mathcal{A}_{DS}[A[n]]\hat{\sigma} & = & \hat{\sigma}(A[n]) \\
\mathcal{A}_{DS}[A[e]]\hat{\sigma} & = & \{-, 0, +\} \\
\mathcal{A}_{DS}[e_1 + e_2]\hat{\sigma} & = & \mathcal{A}_{DS}[e_1]\hat{\sigma} \mathbin{\hat{+}} \mathcal{A}_{DS}[e_2]\hat{\sigma}
\end{array}}$$

where $\hat{\sigma} \in \widehat{State}_{DS}$ is an *abstract state* to specify for each variable its signs.

$$\hat{\sigma} \in \text{Var}_* \rightarrow \mathcal{P}(\text{Sign})$$

In the specifications of Table 4.1.1, $\text{DS}_\circ(l)(l \in \text{Lab}_*)$ works as the $\hat{\sigma}$. For example, $\text{DS}_\circ(l, x) = \{0, +\}$ indicates x is larger or equal than zero at block l.

The sign of constant n depends on whether n is a negative, positive number or number zero. Sign of variable x or $A[n]$ is determined by looking up the abstract cache $\text{DS}_\circ$, which stores the sign information about variables at each block. We approximate the sign of expression-index array, say $A[e]$, to all the three possibilities, since usually the value of e is hard to estimate. The sign of the addition of two expressions are then determined by applying $\hat{+}$ to the signs of both expressions.
Function symbol $\hat{+}$ corresponds to operator $+$, where $\hat{+}$: Sign $\times$ Sign $\rightarrow \mathcal{P}$(Sign) is given by:

$$S_1 \hat{+} S_2 = \{s_1 \hat{+} s_2 \mid s_1 \in S_1, s_2 \in S_2\}$$

$\hat{+}$	$-$	0	$+$
$-$	$\{-\}$	$\{-\}$	$\{-, 0, +\}$
0	$\{-\}$	$\{0\}$	$\{+\}$
$+$	$\{-, 0, +\}$	$\{+\}$	$\{+\}$

where S_1, $S_2 \subseteq$ Sign.
The first row and column of the table list the signs of operantes, while the corresponding cells contain all the possible signs of the result of addition; simply mathematic knowledge tells that the sum of a negative (positive) number and zero can only be negative (positive). The addition of a negative and a positive number could be either negative, zero or positive. In this thesis, we only consider about $+$, other operators, like $-$, $\times$, $\div$ could be defined in a similar way.

$(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models [x := e]^l$	iff $\forall v\ v \in \mathcal{A}_{DS}[e](\mathrm{DS}_\circ(l)) \Rightarrow (x,v) \in \mathrm{DS}_\bullet(l)$ $\forall y\ \forall v\ (y,v) \in \mathrm{DS}_\circ(l) \wedge y \neq x \Rightarrow (y,v) \in \mathrm{DS}_\bullet(l))$
$(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models [A[n] := e]^l$	iff $\forall v\ v \in \mathcal{A}_{DS}[e](\mathrm{DS}_\circ(l)) \Rightarrow (A[n],v) \in \mathrm{DS}_\bullet(l)$ $\forall y\ \forall v\ (y,v) \in \mathrm{DS}_\circ(l) \wedge y \neq A[n] \Rightarrow (y,v) \in \mathrm{DS}_\bullet(l))$
$(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models [A[e] := e']^l$	iff $\forall i\ (A,i) \in \mathrm{Var}_* \Rightarrow \{(A[i],-),(A[i],0),(A[i],+)\} \subseteq \mathrm{DS}_\bullet(l)$ $\forall y\ \forall v\ \forall i\ (y,v) \in \mathrm{DS}_\circ(l) \wedge y \neq A[i] \Rightarrow (y,v) \in \mathrm{DS}_\bullet(l))$
$(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models [Skip]^l$	iff $\mathrm{DS}_\circ(l) = \mathrm{DS}_\bullet(l)$
$(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models [e]^l$	iff $\mathrm{DS}_\circ(l) = \mathrm{DS}_\bullet(l)$
$(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models s_1; s_2$	iff $(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models s_1 \wedge$ $(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models s_2 \wedge$ $\forall l\ (l \in \mathrm{final}(s_1) \Rightarrow \mathrm{DS}_\circ(\mathrm{init}(s_2)) \subseteq \mathrm{DS}_\bullet(l))$
$(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models$ If $[e]^l$ Then s_1 Else s_2	iff $(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models [e]^l \wedge$ $(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models s_1 \wedge$ $(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models s_2 \wedge$ $\mathrm{DS}_\circ(\mathrm{init}(s_1)) \subseteq \mathrm{DS}_\bullet(l) \wedge$ $\mathrm{DS}_\circ(\mathrm{init}(s_2)) \subseteq \mathrm{DS}_\bullet(l)$
$(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models$ While $[e]^l$ Do s	iff $(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models [e]^l \wedge$ $(\mathrm{DS}_\circ,\mathrm{DS}_\bullet) \models s \wedge$ $\mathrm{DS}_\circ(\mathrm{init}(s)) \subseteq \mathrm{DS}_\bullet(l) \wedge$ $\forall l'(l' \in \mathrm{final}(s) \Rightarrow \mathrm{DS}_\circ(l) \subseteq \mathrm{DS}_\bullet(l'))$

Table 4.1: Control Flow Analysis of DS for Elementary Blocks and Statements

Specifications for statements of detection of signs analysis are similar to reaching definition analysis; each statement needs to satisfy the analysis result. Furthermore, information flowing between statements have the subset relationship. Suppose flow(l_1, l_2) exists, information held at the exit of l_1 is a part of information at entry of l_2, hence

$$\mathrm{DS}_\circ(l_1) \subseteq \mathrm{DS}_\bullet(l_2)$$

4.1.2 Implementation

Before going into the details of implementation, let's take a look at predicates used in the generated clause.

Predicates	**Meaning**
DSIN(l, x, s)	at the entry of l, x has the sign s, viz. $(x, s) \in \mathrm{DS}_\circ(l)$
DSOUT(l, x, s)	at the exit of l, x has the sign s, viz. $(x, s) \in \mathrm{DS}_\bullet(l)$
ADD(s_1, s_2, s_3)	the sum of numbers with sign s_1 and s_2 has the sign s_3

Predicate symbol ADD here corresponds to $\hat{+}$. According to the definition of $\hat{+}$, the clause below about predicate ADD is generated

$$\begin{aligned}&\mathrm{ADD}(-,-,-) \wedge \mathrm{ADD}(-,0,-) \wedge \mathrm{ADD}(-,+,-) \wedge \mathrm{ADD}(-,+,0) \wedge \mathrm{ADD}(-,+,+) \wedge\\ &\mathrm{ADD}(0,-,-) \wedge \mathrm{ADD}(0,0,0) \wedge \mathrm{ADD}(0,+,+) \wedge\\ &\mathrm{ADD}(+,-,-) \wedge \mathrm{ADD}(+,-,0) \wedge \mathrm{ADD}(+,-,+) \wedge \mathrm{ADD}(+,0,+) \wedge \mathrm{ADD}(+,+,+)\end{aligned}$$

At the very beginning, there is no sign information about variable. As the analyzer comes to the first assignment statement, information becomes available. Recall that $\mathrm{DS}_\circ(l)$ works as the abstract state $\hat{\sigma}$ to evaluate the possible signs of an expression. It is implemented in the following way,

$$\boxed{\begin{array}{lcl}\mathcal{A}_{DS}[n]\mathrm{DS}_\circ(l) & = & \begin{cases}\{-\} & \text{if } n < 0\\ \{0\} & \text{if } n = 0\\ \{+\} & \text{if } n > 0\end{cases}\\ \mathcal{A}_{DS}[x]\mathrm{DS}_\circ(l) & = & \{v \mid (x, v) \in \mathrm{DS}_\circ(l)\}\\ \mathcal{A}_{DS}[A[n]]\mathrm{DS}_\circ(l) & = & \{v \mid (A[n], v) \in \mathrm{DS}_\circ(l)\}\\ \mathcal{A}_{DS}[A[e]]\mathrm{DS}_\circ(l) & = & \{-, 0, +\}\\ \mathcal{A}_{DS}[e_1 + e_2]\mathrm{DS}_\circ(l) & = & \{v \mid \mathrm{ADD}(v_1, v_2, v)\}\\ & & \text{where } v_1 \in \mathcal{A}_{DS}[e_1]\mathrm{DS}_\circ(l) \text{ and } v_2 \in \mathcal{A}_{DS}[e_2]\mathrm{DS}_\circ(l)\end{array}}$$

which corresponds to the definition of $\mathcal{A}_{DS}$. Clearly, $\mathrm{DS}_\circ(l)$ stored the signs information of each variable at block l, thus v is a possible of variable x or $A[n]$ if (x, v) or $(A[n], v)$

$\in \mathrm{DS}_\circ(l)$. Note that the signs of expression $e_1 + e_2$ is determined in a recursive way that suppose v_1 and v_2 are the signs of e_1 and e_2, respectively, then v is a sign of $e_1 + e_2$ providing that the addition of e_1 and e_2 gives rise to a value having a sign v.
The type of each block determines intra-statement information, while the relationship between blocks determines inter-statement information. Information at one entry of certain block is gathered from all its predecessors. For flow(l_1, l_2), the following clause is generated

$$\forall x\ \forall s\ \mathrm{DSOUT}(l_1, x, s) \Rightarrow \mathrm{DSIN}(l_2, x, s)$$

Example 4.2 Consider the program piece

$$[x := 1]^1; [y := 2]^2; [A[y] := 2]^3; [A[2] := x + y]^4;$$

where array A is declared as *Array A of* [1..2]. The clause generated for the above program is,

