

# Field Sensitive Program Dependences for Large Scale Systems

Shay Litvak<sup>1</sup>

School of Computer Science, Tel-Aviv University, Israel

M.Sc. Thesis

under the supervision of Prof. Mooly Sagiv  
and the consultation of Dr. Nurit Dor and Dr. Noam Rinetzky

July, 2009

<sup>1</sup>shay.litvak@cs.tau.ac.il

# Acknowledgements

I would like to express my sincere gratitude to all those who contributed to the completion of my thesis.

First, I would like to express my deep gratitude, respect, and thanks to Prof. Mooly Sagiv, my supervisor. This thesis would have been impossible to complete without his guidance, patience and support.

I thank Dr. Nurit Dor for her constant encouragement, guidance and support. Nurit's extensive knowledge of the issues researched in this thesis, as well as her willingness to help and to share her knowledge, were a great help in the completion of this thesis.

I am also grateful for the help of Dr. Noam Rinetzky who has dedicated considerable time in reviewing and providing valuable feedback throughout this thesis.

Finally, I would like to thank my beloved wife Nitsan for her encouragement and support.

# Abstract

Computing transitive program dependences is a fundamental operation in many software engineering tools. Computing these dependences efficiently and precisely for aggregate structures is challenging, and the limitations of existing tools affects analysis of practical programs, e.g., large-scale ERP systems.

This thesis presents a **Field-Sensitive Dependence** (*FSD*) analysis. FSD determines dependences between *fields* of aggregate structures. FSD improves efficiency without compromising the precision by maintaining an interval based field attributes to refine the transitive closure of the program (immediate) dependency graph.

The currently state-of-the-art precise algorithm is not scalable. A commonly used scalable algorithm improves the running time by 54% and the memory consumption by 35% but includes a high precision penalty — a false-positive rate of 62% . FSD, which is both precise and scalable, is used to analyze ERP programs of over a million LOCs. Our extensive study of large, real-life programs (100,000s of LOCs) shows that our algorithm improves the running time by 43% and the memory consumption by 31% when compared to the current precise solution known today with the same level of precision. In addition, FSD successfully analyzes large programs which the current precise algorithm has failed to analyze due to exceeded memory consumption.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	The Problem . . . . .	6
1.2	Current Approaches . . . . .	7
1.3	Our Goal . . . . .	7
1.4	Overview of our Approach . . . . .	8
1.4.1	A Vanilla Static Analysis for Field Sensitive Dependences . . . . .	8
1.4.2	An Interval Based Approach for Efficiently Computing Field Sensitive Dependences. . . . .	10
1.4.3	Summary . . . . .	10
1.5	Main Contributions . . . . .	11
1.6	Outline . . . . .	11
<b>2</b>	<b>Preliminary</b>	<b>12</b>
2.1	Program Model . . . . .	12
2.1.1	Syntactic Domains . . . . .	12
2.1.2	Aggregate Structures . . . . .	12
2.1.3	Primitive Statements . . . . .	14
2.1.4	Procedures . . . . .	14
2.1.5	Simplifying Assumptions . . . . .	14
2.2	Program Dependence Graph (PDG) . . . . .	14
2.2.1	Def-Use Chains . . . . .	14
2.2.2	PDG . . . . .	15
<b>3</b>	<b>Field Sensitive Transitive Dependences</b>	<b>16</b>
3.1	Concrete Instrumented Semantics . . . . .	17
3.1.1	Identity Statements . . . . .	18
3.1.2	Write Statements . . . . .	18
3.1.3	Read Statements . . . . .	19
3.1.4	Computational assignment . . . . .	20
3.1.5	Inferring Dependences . . . . .	20
3.2	Vanilla Static Analysis . . . . .	21
3.2.1	Identity Statements . . . . .	21