$\mathrm{ADD}(N,N,N)\wedge\mathrm{ADD}(N,Z,N)\wedge\mathrm{ADD}(N,P,N)\wedge\mathrm{ADD}(N,P,Z)\wedge\mathrm{ADD}(N,P,P)\wedge$
$\mathrm{ADD}(Z,N,N)\wedge\mathrm{ADD}(Z,Z,Z)\wedge\mathrm{ADD}(Z,P,P)\wedge$
$\mathrm{ADD}(P,N,N)\wedge\mathrm{ADD}(P,N,Z)\wedge\mathrm{ADD}(P,N,P)\wedge\mathrm{ADD}(P,Z,P)\wedge\mathrm{ADD}(P,P,P)\wedge$

$\mathrm{DSOUT}(L1,x,P)\wedge$
$(\forall y\ \forall v\ \mathrm{DSIN}(L1,y,v) \wedge y \neq x \Rightarrow \mathrm{DSOUT}(L1,y,v))\wedge$

$\mathrm{DSOUT}(L2,y,P)\wedge$
$(\forall x\ \forall v\ \mathrm{DSIN}(L2,x,v) \wedge x \neq y \Rightarrow \mathrm{DSOUT}(L2,x,v))\wedge$

$\mathrm{DSOUT}(L3,A'1,N)\wedge\mathrm{DSOUT}(L3,A'1,Z)\wedge\mathrm{DSOUT}(L3,A'1,P)\wedge$
$\mathrm{DSOUT}(L3,A'2,N)\wedge\mathrm{DSOUT}(L3,A'2,Z)\wedge\mathrm{DSOUT}(L3,A'2,P)\wedge$
$(\forall x\ \forall v\ \mathrm{DSIN}(L3,x,v) \wedge x \neq A'1 \wedge x \neq A'2 \Rightarrow \mathrm{DSOUT}(L3,x,v))\wedge$

$(\forall v1\ \forall v2\ \forall v\ \mathrm{DSIN}(L4,x,v1)\wedge\mathrm{DSIN}(L4,y,v2)\wedge\mathrm{ADD}(v1,v2,v) \Rightarrow \mathrm{DSOUT}(L4,A'2,v))\wedge$
$(\forall x\ \forall v\ \mathrm{DSIN}(L4,x,v) \wedge x \neq A'2 \Rightarrow \mathrm{DSOUT}(L4,x,v))\wedge$

$(\forall x\ \forall v\ \mathrm{DSOUT}(L1,x,v) \Rightarrow \mathrm{DSIN}(L2,x,v))\wedge$
$(\forall x\ \forall v\ \mathrm{DSOUT}(L2,x,v) \Rightarrow \mathrm{DSIN}(L3,x,v))\wedge$
$(\forall x\ \forall v\ \mathrm{DSOUT}(L3,x,v) \Rightarrow \mathrm{DSIN}(L4,x,v))$

where N(for negative),Z(for zero),P(for positive) represent $-, 0, +$, respectively; $A'i$ represents $A[i]$ and Li represents the block with label i. The summarized result given by

Succinct Solver is,

$DS_\circ$	x	y	$A[1]$	$A[2]$	$DS_\bullet$	x	y	$A[1]$	$A[2]$
$L1$					$L1$	$\{+\}$			
$L2$	$\{+\}$				$L2$	$\{+\}$	$\{+\}$		
$L3$	$\{+\}$	$\{+\}$			$L3$	$\{+\}$	$\{+\}$	$\{-,0,+\}$	$\{-,0,+\}$
$L4$	$\{+\}$	$\{+\}$	$\{-,0,+\}$	$\{-,0,+\}$	$L4$	$\{+\}$	$\{+\}$	$\{-,0,+\}$	$\{+\}$

where blank cells indicate no information is available about the signs of variables.

4.2 Constant Propagation Analysis

Constant propagation is similar to but a bit more complicated than the detection of signs analysis in the sense that DS analysis only approximates the sign information about every variable at each block while constant propagation analysis approximates not the sign but the constant value of each variable.
Constant Propagation Analysis (CP for short) is used to indicate:

> For each program point, whether or not a variable has a constant value whenever execution reaches that point.

The information obtained from the analysis can be used, for instance, as the basis for an optimization Constant Folding, which replace all uses of variables by constant value.

Example 4.3 Consider the program piece:

$$[x := 1]^1;\ \text{If}[x < 1]^2 \text{ Then } [y := x + y]^3 \text{ Else } [z := x + z]^4;$$

At block 3 and 4, x has constant value 1, hence these two blocks may be rewritten to $[y := 1 + y]^3$ and $[z := 1 + z]^4$.

4.2.1 Specification

Constant propagation analysis is *forward must* analysis.
A variable had constant value means that it has either an integer or boolean value. In order to capture the case where a variable is non-constant, symbol $\top$ is introduced. Furthermore, we shall use symbol $\perp$ to indicate no information is available. Thus abstract caches $\text{CP}_\circ$ and $\text{CP}_\bullet$ are defined as

$$\text{CP}_\circ, \text{CP}_\bullet \in \widehat{Cache} = \text{Lab}_* \to (\text{Var}_* \to (\mathbb{Z} \cup \mathbb{T})_\perp^\top)$$

where $(\mathbb{Z} \cup \mathbb{T}_{\perp}^{\top}) = (\mathbb{Z} \cup \mathbb{T} \cup \{\top, \perp\})$
Operation on $(\mathbb{Z} \cup \mathbb{T})_{\perp}^{\top}$ is then defined by taking $v_1\ \hat{op}\ v_2 = v_1\ op\ v_2$ if $v_1, v_2 \in \mathbb{Z}$, $v_1\ \hat{op}\ v_2 = \perp$ if $v_1 = \perp$ or $v_2 = \perp$ and $v_1\ \hat{op}\ v_2 = \top$ otherwise.

$\hat{op}$	$\perp$	v_1	$\top$
$\perp$	$\perp$	$\perp$	$\perp$
v_2	$\perp$	$v_1\ op\ v_2$	$\top$
$\top$	$\perp$	$\top$	$\top$

where $op \in \{+ , - , \times , \div \}$
For each node, information from different predecessors are combined by $\sqcup$(Join) operator.

$\sqcup$	$\perp$	v_1	$\top$
$\perp$	$\perp$	$\perp$	$\perp$
v_2	$\perp$	v_1 (if $v_1 = v_2$)	v_2
		$\perp$ (if $v_1 \neq v_2$)	
$\top$	$\perp$	v_1	$\top$

where v_1 and v_2 represent constant values.
The analysis result is the pair $(\text{CP}_\circ,\text{CP}_\bullet)$. An elementary block or a statement is acceptable if and only if it satisfies the result pair.

$$(\text{CP}_\circ, \text{CP}_\bullet) \models B \text{ and } (\text{CP}_\circ, \text{CP}_\bullet) \models s$$

The detail specifications are listed in Table 4.2.1.
Specifications for assignment statements are different depending on the kinds of assignees. But in assignment statements, all the variables except assignee preserve their information held at the entry of that block, since only the assignee may change its value. The constant information of assignees is then decided by the constant information of the right-hand side expressions. In case of $[A[e] := \ldots]^l$, all the elements of array A are mapped to $\top$. Its safe to do that because the exactly value of e is undetermined. In case of skip statement and conditional e, exit information is exactly the same at entry information, because no variable has changed its value.
Similar to DS analysis, we define a function $\mathcal{A}_{CP}$ to evaluate the whether an expression has a constant value, the functionality of which is,

$$\mathcal{A}_{CP}: \quad \text{Exp} \rightarrow (\widehat{State}_{CP} \rightarrow \text{Value})$$

where Value $\in (\mathbb{Z} \cup \mathbb{T})_{\perp}^{\top}$. $\mathcal{A}_{CP}$ is defined as,

$$\boxed{\begin{array}{lcl} \mathcal{A}_{CP}[n]\hat{\sigma} & = & n \\ \mathcal{A}_{CP}[x]\hat{\sigma} & = & \hat{\sigma}(x) \\ \mathcal{A}_{CP}[A[n]]\hat{\sigma} & = & \hat{\sigma}(A[n]) \\ \mathcal{A}_{CP}[A[e]]\hat{\sigma} & = & \top \\ \mathcal{A}_{CP}[e_1\ op\ e_2]\hat{\sigma} & = & \top \end{array}}$$

$\hat{\sigma} \in \widehat{State}_{CP}$ is an *abstract state* and for each variable x, $\hat{\sigma}(x)$ will give information about whether or not x is a constant and in the latter case which constant.

$$\hat{\sigma} \in \mathrm{Var}_* \rightarrow \mathrm{Value}$$

Values of x and $A[n]$ are simply determined by looking up $\hat{\sigma}$; any expression-indexed array element is thought to have $\top$ and rather than $\mathcal{A}_{CP}[e_1]\hat{\sigma}\ \hat{op}\ \mathcal{A}_{CP}[e_2]\hat{\sigma}$, expression $e_1\ op\ e_2$ will be approximated to $\top$. This simplifies the specification and implementation but causes losing of preciseness. The reason for doing in this way is because Succinct Solve lacks the ability to do arithmetic calculations. Therefore the computation work has to be done by the analyzer to calculate the operation result of any two values. But the problem is the generator has no idea about the value of each variable, thus the only way for it to compute is to enumerate all the constant values appeared in the program and apply the operation to each of the two values. This method is not only un-efficient but even un-computable in some cases.
Again, $\mathrm{CP}_\circ$ works as the $\hat{\sigma}$ in the specifications.

In the lower part of Table 4.2.1, specifications for statements are presented. For each node, information come from different predecessors are combined by $\sqcup$,

$$\mathrm{CP}_\circ(l) = \sqcup \mathrm{CP}_\bullet(l_i)$$

where flow(l_i, l) exists. CP is a *must* analysis, and as mentioned in VB analysis the formula of the above form can not be implemented directly, rather it should be re-written to

$$\mathrm{CP}_\circ(l_2) = \mathrm{CP}_\circ(l_2) \sqcup \mathrm{CP}_\bullet(l_1)$$

for all l_i, where flow(l_1, l_2) exists. In the specifications, we shall use

$$\mathrm{CP}_\bullet(l_1) \subseteqq \mathrm{CP}_\circ(l_2)$$

to represent it.

$(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models [x := e]^l$	iff $\forall v\ v \in \mathcal{A}_{CP}[x](\mathrm{CP}_\circ(l)) \Rightarrow (x,v) \in\mathrm{CP}_\bullet(l)$ $\forall y\ \forall v\ (y,v) \in\mathrm{CP}_\circ(l) \wedge y \neq x \Rightarrow (y,v) \in\mathrm{CP}_\bullet(l))$
$(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models [A[n] := e]^l$	iff $\forall v\ v \in \mathcal{A}_{CP}[A[n]](\mathrm{CP}_\circ(l)) \Rightarrow (A[n],v) \in\mathrm{CP}_\bullet(l)$ $\forall y\ \forall v\ (y,v) \in\mathrm{CP}_\circ(l) \wedge y \neq A[n] \Rightarrow (y,v) \in\mathrm{CP}_\bullet(l))$
$(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models [A[e] := e']^l$	iff $\forall i\ (A,i) \in\mathrm{Var}_* \Rightarrow (A(i),\top) \in\mathrm{CP}_\bullet(l)\wedge$ $\forall y\ \forall v\ \forall i\ (y,v) \in\mathrm{CP}_\circ(l) \wedge y \neq A[i] \Rightarrow (y,v) \in\mathrm{CP}_\bullet(l))$
$(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models [Skip]^l$	iff $\mathrm{CP}_\circ(l) =\mathrm{CP}_\bullet(l)$
$(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models [e]^l$	iff $\mathrm{CP}_\circ(l) =\mathrm{CP}_\bullet(l)$
$(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models s_1; s_2$	iff $(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models s_1\wedge$ $(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models s_2\wedge$ $\forall l\ (l \in\mathrm{final}(s_1) \Rightarrow\mathrm{CP}_\circ(\mathrm{init}(s_2)) \subseteqq\mathrm{CP}_\bullet(l))$
$(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models$ If $[e]^l$ Then s_1 Else s_2	iff $(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models [e]^l\wedge$ $(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models s_1\wedge$ $(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models s_2\wedge$ $\mathrm{CP}_\circ(\mathrm{init}(s_1)) \subseteqq\mathrm{CP}_\bullet(l)\wedge$ $\mathrm{CP}_\circ(\mathrm{init}(s_2)) \subseteqq\mathrm{CP}_\bullet(l)$
$(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models$ While $[e]^l$ Do s	iff $(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models [e]^l\wedge$ $(\mathrm{CP}_\circ,\mathrm{CP}_\bullet) \models s\wedge$ $\mathrm{CP}_\circ(\mathrm{init}(s)) \subseteqq\mathrm{CP}_\bullet(l)\wedge$ $\forall l'(l' \in\mathrm{final}(s) \Rightarrow\mathrm{CP}_\circ(l) \subseteqq\mathrm{CP}_\bullet(l'))$

Table 4.2: Control Flow Analysis of CP for Elementary Blocks and Statements

4.2.2 Implementation

Predicates used in clauses and their meanings are listed in the below table.

Predicate	Meaning
LAB(l)	l is a label of the program $l \in$blocks(p)
JOIN(v_1, v_2, v_3)	the join of v_1 and v_2 gives rise to v_3 $v_1 \sqcup v_2 = v_3$
R(x)	x is a constant or non-constant value $x \in (\mathbb{Z} \cup \mathbb{T}_{\perp}^{\top})$
CPIN(l, x, v)	at entry of block l, x has value v where v could be either constant or non-constant value
CPOUT(l, x, v)	at exit of l, x has value v where v could be either constant or non-constant value

Initially, we assume all the variables at each block of the program have non-constant values, therefore they are mapped to $\top$ by the clause

$$(\forall\ l.\ \text{LAB}(l)\ \Rightarrow\ (\forall\ x.\ \text{VAR}(x) \Rightarrow\ \text{CPIN}(l,\ x,\ \text{TOP})))$$

Information flows from one block to another, if there exists a flow between them. The information at the entry of a block is always the $\sqcup$ of all the information held at the exit of its predecessors. Predicate symbol JOIN is used to represent this operator. Interpretations of JOIN are computed by the following clauses,

$\forall\ x.\ \text{R}(x) \Rightarrow \text{JOIN}(\text{R}(\text{TOP}),\ \text{R}(x),\ \text{R}(x))$
$\forall\ x.\ \text{R}(x) \Rightarrow \text{JOIN}(\text{R}(x),\ \text{R}(\text{TOP}),\ \text{R}(x))$
$\forall\ x.\ \text{R}(x) \Rightarrow \text{JOIN}(\text{R}(\text{BOT}),\ \text{R}(x),\ \text{R}(\text{BOT}))$
$\forall\ x.\ \text{R}(x) \Rightarrow \text{JOIN}(\text{R}(x),\ \text{R}(\text{BOT}),\ \text{R}(\text{BOT}))$
$\forall\ x.\ \forall\ y.\text{R}(x) \wedge \text{R}(y) \wedge x = y \Rightarrow \text{JOIN}(\text{R}(x),\ \text{R}(y),\ \text{R}(x))$
$\forall\ x.\ \forall\ y.\text{R}(x) \wedge \text{R}(y) \wedge x \neq y \wedge x \neq \text{TOP} \wedge y \neq \text{TOP} \wedge x \neq \text{BOT} \wedge y \neq \text{BOT}$
$\Rightarrow \text{JOIN}(\text{R}(x),\ \text{R}(y),\ \text{R}(\text{BOT}))$

The first four clauses say that the Join of $\top$ and x, no matter what value x has, is still x and the Join of $\perp$ and x is $\perp$. The Join of two equal values is that value and the Join of two un-equal constant values are $\perp$.
Now we are able to specify the clause for the relationship between l_1 and l_2, if flow(l_1, l_2 exists.

$$\forall x\ \forall v_1\ \forall v_2\ \forall v_3\ \text{CP}_{\bullet}(l_1, x, v_1) \wedge \text{CP}_{\circ}(l_2, x, v_2) \wedge \text{JOIN}(v_1, v_2, v_3) \Rightarrow \text{CP}_{\circ}(l_2, x, v_3)$$

This clause could be read as: for any variable, say x, if at the exit of l_1 and the entry of l_2, it has value v_1 and v_2, respectively, then at the entry of l_2, x will have a new value v_3, providing that $v_3 = v_1 \sqcup v_2$.

Example 4.4 Consider the program piece

$[y := 3]^1$;
If $[x > 5]^2$ Then $[x := 3]^3$ Else $[x := y]^4$;
While $[A[x] > 3]^5$ Do $[A[1] := x + 4]^6$;
$[A[x] := 3]^7$

Constant propagation analysis will create the following clause,

LAB(L1)$\wedge$LAB(L2)$\wedge$LAB(L3)$\wedge$LAB(L4)$\wedge$ LAB(L5)$\wedge$LAB(L6)$\wedge$LAB(L7)$\wedge$
VAR(x)$\wedge$VAR(y)$\wedge$VAR($A'1$)$\wedge$VAR($A'2$)$\wedge$
R(3)$\wedge$R(4)$\wedge$
($\forall x$ R(x)$\Rightarrow$ JOIN(R(TOP), R(x), R(x)))$\wedge$
($\forall x$ R(x)$\Rightarrow$ JOIN(R(x), R(TOP), R(x)))$\wedge$
($\forall x$ R(x)$\Rightarrow$ JOIN(R(BOT), R(x), R(BOT)))$\wedge$
($\forall x$ R(x)$\Rightarrow$ JOIN(R(x), R(BOT), R(BOT)))$\wedge$
($\forall x$ $\forall y$ R(x)$\wedge$R(y)$\wedge x = y \Rightarrow$ JOIN(R(x), R(y), R(x)))$\wedge$
($\forall x$ $\forall y$ R(x)$\wedge$R(y)$\wedge x \neq y \wedge x \neq$TOP$\wedge y \neq$TOP$\wedge x \neq$BOT$\wedge y \neq$ BOT$\Rightarrow$ JOIN(R(x), R(y), R(BOT)))$\wedge$
($\forall l$ LAB(l)$\Rightarrow$ ($\forall x$ VAR(x)$\Rightarrow$CPIN(l,x,R(TOP))))$\wedge$

CPOUT(L1,y,R(3))$\wedge$
($\forall x$ $\forall v$ CPIN(L1,x, v)$\wedge$x$\neq$y $\Rightarrow$ CPOUT(L1,x, v))$\wedge$

($\forall x$ $\forall v$ CPIN(L2,x, v)$\Rightarrow$ CPOUT(L2,x, v))$\wedge$

CPOUT(L3,x,R(3))$\wedge$
($\forall y$ $\forall v$ CPIN(L3,y, v)$\wedge$y$\neq$x $\Rightarrow$ CPOUT(L3,y, v))$\wedge$

($\forall v$ CPIN(L4,y, v)$\Rightarrow$ CPOUT(L4,x, v))$\wedge$
($\forall y$ $\forall v$ CPIN(L4,y, v)$\wedge y \neq x \Rightarrow$ CPOUT(L4,y, v))$\wedge$

($\forall x$ $\forall v$ CPIN(L5,x, v)$\Rightarrow$ CPOUT(L5,x, v))$\wedge$

CPOUT(L6,$A'1$,R(TOP))$\wedge$
($\forall x$ $\forall v$ CPIN(L6,x, v)$\wedge x \neq A'1 \Rightarrow$ CPOUT(L6,x, v))$\wedge$

CPOUT(L7,$A'1$,R(TOP))$\wedge$

CPOUT(L7,$A'2$,R(TOP))$\wedge$
($\forall x\ \forall v$ CPIN(L7,x, v)$\wedge x \neq A'1 \wedge x \neq A'2 \Rightarrow$ CPOUT(L7,x, v))$\wedge$