3.2.2	Write Statements . . . . .	22
3.2.3	Read Statements . . . . .	22
3.2.4	Computational Statements . . . . .	23
3.2.5	Computing the Dependences . . . . .	23
3.2.6	Complexity . . . . .	24
<b>4</b>	<b>Atomization Based Field Sensitive Dependences</b>	<b>27</b>
4.1	Atomization Based Algorithm . . . . .	28
4.1.1	Identity Statements . . . . .	28
4.1.2	Computational Statements . . . . .	28
4.1.3	Computing the Dependences . . . . .	29
4.1.4	Complexity . . . . .	32
<b>5</b>	<b>Interval Based Field Sensitive Dependences</b>	<b>33</b>
5.1	Transfer Function . . . . .	33
5.1.1	Identity Statements . . . . .	33
5.1.2	Write Statements . . . . .	34
5.1.3	Read Statements . . . . .	35
5.1.4	Computational Statements . . . . .	36
5.2	Computing The Dependences . . . . .	36
5.2.1	Basic Interval Operations . . . . .	37
5.2.2	Complexity . . . . .	41
<b>6</b>	<b>Analyzing Large Scale Programs</b>	<b>42</b>
6.1	Field Sensitive Dependence Analysis on the PDG . . . . .	43
6.1.1	Constructing Interval Label PDG . . . . .	43
6.1.2	The Abstract State . . . . .	43
6.1.3	The Transfer Function . . . . .	44
6.1.4	Exploring the Transfer Function . . . . .	45
6.2	Weakly Typed Languages . . . . .	46
6.3	Inter-procedural Analysis . . . . .	49
<b>7</b>	<b>Empirical Results</b>	<b>50</b>
7.1	The PanayaIA Tool . . . . .	50
7.2	The Benchmark . . . . .	51
7.3	Targeting More Precise Atomization . . . . .	52
<b>8</b>	<b>Related Work</b>	<b>55</b>
<b>9</b>	<b>Further Work and Conclusion</b>	<b>56</b>
	<b>References</b>	<b>58</b>

**List of Figures**

**60**

**List of Tables**

**61**

# Chapter 1

## Introduction

Research in improving efficiency and precision of computing program dependences (in particular in the context of static program slicing [Wei84]) has primarily focused on context sensitivity (e.g., [RHS95, SRH96]) and object-field sensitivity (e.g., [SFB07]). We have found that imprecise handling of fields in *aggregate structures* (e.g., C structs) can also significantly decrease the precision of the analysis. While field-sensitive transitive dependences are easy to compute in programs with small aggregates, when many fields are present, scalability is adversely affected (see Chapter 7).

One class of programs with large aggregate structures can be found in ERP systems. Dominant in the world of business applications, these systems are challenging to analyze. An ERP system is typically comprised of hundreds or thousands of programs. Each program represents a business process and may contain over a million LOCs. ERP programs make heavy use of aggregate structures to control many aspects of their execution (see Section 7.1). It is not unusual that a single structure contains hundreds of fields and a straightforward dependence analysis may require more than 15GB of memory.

In this thesis, we present an algorithm for computing field-sensitive (transitive) program dependences. We evaluate the algorithm using impact analysis for ERP systems, but the algorithm has applications in other software-engineering tasks, such as program maintenance [Gal90, GL91], debugging [LW86], testing [Bin92, BH93], semantic differencing [Hor90], slicing [HRB90], reuse [NEK94], and merging [HPR89a].

### 1.1 The Problem

We say that a variable  $v$  used in statement  $st$  has a *program dependence* on the value of variable  $seed$  assigned in statement  $st_{seed}$  if the value of  $seed$  in  $st_{seed}$  may flow across one or more assignments to a memory location which is either (i) used in  $st$  or (ii) used in the evaluation of a condition which controls whether  $st$  is executed or not. *Program dependences* are typically computed as a transitive closure of *immediate data* and *control* dependences among statements [OO84].<sup>1</sup> Let us illustrate why the transitive closure of immediate dependences computes an imprecise program dependence in the presence of aggregate structures and Big L-Value

---

<sup>1</sup>Informally, statement  $st_2$  has an immediate *data dependence* on statement  $st_1$  if  $st_1$  defines an l-value that is used in  $st_2$ . Statement  $st_2$  has an immediate *control dependence* on statement  $st_1$  if  $st_1$  evaluates a condition which controls whether statement  $st_2$  is executed. For a more formal definition, see Section 2.2.

constructs (an assignment is said to be a Big L-value operation if the L-value assigned to is a compound memory location such as array or structure).

Consider the example in Fig. 1.1(a), on which we wish to compute program dependences on the definition of `seed` in statement  $l_1$ . Fig. 1.1(b) shows the *Program Dependence Graph (PDG)* of the immediate dependences in the program:  $(l_1, l_2)$ ,  $(l_2, l_3)$ ,  $(l_3, l_4)$ , and  $(l_3, l_5)$ . All of these dependences are precise, yet their transitive closure yields a spurious dependence of `t.g` in  $l_4$  on the definition of `seed` in  $l_1$ . We wish to compute program dependences that are field sensitive, i.e., they avoid such spurious dependences.

## 1.2 Current Approaches

Currently, there are two basic approaches for computing program dependences in programs manipulating aggregate structures. The approaches represent different tradeoffs between efficiency and precision. The first one is efficient but not field sensitive. The second one has opposite properties.

**Whole Structure Approach (WS)** Handles a definition point or a use point of a structure field as if it may define or use the whole structure, respectively [OO84, Ly184, Muc97a].

**Atomization Approach (ATOM)** Performs a pre-step of *atomization* that disassembles the structure to its primitive components [RFT99, Muc97b].

Both approaches have drawbacks: WS can yield superfluous dependences when manipulating structure fields, while ATOM may be too expensive on programs with large structures (due to the large increase in the total number of variables and statements). Indeed, while the whole structure approach detects dependences only at the structure level, the atomization approach infers *field-sensitive* dependences, i.e., which fields of the dependent variable depends on which field of the seed.

Computing program dependences using the WS approach may lead to spurious dependences. Consider the example shown in Fig. 1.1(a) on which we wish to compute dependences on `seed`. WS handles accesses to structure variables at the granularity of whole structures. Thus, essentially, WS computes the dependences by transitive closure on the PDG shown in Fig. 1.1(b) and imprecisely infers that `t.g` in line  $l_4$  may depend on `seed` in line  $l_1$ .

The atomization approach, on the other hand, can detect that `t.g` does not depend on `seed`. However, essentially, it requires analyzing the program shown in Fig. 1.1(c), by transitive closure on the PDG shown in Fig. 1.1(d). Note that every structure variable has been disassembled into two `int` variables and that statement  $l_3$  in Fig. 1.1(a) has been replaced by two statements,  $l_{3,1}$  and  $l_{3,2}$ , in Fig. 1.1(c). Finally, recall that in real life programs, structures may contain hundreds of fields which make this approach infeasible in practice.

## 1.3 Our Goal

This thesis defines an algorithm for computing (transitive) field-sensitive dependences between aggregate fields in a direct way (i.e., without the need to convert the program into atomized form). Our algorithm also reports which fields *transmit* the dependence. Specifically, it computes for