($\forall x\ \forall v$ CPOUT(L1,x, v)$\Rightarrow$ CPIN(L2,x, v))$\wedge$
($\forall x\ \forall v$ CPOUT(L2,x, v)$\Rightarrow$ CPIN(L3,x, v))$\wedge$
($\forall x\ \forall v$ CPOUT(L2,x, v)$\Rightarrow$ CPIN(L4,x, v))$\wedge$
($\forall x\ \forall v1\ \forall v2\ \forall v3$ CPOUT(L3,$x, v1$)$\wedge$CPIN(L5,$x, v2$)$\wedge$JOIN($v1, v2, v3$)$\Rightarrow$CPIN(L5,$x, v3$))$\wedge$
($\forall x\ \forall v1\ \forall v2\ \forall v3$ CPOUT(L4,$x, v1$)$\wedge$CPIN(L5,$x, v2$)$\wedge$JOIN($v1, v2, v3$)$\Rightarrow$CPIN(L5,$x, v3$))$\wedge$
($\forall x\ \forall v1\ \forall v2\ \forall v3$ CPOUT(L6,$x, v1$)$\wedge$CPIN(L5,$x, v2$)$\wedge$JOIN($v1, v2, v3$)$\Rightarrow$CPIN(L5,$x, v3$))$\wedge$
($\forall x\ \forall v$ CPOUT(L5,x, v)$\Rightarrow$ CPIN(L6,x, v))$\wedge$
($\forall x\ \forall v$ CPOUT(L5,x, v)$\Rightarrow$ CPIN(L7,x, v))

The solving result given by Succinct Solver about the above clause could be summarized to the below table,

$CP_\circ$	x	y	$A[1]$	$A[2]$	$CP_\bullet$	x	y	$A[1]$	$A[2]$
L1	$\top$	$\top$	$\top$	$\top$	L1	$\top$	3	$\top$	$\top$
L2	$\top$	3	$\top$	$\top$	L2	$\top$	3	$\top$	$\top$
L3	$\top$	3	$\top$	$\top$	L3	3	3	$\top$	$\top$
L4	$\top$	3	$\top$	$\top$	L4	3	3	$\top$	$\top$
L5	3	3	$\top$	$\top$	L5	3	3	$\top$	$\top$
L6	3	3	$\top$	$\top$	L6	3	3	$\top$	$\top$
L7	3	3	$\top$	$\top$	L7	3	3	$\top$	$\top$

4.3 Array Bounds Analysis

Based on detection of signs analysis, we could develop a more complicated analysis-Array bounds analysis. Detection of signs analysis concerns about the signs of each individual variable but could not record the interdependencies between signs of variables (e.g. two variables, say x and y will always have the same sign); while array bounds analysis does better than it in the sense that the analysis results may reflect the relationship between the signs and variables.
Array bounds analysis (AB for short) is used to determine

> both the signs and the difference in magnitude between the possible values of two variables at each program point

The information obtained could be used to tell whether an array index is always within the bounds of the array. If this within-the-bounds information is confirmed, a number of run-time checks may be eliminated.

Example 4.5 Consider the statements,

$$[y := 3]^1;\ \text{If } [x < 10]^2 \text{ Then } [x := y + 1]^3 \text{ Else } [x := y + 5]^4$$

At blocks 3 and 4, both x and y are positive numbers. The magnitude difference x and y is $+1$ at block 3 and is either 0 or $> +1$ at block 4.

4.3.1 Specification

Array bounds analysis is a *forward may* analysis.
Like before, we define two abstract caches $\mathrm{AB}_\circ$ and $\mathrm{AB}_\bullet$, which map a pair of variables to their signs and magnitude difference at each block.

$$\mathrm{AB}_\circ, \mathrm{AB}_\bullet \in \widehat{Cache} = \mathrm{Lab}_* \to ((\mathrm{Var}_* \times \mathrm{Var}_*) \to (\mathrm{Sign} \times \mathrm{Sign} \times \mathrm{Range})$$

where $\mathrm{Sign} = \{-, 0, +\}$ and $\mathrm{Range} = \{< -1, -1, 0, +1, > +1\}$ is defined as the difference between the absolute values of two integers. More precisely, assume the values of the pair of variables are z_1 and z_2 and $z = |z_1| - |z_2|$, then

$$z \in \mathrm{Range} = \begin{cases} < -1 & \text{if } z < -1 \\ -1 & \text{if } z = -1 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z = +1 \\ > +1 & \text{if } z > +1 \end{cases}$$

Elementary blocks and statements must satisfy the analysis result in order to be acceptable.

$$(\mathrm{AB}_\circ, \mathrm{AB}_\bullet) \models B \text{ and } (\mathrm{AB}_\circ, \mathrm{AB}_\bullet) \models s$$

Table 4.3.1 and Table 4.3.1 contain the control flow specifications for elementary blocks and statements, where $\mathrm{ODD}(x)$ and $\mathrm{EVEN}(x)$ indicate whether x is an odd or an even number and $\mathcal{S}(n)$ is used to determine the sign of constant n.
Generally speaking, it needs some intelligence to judge the difference in magnitude of two variable according to an assignment statement. Here we only list some possible forms of expressions on the right-hand side of a assignment statement. In the case the expression is too complicated hence the it's hard to determine the range, we simply set the magnitude difference between assignee x and other variables to all the possibilities, say $< -1, -1, 0, +1$ and $> +1$.
In case of $[x := n]^l$, x will have the sign of n and the ranges between x and other variables could be either < -1,-1,0,+1 or $> +1$. Information between other variables than x

remains the same. In case of $[x := y]^l([x := A[n]]^l)$, information about x is replaced by the information of $y(A[n])$ and ranges between x and other variables are also changed to the ranges between $y(A[n])$ and the variables. In case of $[x := A[e]]^l$, since the value of x is undetermined, the sign and range information about x could be all the possibilities. In case of $[x := y + n]^l$, the difference between x and y could be roughly determined by whether n is 1, odd number or even number. Clearly, when $n = 1$ magnitude difference between x and y could only be $+1$; when n is odd (even), magnitude difference could be either $+1(0)$ or $> +1$. The sign of x is decided by the sign of $y + n$ at block l. In case of $[x := y+z]^l$, $[x := y+A[n]]^l$ and $[x := y+A[e]]^l$ we simply set the sign of x and magnitude difference between x and other variables to all the possibilities. In case of the assignee is expression-indexed array, say $A[e]$, information about all the array elements are set to all the possibilities.

$(\mathrm{AB}_\circ,\mathrm{AB}_\bullet) \models [x := n]^l$	iff $\bullet\ \forall v\ \forall y\ \forall v_1\ \forall v_2\ \forall r\ \forall r_1$ $v = \mathcal{S}(n) \wedge (x,y,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \wedge \mathrm{RANGE}(r_1) \Rightarrow$ $(x,y,s,v_2,r_1) \in \mathrm{AB}_\bullet(l) \wedge$ $\bullet\ \forall y\ \forall z\ \forall v_1\ \forall v_2\ \forall r\ (y,z,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \wedge$ $y \neq x \wedge z \neq x \Rightarrow (y,z,v_1,v_2,r) \in \mathrm{AB}_\bullet(l))$
$(\mathrm{AB}_\circ,\mathrm{AB}_\bullet) \models [x := y]^l$ (or $[x := A[n]]^l$)	iff $\bullet\ \forall v_1\ \forall v_2\ \forall r\ (x,y,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \Rightarrow$ $(x,y,v_2,v_2,0) \in \mathrm{AB}_\bullet(l) \wedge$ $\bullet\ \forall z\ \forall v_1\ \forall v_2\ \forall r\ (y,z,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \wedge z \neq x \Rightarrow$ $(x,z,v_1,v_2,r) \in \mathrm{AB}_\bullet(l) \wedge$ $\bullet\ \forall y\ \forall z\ \forall v_1\ \forall v_2\ \forall r\ (y,z,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \wedge$ $y \neq x \wedge z \neq x \Rightarrow (y,z,v_1,v_2,r) \in \mathrm{AB}_\bullet(l))$
$(\mathrm{AB}_\circ,\mathrm{AB}_\bullet) \models [x := y + n]^l$	iff $\bullet\ n = 1 \wedge \forall v\ \forall v_1\ \forall v_2\ \forall v_3\ \forall r\ (x,y,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \wedge$ $\mathrm{ADD}(v_2,+,v) \Rightarrow (x,y,v,v_2,+1) \in \mathrm{AB}_\bullet(l) \wedge$ $\bullet\ n \neq 1 \wedge \mathrm{ODD}(n) \wedge \forall v\ \forall v_1\ \forall v_2\ \forall r\ (x,y,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \wedge$ $v = \mathcal{S}(n) \wedge \mathrm{ADD}(v_2,v,v_3) \Rightarrow$ $\{(x,y,v_3,v_2,+1),(x,y,v_3,v_2,>+1)\} \subseteq \mathrm{AB}_\bullet(l) \wedge$ $\bullet\ n \neq 1 \wedge \mathrm{EVEN}(n) \wedge \forall v\ \forall v_1\ \forall v_2\ \forall r\ (x,y,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \wedge$ $v = \mathcal{S}(n) \wedge \mathrm{ADD}(v_2,v,v_3) \Rightarrow$ $\{(x,y,v_3,v_2,0),(x,y,v_3,v_2,>+1)\} \subseteq \mathrm{AB}_\bullet(l) \wedge$ $\bullet\ \forall y\ \forall z\ \forall v_1\ \forall v_2\ \forall r\ (y,z,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \wedge$ $y \neq x \wedge z \neq x \Rightarrow (y,z,v_1,v_2,r) \in \mathrm{AB}_\bullet(l))$
$(\mathrm{AB}_\circ,\mathrm{AB}_\bullet) \models [x := others]^l$	iff $\bullet\ \forall v\ \forall r\ \forall v_1\ \forall v_2\ \forall r_1\ \forall y\ \mathrm{SIGN}(v) \wedge \mathrm{RANGE}(r) \wedge$ $(x,y,v_1,v_2,r_1) \in \mathrm{AB}_\circ(l) \Rightarrow (x,y,s,v_2,r) \in \mathrm{AB}_\bullet(l) \wedge$ $\bullet\ \forall y\ \forall z\ \forall v_1\ \forall v_2\ \forall r\ (y,z,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \wedge$ $y \neq x \wedge z \neq x \Rightarrow (y,z,v_1,v_2,r) \in \mathrm{AB}_\bullet(l))$
$(\mathrm{AB}_\circ,\mathrm{AB}_\bullet) \models [A[e] := e']^l$	iff $\bullet\ \forall i\ \forall v\ \forall r\ \forall r_1 \forall y\ \forall v_1\ \forall v_2\ (A,i) \in \mathrm{Var}_* \wedge \mathrm{SIGN}(v) \wedge \mathrm{RANGE}(r) \wedge$ $(A[i],y,v_1,v_2,r_1) \in \mathrm{AB}_\circ(l) \Rightarrow (A[i],y,v,v_2,r) \in \mathrm{AB}_\bullet(l) \wedge$ $\bullet\ \forall i\ \forall y\ \forall z\ \forall v_1\ \forall v_2\ \forall r\ A[i] \in \mathrm{Var}_* \wedge (y,z,v_1,v_2,r) \in \mathrm{AB}_\circ(l) \wedge$ $y \neq A[i] \wedge z \neq A[i] \Rightarrow (y,z,v_1,v_2,r) \in \mathrm{AB}_\bullet(l))$
$(\mathrm{AB}_\circ,\mathrm{AB}_\bullet) \models [Skip]^l$	iff $\mathrm{AB}_\circ(l) = \mathrm{AB}_\bullet(l)$
$(\mathrm{AB}_\circ,\mathrm{AB}_\bullet) \models [e]^l$	iff $\mathrm{AB}_\circ(l) = \mathrm{AB}_\bullet(l)$