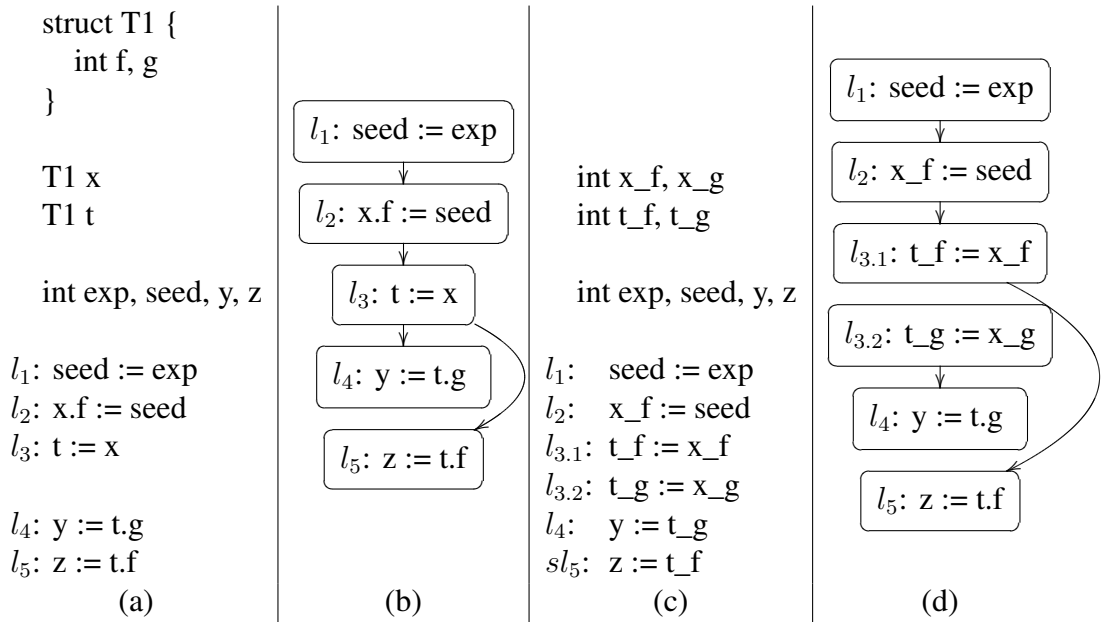


Figure 1.1: (a) A program using user-defined type `T1` and (b) its PDG. (c) The program after atomization. (d) The atomized PDG.

each program dependence which fields of the seed affect which fields of the target variable. For example, consider the program shown in Fig. 1.2(a). Our algorithm computes for the statement in  $l_7$  that  $b.g$  depends on  $seed.f$  and that  $b.f$  depends on  $seed.g$ .

Tracking field-sensitive dependences is non-trivial because field level operations are interleaved with whole-structure level operations (Big L-Value constructs). Moreover, the presence of control statement drastically complicates the problem. However, tracking field sensitive is critical for precision: Our experimental results, reported in Chapter 7, show that a transitive closure of the program (direct) dependency graph (i.e., the WS approach) computes, on average, 62% more spurious dependences than the precise field sensitive dependence analysis (the ATOM approach).

## 1.4 Overview of our Approach

We now present the key ideas behind our algorithm by applying it in a simplified setting of programs containing only memory-copy assignments. In addition, we limit the algorithm to compute the dependences on a fixed seed statement  $l_1$ , the only statement that defines the variable *seed*. We further ease the presentation of our algorithm, by presenting it in two phases: We first present a vanilla static analysis that computes field sensitive dependences. Then, we present an equally-precise but more efficient interval-based algorithm.

### 1.4.1 A Vanilla Static Analysis for Field Sensitive Dependences

Field sensitive dependences can be computed using a static analysis which iteratively traverses the program's *control flow graph* (CFG) and computes a set of *value dependences* for every CFG

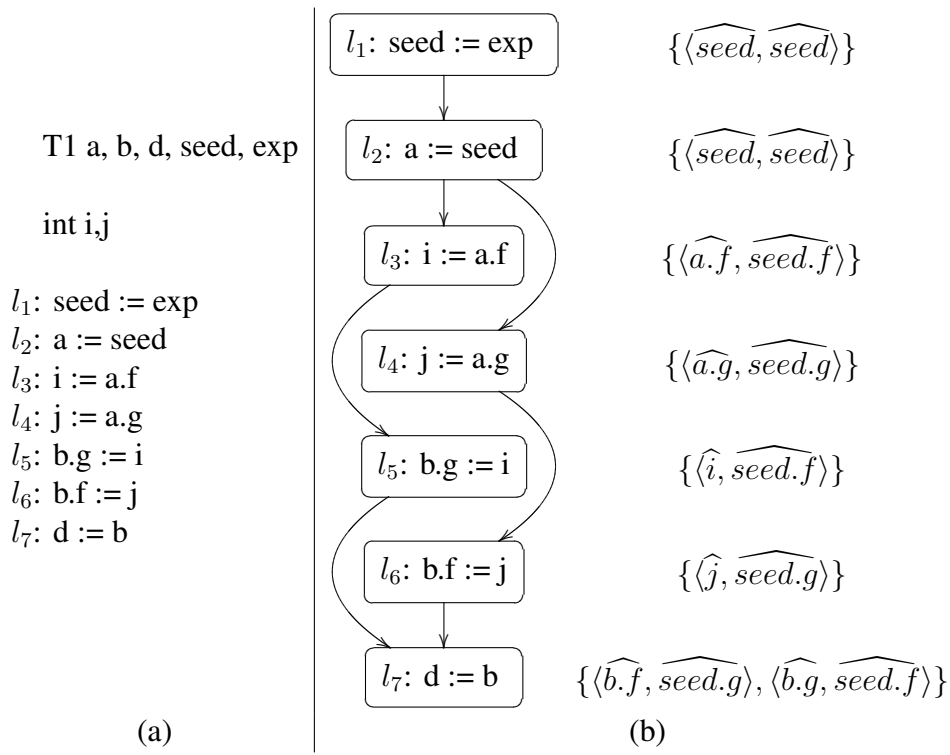


Figure 1.2: (a) a program manipulating user-defined type  $T1$ , defined in Fig. 1.1, and (b) its PDG annotated with the abstract states of the interval based algorithm at every statement.

node. The analysis computes at every statement  $st$  a set of program dependences. A dependence at statement  $st$   $\langle x.\alpha, seed.\beta \rangle$  represents that  $x.\alpha$  in  $st$  (may) depend on the definition of  $seed.\beta$ .

Consider for example Fig. 1.3, which shows the result of the vanilla static analysis which computes dependences for the program shown in Fig. 1.1(a). The analysis can infer a dependence from  $l_1$  to  $l_5$  (due to the use of  $t.f$  which is dependent on  $seed$ ) while avoiding the spurious dependence from  $l_1$  to  $l_4$  (since the use of  $t.g$  does not depend on  $seed$ ).

Note, however, that the analysis records at each statement a set of all transitive dependences between the seed def variable and every use variable along the dependency path. In the following section, we show how we can record at every statement only the set of dependences pertaining to a single variable - the statement use variable.

$$\begin{array}{l|l}
 l_1: seed := exp & \sigma_1 = \emptyset \\
 l_2: x.f := seed & \sigma_2 = \{\langle seed, seed \rangle\} \\
 l_3: t := x & \sigma_3 = \sigma_2 \cup \{\langle x.f, seed \rangle\} \\
 l_4: y := t.g & \sigma_4 = \sigma_3 \cup \{\langle t.f, seed \rangle\} \\
 l_5: z := t.f & \sigma_5 = \sigma_4 \\
 & \sigma_6 = \sigma_5 \cup \{\langle z, seed \rangle\}
 \end{array}$$

Figure 1.3: Dependences computed by vanilla static analysis over the CFG,  $\sigma_1$  is the initial dependences set and  $\sigma_i$  is the set of dependences computed before processing statement  $l_i$ .

## 1.4.2 An Interval Based Approach for Efficiently Computing Field Sensitive Dependences.

We compute field sensitive dependences more efficiently by performing static analysis over the graph of (direct) program dependences. Let  $use(st)$  be the variable used (or which one of its fields is being used) at  $st$ . The algorithm computes at  $st$  the same dependences for  $use(st)$  as the vanilla algorithm, but avoids computing dependences for other variables. In addition, the algorithm can, in many cases, compute one dependency that represents a set of field dependences. An interval, which is a pair of integers  $\langle i, j \rangle$ , of variable  $v$  represents the memory area of  $v$  that starts with the  $i$ -th byte and ends at the  $j$ -th byte. A single interval  $\langle i, j \rangle$  representation for an allocated aggregate can be used to represent all fields that are allocated within this interval. The key elements that contribute to the algorithm's efficiency are:

**Observation I** Maintaining dependences of  $use(st)$  at  $st$  suffices to compute transitive dependences by following the edges of *PDG*. (Recall that the *PDG* is the graph of *immediate* program dependences).

**Observation II** Because a single interval can be used to represent a set of adjacent fields, a single interval-based dependence can be used to represent a set of adjacent field dependences.

Our analysis computes for every statement a set of tuples of the form  $\langle \widehat{t.\alpha}, \widehat{s.\beta} \rangle$ , where  $\widehat{t.\alpha}$  and  $\widehat{s.\beta}$  are the memory intervals that represent the access paths  $t.\alpha$  and  $s.\beta$  respectively. Every tuple  $\langle \widehat{t.\alpha}, \widehat{seed.\beta} \rangle$  at statement  $st$  represents a may dependence from the interval  $\widehat{t.\alpha}$  of  $use(st)$  on the interval  $\widehat{seed.\beta}$  of the def of the seed statement. For example, the analysis of the program shown in Fig. 1.2 computes at statement  $l_7$  the set of tuples  $\langle \widehat{b.f}, \widehat{seed.g} \rangle$  and  $\langle \widehat{b.g}, \widehat{seed.f} \rangle$ , which means that  $b.f$  and  $b.g$  may depend on  $seed.g$  and  $seed.f$ , respectively, at  $l_7$ . Fig. 1.4 presents an additional example. It shows the result of the analysis of the program shown in Fig. 1.1(a). The analysis associates the empty state to  $l_4$ . Thus, it infers that the value of  $t.g$  at  $l_4$  does not depend on the seed.