Table 4.3: Control Flow Analysis of AB for Elementary Blocks

$(\text{AB}_\circ,\text{AB}_\bullet) \models s_1; s_2$	iff $(\text{AB}_\circ,\text{AB}_\bullet) \models s_1 \wedge$ $(\text{AB}_\circ,\text{AB}_\bullet) \models s_2 \wedge$ $\forall l\ (l \in \text{final}(s_1) \Rightarrow \text{AB}_\circ(\text{init}(s_2)) \subseteq \text{AB}_\bullet(l))$
$(\text{AB}_\circ,\text{AB}_\bullet) \models$ If $[e]^l$ Then s_1 Else s_2	iff $(\text{AB}_\circ,\text{AB}_\bullet) \models [e]^l \wedge$ $(\text{AB}_\circ,\text{AB}_\bullet) \models s_1 \wedge$ $(\text{AB}_\circ,\text{AB}_\bullet) \models s_2 \wedge$ $\text{AB}_\circ(\text{init}(s_1)) \subseteq \text{AB}_\bullet(l) \wedge$ $\text{AB}_\circ(\text{init}(s_2)) \subseteq \text{AB}_\bullet(l)$
$(\text{AB}_\circ,\text{AB}_\bullet) \models$ While $[e]^l$ Do s	iff $(\text{AB}_\circ,\text{AB}_\bullet) \models [e]^l \wedge$ $(\text{AB}_\circ,\text{AB}_\bullet) \models s \wedge$ $\text{AB}_\circ(\text{init}(s)) \subseteq \text{AB}_\bullet(l) \wedge$ $\forall l'(l' \in \text{final}(s) \Rightarrow \text{AB}_\circ(l) \subseteq \text{AB}_\bullet(l'))$

Table 4.4: Control Flow Analysis of AB for Statements

For inter-statement information flow, suppose flow(l_1, l_2) exists, information at the exit of l_1 is a subset of it at the entry of l_2, therefore

$$\text{AB}_\bullet(l_1) \subseteq \text{AB}_\circ(l_2)$$

4.3.2 Implementation

Predicates used in clauses and their meanings are listed in the below table.

Predicate	Meaning
INIT(l)	l is the initial label of the program
ADD(s_1, s_2, s_3)	the sum of numbers with sign s_1 and s_2 has the sign s_3
SIGN(s)	s is a possible sign $s \in \{-, 0, +\}$
RANGE(r)	r is a possible range $r \in \{< -1, -1, 0, +1, > +1\}$
ABIN(l, x, y, s_1, s_2, r)	at entry of block l, x and y have signs s_1 and s_2, respectively and the difference is r
ABOUT(l, x, y, s_1, s_2, r)	at exit of block l, x and y have signs s_1 and s_2, respectively and the difference is r

Before specifying the clauses for blocks and statements, predicates SIGN and RANGE are constructed by the clause

$$\begin{aligned}&\mathrm{SIGN}(-) \wedge \mathrm{SIGN}(0) \wedge \mathrm{SIGN}(+)\wedge\\&\mathrm{RANGE}(< -1) \wedge \mathrm{SIGN}(-1) \wedge \mathrm{SIGN}(0) \wedge \mathrm{SIGN}(+1) \wedge \mathrm{SIGN}(> +1)\end{aligned}$$

At the very beginning, each variable pair at the initial block is mapped to a tuple $(\top, \top, \top)$. Note that to avoid redundancy, (x, y) and (y, x) are regarded as the same pair. For each variable pair, say (x, y), analyzer will generate the clause(assume L1 is the label of initial block)

$$\mathrm{ABIN}(\mathrm{L1}, x, y, \top, \top, \top)$$

Clauses for elementary blocks is determined by the kinds of the blocks, as it is specified in Table 4.3.1. In case of $[x := y + n]^1$, three kinds of specifications are given according to the property of n. It is then the analyzer's responsibility to judge whether n is number 1 ,an odd number or an even number, depending on which clause is created.
For example,

- if $n = 1$, the clause will be

$$\forall s_1\ \forall s_2\ \forall s_3\ \forall r\ \mathrm{ABIN}(l, x, y, s_1, s_2, r)\wedge\mathrm{ADD}(+, s_2, s_3) \Rightarrow\mathrm{ABOUT}(l, x, y, s_3, s_2, +1)$$

- if n is an odd and positive number, the clause will be

$$\begin{aligned}&\forall s_1\ \forall s_2\ \forall s_3\ \forall r\ \mathrm{ABIN}(l, x, y, s_1, s_2, r)\wedge\mathrm{ADD}(+, s_2, s_3) \Rightarrow\\&(\mathrm{ABOUT}(l, x, y, s_3, s_2, +1)\wedge\mathrm{ABOUT}(l, x, y, s_3, s_2, > +1))\end{aligned}$$

- if n is an even and negative number, the clause will be

$$\forall s_1\ \forall s_2\ \forall s_3\ \forall r\ \text{ABIN}(l,x,y,s_1,s_2,r) \land \text{ADD}(-,s_2,s_3) \Rightarrow \\ (\text{ABOUT}(l,x,y,s_3,s_2,0) \land \text{ABOUT}(l,x,y,s_3,s_2,>+1))$$

Clauses for blocks other than $[x := y + n]^1$ are then quite straight forward and can be generated directly from the specifications.
Clause for relationship between blocks is specified as, (for flow(l_1, l_2))

$$\forall x\ \forall y\ \forall s_1\ \forall s_2\ \forall r\ \text{ABOUT}(l_1,x,y,s_1,s_2,r) \Rightarrow \text{ABIN}(l_2,x,y,s_1,s_2,r)$$

which tells that information about any two variables held at the exit of l_1 still holds at the entry of l_2.

Example 4.6 Let's return to the program piece in example 4.4. The clause generated by AB analyzer is

ADD(N,N,N)$\land$ADD(N,Z,N)$\land$ADD(N,P,N)$\land$ADD(N,P,Z)$\land$ADD(N,P,P)$\land$
ADD(Z,N,N)$\land$ADD(Z,Z,Z)$\land$ADD(Z,P,P)$\land$
ADD(P,N,N)$\land$ADD(P,N,Z)$\land$ADD(P,N,P)$\land$ADD(P,Z,P)$\land$ ADD(P,P,P)$\land$
VAR(x)$\land$VAR(y)$\land$VAR($A'1$)$\land$VAR($A'2$)$\land$
SIGN(N)$\land$SIGN(Z)$\land$SIGN(P)$\land$
RANGE(LEN1)$\land$RANGE(EN1)$\land$RANGE(EZ)$\land$RANGE(LEP1)$\land$RANGE(EP1)$\land$

ABIN(L1,x, y,TOP,TOP,TOP)$\land$ABIN(L1,$x, A'1$,TOP,TOP,TOP)$\land$
ABIN(L1,$x, A'2$,TOP,TOP,TOP)$\land$ABIN(L1,$y, A'1$,TOP,TOP,TOP)$\land$
ABIN(L1,$y, A'2$,TOP,TOP,TOP)$\land$ABIN(L1,$A'1, A'2$,TOP,TOP,TOP)$\land$

($\forall x\ \forall v1\ \forall v2\ \forall r1\ \forall r$ (ABIN(L1,$y, x, v1, v2, r1$)$\lor$ABIN(L1,$x, y, v2, v1, r1$))$\land$RANGE(r)$\Rightarrow$
ABOUT(L1,y, x,P, $v2, r$))$\land$
($\forall x\ \forall z\ \forall v1\ \forall v2\ \forall r$ ABIN(L1,$x, z, v1, v2, r$)$\land x \neq y \land z \neq y \Rightarrow$ABOUT(L1,$x, z, v1, v2, r$))$\land$

($\forall x\ \forall y\ \forall v1\ \forall v2\ \forall r$ ABIN(L2,$x, y, v1, v2, r$) $\Rightarrow$ABOUT(L2,$x, y, v1, v2, r$))$\land$

($\forall y\ \forall v1\ \forall v2\ \forall r1\ \forall r$ (ABIN(L3,$x, y, v1, v2, r1$)$\lor$ABIN(L3,$y, x, v2, v1, r1$))$\land$RANGE(r)$\Rightarrow$
ABOUT(L3,x, y,P, $v2, r$))$\land$
($\forall y\ \forall z\ \forall v1\ \forall v2\ \forall r$ ABIN(L3,$y, z, v1, v2, r$)$\land y \neq x \land z \neq x \Rightarrow$ABOUT(L3,$y, z, v1, v2, r$))$\land$

($\forall v1\ \forall v2\ \forall r$ (ABIN(L4,$x, y, v1, v2, r$)$\lor$ABIN(L4,$y, x, v2, v1, r$))$\Rightarrow$ ABOUT(L4,$x, y, v2, v2$,EZ))$\land$
($\forall z\ \forall v1\ \forall v2\ \forall r$ (ABIN(L4,$y, z, v1, v2, r$)$\lor$ABIN(L4,$z, y, v2, v1, r$))$\land z \neq x \Rightarrow$
ABOUT(L4,$x, z, v1, v2, r$))$\land$

$(\forall y\ \forall z\ \forall v1\ \forall v2\ \forall r\ \text{ABIN}(\text{L4},y,z,v1,v2,r) \wedge y \neq x \wedge z \neq x \Rightarrow \text{ABOUT}(\text{L4},y,z,v1,v2,r)) \wedge$

$(\forall x\ \forall y\ \forall v1\ \forall v2\ \forall r\ \text{ABIN}(\text{L5},x,y,v1,v2,r) \Rightarrow \text{ABOUT}(\text{L5},x,y,v1,v2,r)) \wedge$

$(\forall v1\ \forall v2\ \forall v3\ \forall r\ (\text{ABIN}(\text{L6},A'1,x,v1,v2,r) \vee \text{ABIN}(\text{L6},x,A'1,v2,v1,r)) \wedge \text{ADD}(v2,P,v3) \Rightarrow$
$(\text{ABOUT}(\text{L6},A'1,x,v3,v2,EZ) \wedge \text{ABOUT}(\text{L6},A'1,x,v3,v2,LP1))) \wedge$
$(\forall y\ \forall z\ \forall v1\ \forall v2\ \forall r\ \text{ABIN}(\text{L6},y,z,v1,v2,r) \wedge y \neq A'1 \wedge z \neq A'1 \Rightarrow \text{ABOUT}(\text{L6},y,z,v1,v2,r)) \wedge$

$(\forall x\ \forall v\ \forall v1\ \forall v2\ \forall r\ \forall r1\ \ (\text{ABIN}(\text{L7},A'1,x,v1,v2,r) \vee \text{ABIN}(\text{L7},x,A'1,v2,v1,r)) \wedge$
$\text{SIGN}(v) \wedge \text{RANGE}(r1) \Rightarrow \text{ABOUT}(\text{L7},A'1,x,v,v2,r1)) \wedge$
$(\forall x\ \forall v\ \forall v1\ \forall v2\ \forall r\ \forall r1\ \ (\text{ABIN}(\text{L7},A'2,x,v1,v2,r) \vee \text{ABIN}(\text{L7},x,A'2,v2,v1,r)) \wedge$
$\text{SIGN}(v) \wedge \text{RANGE}(r1) \Rightarrow \text{ABOUT}(\text{L7},A'2,x,v,v2,r1)) \wedge$
$(\forall y\ \forall z\ \forall v1\ \forall v2\ \forall r\ \text{ABIN}(\text{L7},y,z,v1,v2,r) \wedge y \neq A'1 \wedge y \neq A'2 \wedge z \neq A'1 \wedge z \neq A'2 \Rightarrow$
$\text{ABOUT}(\text{L7},y,z,v1,v2,r)) \wedge$

$(\forall x\ \forall y\ \forall v1\ \forall v2\ \forall r\ \text{ABOUT}(\text{L1},x,y,v1,v2,r) \Rightarrow \text{ABIN}(\text{L2},x,y,v1,v2,r)) \wedge$
$(\forall x\ \forall y\ \forall v1\ \forall v2\ \forall r\ \text{ABOUT}(\text{L2},x,y,v1,v2,r) \Rightarrow \text{ABIN}(\text{L3},x,y,v1,v2,r)) \wedge$
$(\forall x\ \forall y\ \forall v1\ \forall v2\ \forall r\ \text{ABOUT}(\text{L2},x,y,v1,v2,r) \Rightarrow \text{ABIN}(\text{L4},x,y,v1,v2,r)) \wedge$
$(\forall x\ \forall y\ \forall v1\ \forall v2\ \forall r\ \text{ABOUT}(\text{L3},x,y,v1,v2,r) \Rightarrow \text{ABIN}(\text{L5},x,y,v1,v2,r)) \wedge$
$(\forall x\ \forall y\ \forall v1\ \forall v2\ \forall r\ \text{ABOUT}(\text{L4},x,y,v1,v2,r) \Rightarrow \text{ABIN}(\text{L5},x,y,v1,v2,r)) \wedge$
$(\forall x\ \forall y\ \forall v1\ \forall v2\ \forall r\ \text{ABOUT}(\text{L6},x,y,v1,v2,r) \Rightarrow \text{ABIN}(\text{L5},x,y,v1,v2,r)) \wedge$
$(\forall x\ \forall y\ \forall v1\ \forall v2\ \forall r\ \text{ABOUT}(\text{L5},x,y,v1,v2,r) \Rightarrow \text{ABIN}(\text{L6},x,y,v1,v2,r)) \wedge$
$(\forall x\ \forall y\ \forall v1\ \forall v2\ \forall r\ \text{ABOUT}(\text{L5},x,y,v1,v2,r) \Rightarrow \text{ABIN}(\text{L7},x,y,v1,v2,r))$

Like in DS analysis, item N,Z and P represent symbol $-$,0 and $+$, respectively. Furthermore, LEN1 stands for less and equal than negative one(< -1), EN1 stands for equal to negative one(-1), ZE stands for equal to zero(0), EP1 stands for equal to positive one($+1$) and LEP1 stands for larger and equal than positive one($> +1$).
The solving result given by Succinct Solver is listed below. To facilitate reading, here we

only present the summarized information at the beginning point of each block.

$AB_\circ$	variable	signs and difference information
L2	(y, x)	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[1])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[2])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
L3	(y, x)	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[1])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[2])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
L4	(y, x)	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[1])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[2])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
L5	(x, y)	$(+, +, \{< -1, -1, 0, +1, > +1\})$
	$(x, A[1])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
		$(+, +, \{< -1, 0\})$
	$(x, A[2])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[1])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[2])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
L6	(x, y)	$(+, +, \{< -1, -1, 0, +1, > +1\})$
	$(x, A[1])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
		$(+, +, \{< -1, 0\})$
	$(x, A[2])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[1])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[2])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
L7	(x, y)	$(+, +, \{< -1, -1, 0, +1, > +1\})$
	$(x, A[1])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
		$(+, +, \{< -1, 0\})$
	$(x, A[2])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[1])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$
	$(y, A[2])$	$(+, \top, \{< -1, -1, 0, +1, > +1\})$

The tuple of the *signs and difference information* column for each variable pair, say (x, y) contains the signs of x and y, respectively as well as all the possible ranges of these two variables,i.e. $\{< -1, 0\}$ indicates both < -1 and 0 might be the magnitude difference between x and y.

4.4 Summary

In this chapter, we present flow logic specifications and implementations of three analyses, all the which are monotone analyses. These three analyses include detection of signs analysis, constant propagation analysis and array bounds analysis. The latter two are developed based on detection of signs analysis. But constant propagation is different from the other two in the sense that it is a *must* analysis. Thus its specifications for statements is also different, which is explained in the below part.

Strategies of writing flow logic specifications for monotone analyses are similar to bit-vector monotone in the sense that information are split into two categories, within a block and between blocks, and treated separately. These two categories correspond to the specifications for elementary blocks and statements. For elementary blocks, specifications state both the information newly generated by analyzing the corresponding block and the information transferred from the beginning to the ending point of the block. For statements, specifications state the information relationship between two blocks. What exactly the relationship is depends on the nature of the analysis, *must* or *may*. In case of *may* analysis, information at one block is a subset of the information at its successor. In case of *must* analysis, this relationship is not fixed and subjects to particular analysis, i.e. the $\sqcup$ operator in constant propagation analysis.

Chapter 5

Conclusion

Quite often, it is desirable to predict in advance the set of values resulting from programs and verify certain properties regarding their runtime behavior. For this purpose, the area of static program analysis offers compile-time computable techniques that can be used to safely approximate properties and values of programs without the need to execute them directly on computers. The functionality covered by static analyzers is wide and ranges from simple syntactic verifications that can be used in program transformations to complex runtime properties related to issues of security and optimization. In general, many approaches exist for building static analyzers, the main four of them are: Data Flow Analysis; Constraint Based Analysis; Abstract Interpretation; and Type and Effect System. Based on these approaches, a new term, Flow Logic, is proposed by Nielson and Nielson. It is based on constraint based analysis but also allows to integrate state-of-the-art insights from data flow analysis and abstract interpretation.

Flow logic focuses on specifying what it means for an analysis estimate to be acceptable for a program. One of the most important features of flow logic is that it separates the specification from the actual implementation of analysis. Thus the specification can be given differently depending on the interest in implementation details. There are two sets of criteria emerging for classifying flow logic specifications:

- *abstract* versus *compositional*, and
- *succinct* versus *verbose*

In this thesis, we present flow logic specifications for a variety of data flow analyses, all of which are *compositional*, meaning that the analysis is defined by inductive methods and *verbose*, meaning that it reports all the internal flow information. As a comparison, data flow specifications of four classical bit-vector analyses are presented as well.

For all the analyses concerned in this thesis, specifications of them are expressed and implemented in ALFP forms, which are then accepted by Succinct Solver as input clauses.

Succinct Solver is a generic tool for solving program analysis problems specified in Alternating Least Fixed Point Logic and it has been used by a lot of research projects as the back-end constraint solver. In this thesis, Succinct Solver is continually employed to compute the least solutions for all the implemented data flow and flow logic specifications. For the flow logic implementations of all classical bit vector analyses, it exhibits a lower asymptotic complexity. For monotone framework analyses, Succinct Solver shows its expressiveness and correctness when calculating the final result.