The analysis computes the dependences by first constructing a labeled *PDG*. An edge between  $st1$  and  $st2$  labeled with an interval  $u$ , represents an immediate dependency between the actual use interval  $u$  of  $use(st2)$  and  $def(st1)$ . When traversing an edge from  $st$  to  $st'$  in the *PDG*, the analysis defines how to transform interval dependences on  $use(st)$  into interval dependences on  $use(st')$  according to the state in  $st$  and statement  $st$ .

The added efficiency compared to the vanilla analysis comes from two facts: (i) the analysis does not maintain irrelevant dependences: the dependences of  $use(st')$  can be computed by a (distributive) function on the set of dependences of variables  $use(st)$  such that  $st'$  immediately depends on  $st$ ; (ii) multiple adjacent field dependences are represented by a single interval dependence.

## 1.4.3 Summary

FSD obtains a precision as the full atomization, the most precise solution, with running time as good as the whole structure approach, the fastest one. Moreover, we display a real life case study, for which due to the large structures in use (hundreds of fields) in large programs (hundreds of thousands of lines of code) it is impractical to perform the full structures atomization. Thus making our approach superior to the conventional way for large scale systems.

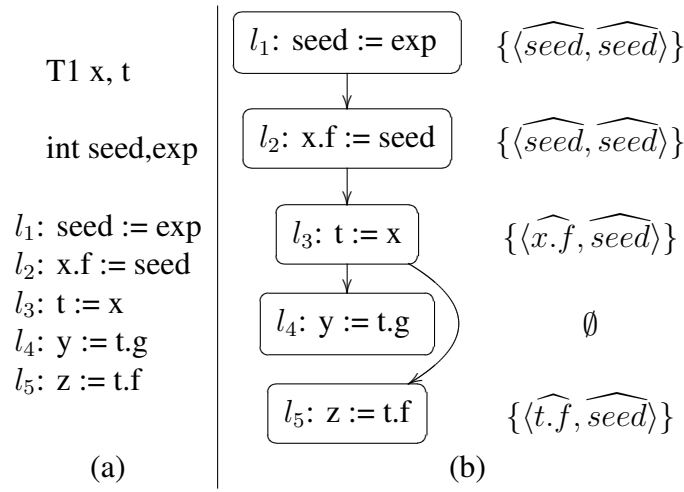


Figure 1.4: FSD analysis of the program in Fig. 1.1 filters out the spurious dependency of  $t.g$  on  $\text{seed}$ . Type T1 is defined in Fig. 1.1.

## 1.5 Main Contributions

This thesis makes the following contributions:

- We present a novel Interval Based *Field Sensitive Dependences* analysis (FSD), that tackles the problem of handling large structure variables in a highly scalable way with exactly the same precision as the ATOM algorithm.
- We formalize the problem of field-sensitive dependences and proof that FSD is complete with respect to the ATOM analysis (i.e., they report exactly the same results).
- We implemented our algorithm and applied it to real-life ERP programs of over a million LOCs as part of PanayaIA, an industrial impact analysis tool [DLAL<sup>+</sup>08, Pan09].
- We extensively compared our dependences analysis to the two common approaches for dependences using real life programs. We show that our algorithm is more efficient than the state-of-the-art atomization with no loss of precision.

## 1.6 Outline

The remainder of this thesis proceeds as follows: Chapter 2 defines our program model and the standard notion of a program dependence graph. Chapter 3 formally defines the problem of field sensitive dependences and provides an instrumented semantics and the vanilla based static analysis for a simplified setting of the problem. Next, we formally define the ATOM algorithm in Chapter 4 and the FSD algorithm in Chapter 5 for the same simplified settings and show that they are both complete with respect to the vanilla approach. The full FSD for real world large scale programs is presented in Chapter 6. Chapter 7 describes our implementation and experimental results. Related work is summarized in Chapter 8 and Chapter 9 discusses further work and concludes the thesis.

# Chapter 2

## Preliminary

In this chapter, we describe our program model and the standard notion of a program dependency graph.