5.1 Summarized Insights

Although, bit-vector analyses are developed using two different approaches, there are some connections between the two kinds of specifications, in the sense that the flow logic specifications may be developed based on the data flow specifications. In the summary parts of chapter 3 and 4, we give some insights for writing flow logic specifications for both bit-vector analyses and monotone analyses. These insights could be summarized to

- for each block, specify
  1. the information newly generated by analyzing the block
  2. the information not being affected by the block, meaning that if it holds before analyzing the block, then it still holds when leaving the block
- if there is a flow (or reverse flow) relationship existing between a pair of blocks, specify what's the relationship between information held by these two blocks. Normally, it could be a subset or superset relation depending on whether a *may* or *must* analysis is performed.

  for *may* analysis , the operator is $\sqcup$ meaning that information at the *beginning* point of one block, say l, is always a superset of the information coming from the *ending* points of each predecessor of l[1]. Which point of a block is a beginning point or ending point is determined by the direction of the analysis (i.e. the direction of the flows).

  In case of *forward* analyses , information flows from the exit of a block to the entry of another block, thus the beginning point of block l is the entry of the block. For each flow(l_1, l_2),

$$Analysis_{\bullet}(l_1) \subseteq Analysis_{\circ}(l_2)$$

[1]Here predecessor or successor is defined according to the direction of flow

where *Analysis* is the name of the performed analysis

In case of *backward* analyses , information flows from the entry of a block to the exit of another block, thus the beginning point of block l is the exit of the block. For each reverse flow(l_1, l_2)

$$\mathit{Analysis}_\circ(l_2) \subseteq \mathit{Analysis}_\bullet(l_1)$$

for *must* analysis , the operator is $\sqcap$ meaning that information at the *beginning* point of a block is a subset of the information coming from its predecessor. In order to deal with $\sqcap$, we introduce a flag, *Valid* or *Invalid*, for each information at certain block to indicate whether it still holds or not. After incorporating the flag, operator $\sqcap$ is represented by the symbol $\Subset$.

In case of *forward* analyses , information flows from the exit of a block to the entry of another block. Thus the relationship between blocks is, if flow(l_1, l_2) exists,

$$\mathit{Analysis}_\circ(l_2) \Subset \mathit{Analysis}_\bullet(l_1)$$

In case of *backward* analyses , information flows from the entry of a block, say l_1, to the exit of another block, say l_2. Thus

$$\mathit{Analysis}_\bullet(l_1) \Subset \mathit{Analysis}_\circ(l_2)$$

5.2 Future Work

The restriction on time left us with no choice but to cut down on some areas, which include

5.2.1 Combine analyses to get a more precise analysis result

In the WHILE language developed for the analyses, we extend variables to include arrays. This makes it more closely to the programming languages in the real world and paves the way to analyzing those languages. But when come to certain array element with an expression index, say $A[e]$, we always estimate it to all the possible elements in array A. For example, in reaching definition analysis, if the assignee of an assignment statement is $A[e]$, all the elements of array A are thought to be defined at this block. It is a safe yet a less precise approximation. If the value of e can be estimated to a more narrow range than the bounds of array A, then its not necessary to regard all the element of A as defined at the current block. This requires an approximation on the value of e, which can be done by applying another analysis. In this section, we use some examples to illustrate that the combination of analyses may yield a more precise result than single analysis.

Detection of Signs analysis + Reaching Definition Analysis

The result given by DS analysis reveals the possible signs of variables at each program point, which can be used to estimate the value of an expression. This information is quite useful for RD analysis.

Example 5.1 Consider the statement

$$[y := -1]^1;\ [x := 0]^2;\ \text{if } [x > 0]^3 \text{ Then } [A[x] := x + A[1]]^4 \text{ Else } [A[y] := A[x]]^5$$

where array A is declared as *Array A of* $[-2..1]$. The defined place of each element of A given by RD analysis could be summarized to the below table

$\text{RD}_\bullet$	defined place
L4	$(A[-2], lab0), (A[-2], \text{L4})$ $(A[-1], lab0), (A[-1], \text{L4})$ $(A[0], lab0), (A[0], \text{L4})$ $(A[1], lab0), (A[1], \text{L4})$
L5	$(A[-2], lab0), (A[-2], \text{L5})$ $(A[-1], lab0), (A[-1], \text{L5})$ $(A[0], lab0), (A[0], \text{L5})$ $(A[1], lab0), (A[1], \text{L5})$

At the exit of block 4 (block 5), all the elements are defined at either block $lab0$ or block 4 (block 5) because the values of x and y, each of which works as an index of A, are unknown. But Detection of Signs analysis estimates that at block 4, x has the sign 0 (e.g. $x = 0$) and at block 5, y is a negative number. By utilizing this information, now RD analysis could say that, at block 4, $A[0]$ is defined at block 4 while other elements of A are defined at block $lab0$ and at block 5, since y is negative, all the elements of A with negative indexes are defined at either block $lab0$ or block 4; other elements then can only be defined at $lab0$. Note that actually RD analysis can not use the information gained from DS analysis directly, as long as its current specifications are not changed. One possibility is to have

Succinct Solver to do this job. Now, the RD analysis is able to give the result,

$RD_\bullet$	defined place
L4	$(A[-2], lab0)$ $(A[-1], lab0)$ $(A[0],$L4$)$ $(A[1], lab0)$
L5	$(A[-2], lab0), (A[-2],$L5$)$ $(A[-1], lab0), (A[-1],$L5$)$ $(A[0], lab0)$ $(A[1], lab0)$

By comparing the above two tables, clearly we get a more precise result by combining these two analyses.

Constant Propagation analysis + Reaching Definition Analysis

Constant Propagation analysis approximates the value of each variable to either a non-constant or a certain constant. Thus, for example, if x is estimated to constant 6, then approximating the value of $A[x]$ is equal to approximate the value of $A[6]$.

Example 5.2 Returning to the previous example statement, the result of CP analysis will show that at block 4, x has constant value 0 and at block 5, y has value -1, Thus these two blocks can be rewritten to $[A[0] := 0 + A[1]]^4$ and $[A[-1] := A[0]]^5$, respectively. This step could be done by, say, constant folding optimization. Applying RD analysis to the new statement can yield the following result

$RD_\bullet$	defined place
L4	$(A[-2], lab0)$ $(A[-1], lab0)$ $(A[0],$L4$)$ $(A[1], lab0)$
L5	$(A[-2], lab0)$ $(A[-1],$L5$)$ $(A[0], lab0)$ $(A[1], lab0)$

Again, the preciseness of the analysis result is improved.

Array Bounds Analysis + Reaching Definition Analysis

Array Bounds Analysis is used to estimate the signs and magnitude difference between a pair of variables, which normally appear as a variable and an array bound. This information can also used to improve the preciseness of RD analysis. For example, if the bounds of array A are declared to be the values of y and z, which is immediately followed by the statement $[A[x] := 6]^l$, approximating the signs and magnitude difference between x and y, x and z may help to determine which elements of A are defined at block l. This requires that the bounds of arrays have to be declared dynamically according to the values of variables. But in our WHILE language array bounds must be declared statically, meaning that they must be certain constants. Thus for the WHILE language, we can not use the combination of AB and RD analyses, which may be possible for other programming languages. We won't go deeper into it here.

5.2.2 Use other methods to implement flow logic specifications

Besides Succinct Solver, there are some other methods for computing the least solutions to program analysis problems considered in data flow analysis. One of them is the well-known worklist algorithm. It might be of interest to compare Succinct Solver to worklist algorithm with respect to their efficiencies for solving constraints as well as data flow equations. But to do that, it is required to make some extensions to worklist algorithm in order for it to accept ALFP clauses.

5.2.3 Extend Succinct Solver to have more features

Another area of relevant is most likely to be improving Succinct Solver by incorporating some extra features. Over the recent years, a lot of effort has been put into increasing the integrity and efficiency of Succinct Solver. But during the whole period of developing analyzers, we find that occasionally some more powerful features are required in order to solve the problems at hand.

One example would be adding simple arithmetic and boolean calculation abilities and thus extend pre-condition in ALFP to include

$$pre \quad ::= \quad t_1 \; op \; t_1$$

where t_1 and t_2 are terms and *op* is either an arithmetic or boolean operator.

If this calculation ability is available, constant propagation, for instance, could be benefited from it in the sense that, as mentioned in chapter 4.2, the approximate will be more precisely when evaluating statement of the form $[x := e_1\ op\ e_2]^l$.

We believe that by incorporating these extra features in the Succinct Solver, its usage as a generic tool could be enhanced for other research projects.

Appendix A

Test Setup

A.1 Test Suites

A.1.1 Program a-*m*

The basic structure of scalable program a-m is,

```
$i := 1$;
while $i \leq 2$ do
    ($result[i] := result[i] + A[i]$;
    $i := i + 1$)
```

The function of program a-m is to get the sum of m matrixes, stored in array named *result*, meaning that it has m above program pieces.

A.1.2 Program b-*m*

Scalable program b-m is used to calculate the sum of all the positive values held by elements of array A

```
while $i \leq m$ do
    (If A[i] $> 0$ Then $result := result + A[i]$
    Else $result := result$)
```

The size of array A corresponds to parameter m.

A.2 Timing the Experiments

The working procedure of Succinct Solver is split into two distinct steps. In the initialization step the clauses are loaded into memory and all internal structure are initialized. In the solving step the actual analysis result is then computed. These two steps are timed separately and only the time for second step are measured. All times are collected without the time specifically used for garbage collection, which are less informative as they show great variation when the same experiment is repeated. Times measured in all the experiments are in the term of CPU-time, which is recorded by the SML interpreter.