### 2.1 Program Model

We analyze imperative procedural single-threaded programs. A program consists of a collection of procedures. The programmer can also define his own types (à la C structs). Pointers and references are forbidden.

#### 2.1.1 Syntactic Domains

We assume the syntactic domains  $x \in \mathcal{V}$  of variable identifiers,  $f \in \mathcal{F}$  of field identifiers,  $T \in \mathcal{T}$  of type identifiers, and  $p \in \mathcal{P}$  of procedure identifiers. We assume that every variable, type, field, and procedure has a unique identifier in every program.

For simplicity, we assume that we work with a fixed arbitrary program  $P$  with statements *LABEL*. We denote the set of variables in  $P$  by  $\mathcal{V}_P \subset_{fin} \mathcal{V}$ , and the set of global variables in  $P$  by  $G_P \subset \mathcal{V}_P$ . We denote the type of a variable  $x \in \mathcal{V}_P$  by  $T(x)$ .

#### 2.1.2 Aggregate Structures

The declaration `struct  $T$  { $T_1 f_1$ ; ... ; $T_k f_k$ }` declares  $T$  to be an aggregate type containing fields  $f_1, \dots, f_k$  of types  $T_1, \dots, T_k$ , respectively. The type of a field  $f$  is either `int`, `boolean`, or an aggregate structure. Structures can be nested. However, recursive types or not allowed, i.e.,  $T$  cannot (transitively) contain a field of type  $T$ . We denote the type of a field  $f$  by  $T(f)$ .

**Field Path.** A *field path*  $\alpha, \beta \in \mathcal{FP} = \mathcal{F}^*$  is a (possibly empty) sequence of field identifiers separated by a dot. The empty sequence is denoted by  $\epsilon$ . We denote the type of a field path  $\alpha.f$  by  $T(\alpha.f) = T(f)$ . A field path  $\alpha$  is *legal* if either (i)  $\alpha = \epsilon$ , (ii)  $\alpha = f$ , or (iii)  $\alpha = \beta.f.g$ ,  $\beta.f$  is legal, and  $T(f)$  has a field named  $g$ . We denote by  $\mathcal{FP}_T$  the set of valid field paths starting at a field of  $T$ . (Note that  $\mathcal{FP}_T$  is a bounded set.) We denote by  $\mathcal{FP}$  the union of  $\mathcal{FP}_T$  for every type  $T$  in  $\mathcal{T}_P$ .

We define the relation  $\alpha \supset \beta$  for representing that field path  $\alpha$  is a prefix of the field path  $\beta$  as follow:

$$\alpha \supset \beta = \begin{cases} true & \exists \alpha' \in \mathcal{FP}_{T(\alpha)} : \beta = \alpha.\alpha' \wedge \alpha' \neq \epsilon \\ false & otherwise \end{cases}$$

In a similar way we define the relation  $\alpha \supseteq \beta$ .

**Access Path.** An *access path*  $\langle x, \alpha \rangle \in \mathcal{AP} = \mathcal{V} \times \mathcal{FP}$  is a pair consisting of a variable and a field path. An access path  $\langle x, \alpha \rangle$ , also denoted as  $x.\alpha$ , is *legal* if  $\alpha$  is a legal field path and if  $\alpha = f.\beta$  then  $T(x)$  has a field named  $f$ .

We define the relation  $x.\alpha \supset y.\beta$  for representing that access path  $x.\alpha$  is a prefix of the access path  $y.\beta$  as follow:

$$x.\alpha \supset y.\beta = \begin{cases} true & x = y \wedge \alpha \supset \beta \\ false & otherwise \end{cases}$$

In a similar way we define the relation  $x.\alpha \supseteq y.\beta$ .

**Transitive Primitive Field Paths.** We denote the group of *transitive primitive field paths* of an access path  $x.\alpha$  as  $\text{pfp}(x.\alpha)$ . We define  $\text{pfp}(x.\alpha)$  as follows:

$$\text{pfp}(x.\alpha) = \{\alpha' \mid \alpha' \in \mathcal{FP}_{T(\alpha)} \wedge \neg \exists \alpha'' \in \mathcal{FP}_{T(\alpha')} : \alpha'' \neq \epsilon\}$$

Essentially, this group contains all the transitive valid field paths that represent primitive fields (i.e., fields with a primitive type). In addition, access paths that represent a primitive fields are referred to as *primitive* access paths.