A.3 Presenting the Results

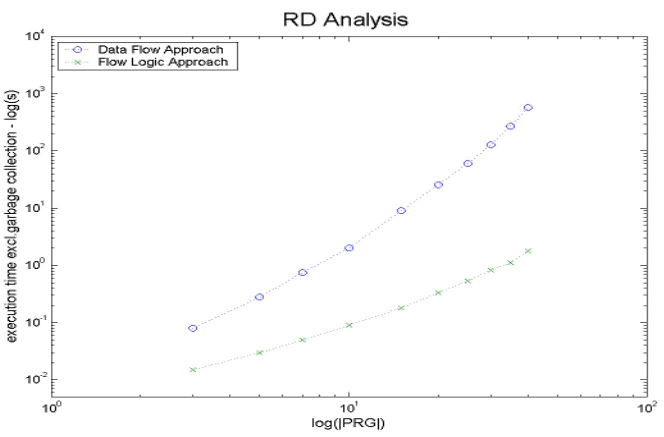
Taking the approaches of [5], we assume the execution time t for solving step can be expressed as a function of program size m,

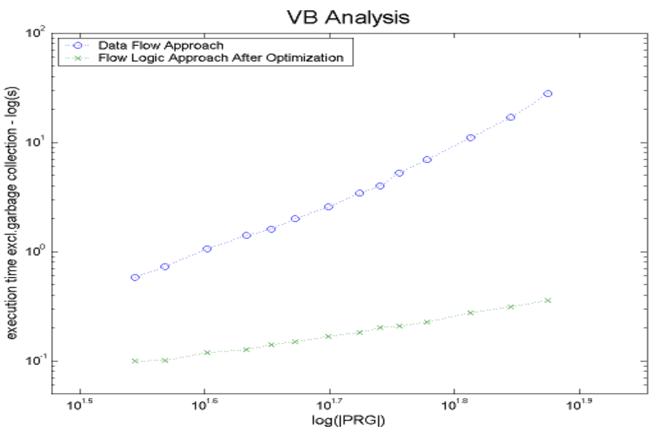
$$t = c \cdot m^r + c_0$$

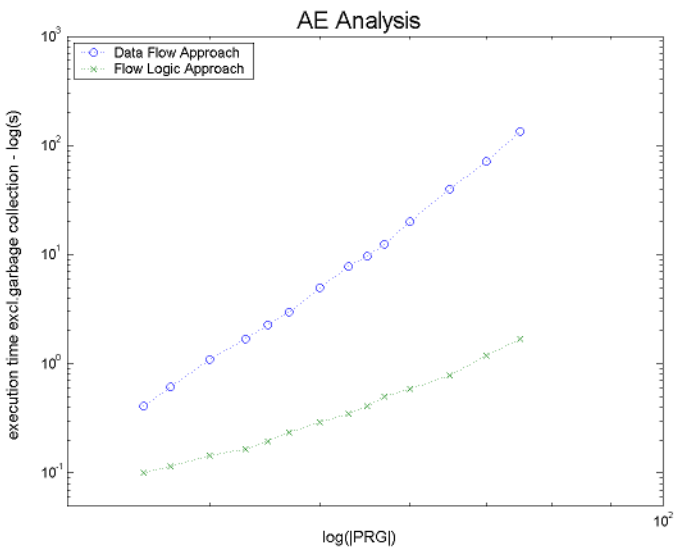
for some number r $\in \mathbb{R}$. Due to separation of initialization and solving we assume c_0 to be 0, since when m=0 no time is used. In subsequent chapters, we shall plot corresponding values of t and m on a double logarithmic scale, meaning that the above formula is rewritten to $\log t = \log c + r \cdot \log m$. In this way, r can be recognized as the degree of curves, which facilitate us to estimate c and r.

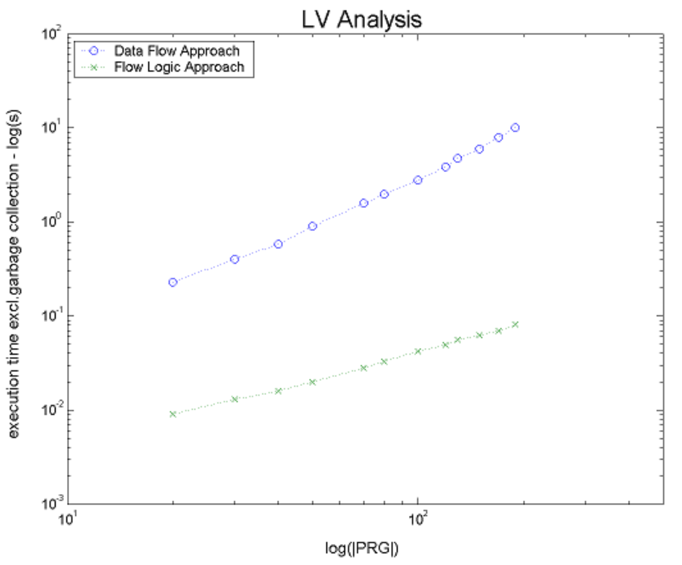
Appendix B

Experiments on Program b-m






AE Analysis
Data Flow Approach
Flow Logic Approach
execution time excl.garbage collection - log(s)
log(|PRG|)
10^3
10^2
10^1
10^0
10^-1
10^2


LV Analysis
Data Flow Approach
Flow Logic Approach
execution time excl.garbage collection - log(s)
log(|PRG|)
10^2
10^1
10^0
10^-1
10^-2
10^-3
10^1
10^2

Appendix C

User's Guide

C.1 Installation Guide

The analyzers require the following softwares to be installed:

- Moscow ML version 2.01 available at
  http://www.dina.dk/~sestoft/mosml.html
- New Jersey SML version 110.0.7 available at
  http:://www.smlnj.org
- The Succinct Solver version 2.0 available at
  http://www.imm.dtu.dk/cs_SuccinctSolver/

Follow the installation procedures at the respective websites. Next, the program analyzers should be installed:

- Download the file
  http://www.student.dtu.dk/~s020937
- Unpack the *analyzers.rar*, which creates a subdirectory called *analyzers*
- Edit the file *analyzers/run.sml* and set the values of parameters
  1. set *pro_name* to the path and name of the file containing the program to be analyzed.
  2. set *analyzer* to the name of analyzer to be performed (please see C.2 for the name of each analyzer)
  3. set *dest_dir* to the directory *Application/Examples/* in your installation of the Succinct Solver (typically the directory *HORN/Application/Examples* in your Succinct Solver installation)

Now, the analyzers are ready to run. For example, to run the file *run.sml* (1) change to the directory *analyzer*, (2) start Moscow ML (e.g. by typing *mosml* in a shell), and (3) execute the command `use "run.sml"`

The generated ALFP clause for the program is then contained in the file

Application/Examples/result.cl

Succinct Solver is needed to be run to compute the result for the clause. This can be done by (1) change to the directory *Application*, (2) start New Jersey SML (e.g. by typing *sml* in a shell), (3) compile Succinct Solver by executing the command `use "run"`, and (4) execute the command `ex("result")`. The final result for solving the clause can be found in the file *Application/Examples/result.cl.al*

C.2 Names of Analyzers

- Reaching Definition data flow analyzer – *rd.sml*
- Reaching Definition flow logic analyzer – *rdimp.sml*
- Very Busy Expression data flow analyzer – *vb.sml*
- Very Busy Expression flow logic analyzer – *vbimp.sml*
- Available Expression data flow analyzer – *ae.sml*
- Available Expression flow logic analyzer – *aeimp.sml*
- Live Variable data flow analyzer – *lv.sml*
- Live Variable flow logic analyzer – *lvimp.sml*
- Detection of Signs analyzer – *ds.sml*
- Constant Propagation analyzer – *cp.sml*
- Array Bounds analyzer – *ab.sml*

Bibliography

[1] H.Riis.Nielson and F.Nielson. Flow Logic: A Multi-paradigmatic Approach to Static Analysis. The Essence of Computation - Complexity, Analysis, Transformation, volume 2566 of Lecture Notes in Computer Science, pages 223-244. Stringer Verlag, 2002.

[2] F.Nielson, H.Riis.Nielson, and C.Hankin. Principles of Program Analysis. Springer, 1999

[3] H.Sun. User's Guide for the Succinct Solver (V2.0). Draft, 2003

[4] F.Nielson, H Seidl, and H.Riis.Nielson. A Succinct Solver for ALFP. Nordic Journal of Computing. 2003.

[5] M.Buchholtz, H.Riis Nielson, and F. Nielson. Experiments with Succinct Solvers. Technical Report IMM-TR-2004-4, Technical University of Denmark, 2002

[6] H.Pilegaard. A Feasibility Study-The Succinct Solver v2.0, XSB Prolog v2.6, and Flow-Logic Based Program Analysis for Carmel. Technical University of Denmark, 2003

[7] Shark, M., "Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers", Computer Languages, 5, Peramon Press Ltd., 1980.

[8] Baudouin Le Charlier and Pascal Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.

[9] D.McAllester. On the complexity analysis of static analyses. In Static Analysis Symposium, volumn 1694 of Lecture Notes in Computer Science, page 312-329,1999.

[10] F.Nielson, H.Riis Nielson, H.Yan Sun, M.Buchholtz, R.R.Hansen,H.Pilegaard and H.Seidl. The Succinct Solver Suite.

[11] F.Nielson, H.Riis and H.Seidl. Automatic Complexity Analysis. In European Symposium on Programming (ESOP), volume 2305 of Lecture Notes in Computer Science, pages 243-261. Springer Verlag,2002.

[12] K.Sagonas, T.Swift, D.S.Warren, J.Freire, P.Rao, B.Cui and E.Johnson. The XSB System. Web page: http://xsb.sourceforge.net/,2003.

[13] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, H. Riis Nielson, Automatic Valida-

tion of Protocol Narration, Proceedings of the 16th Computer Security Foundations Workshop (CSFW 03)., pp. 126-140, IEEE Computer Society Press, 2003.

[14] R.R.Hansen, A Prototype Tool for JavaCard Firewall Analysis, Nordic Workshop on Secure IT-Systems, 2002.

[15] Riis Nielson, H., Nielson, F., Buchholtz, M., Security for Mobility, Foundations of Security Analysis and Design II - FOSAD 2001/2002 Tutorial Lectures, vol. 2946, pp. 207-265, 2004.

[16] H.Riis Nielson and F.Nielon. Semantics with Applications: A Formal Introduction. Wiley, 1992. (See http://www.daimi.au.dk/ hrn for an on-line version)