**Interval Representation.** For each type  $T$  we define its length as follows:

$$length(T) = \begin{cases} 1 & T \equiv \text{boolean} \\ 4 & T \equiv \text{int} \\ \sum_{f_i \in T} \{length(T(f_i))\} & T \text{ is an aggregate} \end{cases}$$

A structure type  $T$  contains a continuous and ordered list of fields  $f_1, \dots, f_k$  of types  $T_1, \dots, T_k$ , respectively. For each field we define a start-offset (included as part of the field)  $start_f$  and an end-offset (not included as part of the field)  $end_f$ .

The end-offset of a field  $f$  is defined as:

$$end_f = start_f + length(T(f))$$

Note that

$$start_{f_1} = 0, \quad start_{f_{i+1}} = end_{f_i}$$

We extend this definition to types and say that for a type  $T$ :

$$start_T = 0, \quad end_T = length(T)$$

We denote the interval representation of field path  $\alpha$  by  $\hat{\alpha} = \langle start_\alpha, end_\alpha \rangle$ . Similarly, we denote the interval representation of access path  $x.\alpha$  by  $\widehat{x.\alpha} = \langle x, \hat{\alpha} \rangle$ . Essentially, for a field path  $\alpha$  (resp. access path  $x.\alpha$ ), the interval  $\hat{\alpha}$  (resp.  $\widehat{x.\alpha}$ ) represent the memory interval that corresponds to the specific field.

### 2.1.3 Primitive Statements

We define the syntax of primitive statements. A statement can be either a simple statement,  $st_{simp}$ , which has its standard meaning, or a guarded command,  $st = x ? st_{simp}$ , which executes  $st_{simp}$  iff the value of boolean variable  $x$  is true.

$st$	:	$st_{simp} \mid x ? st_{simp}$	
$st_{simp}$	:	$x := y$	<i>Identity (I)</i>
		$x := y.f$	<i>Read (R)</i>
		$x.f := y$	<i>Write (W)</i>
		$x := exp(y_1, y_2, \dots, y_n)$	<i>Computational (C)</i>
		$x := call\ foo(y_1, y_2, \dots, y_n)$	<i>Procedure Call (P)</i>
		$goto\ l$	<i>Branch (B)</i>

### 2.1.4 Procedures

A procedure  $p$  has local variables ( $V_p$ ) and formal parameters ( $F_p$ ), which are considered to be local variables, i.e.,  $F_p \subseteq V_p$ . A procedure can only access global variables and its own local variables. We assume control flow is implemented using guarded commands and `goto` statements. (It is straightforward to transform conditional and loops into this format).

### 2.1.5 Simplifying Assumptions

To simplify the presentation, we make the following assumptions. These assumptions simplify the definition of the analysis. In principle, different ones could be used with minor effects on the capabilities of our approach.

1. Formal parameters *cannot* be assigned to.
2. Statements contain at most one field operation. Statements with more than one field access operation are transformed to a sequence of statements with additional temporary variables. For example, statement of the form  $x.g := y.f$  is transformed to  $t:=y.f; x.g := t$ . In addition, we assume that each statements defines a single variable or a single field of a single variable.
3. Only simple variables are used as the arguments to expression statements and procedure calls.
4. Guarded commands are not nested.

## 2.2 Program Dependence Graph (PDG)

### 2.2.1 Def-Use Chains

A statement  $st$  *uses* resp. *defines* an access path if the latter denotes an l-value which is read resp. written by  $st$ .

A basic representation of data flow dependency is def-use chains [OO84]. A def-use edge between statement  $st_1$  and  $st_2$  represents that during execution of  $st_1$  some access-path is assigned which may be (partially) used in the execution of  $st_2$ .

### 2.2.2 PDG

The *program dependence graph* (PDG) represents the data and control dependencies in a procedure. The nodes of the graph are the statements. The edges are the def-use edges and control dependency edges which are computed by postdomination on the control flow graph [Tip95]. The PDG contains *data* dependency edges and *control* dependency edges. A data dependency between  $st_1$  and  $st_2$  represents that  $st_1$  defines an l-value that is used in  $st_2$ . A control dependency between  $st_1$  and (the necessarily guarded command)  $st_2$  represents that  $st_1$  defines a variable  $x$  which is the guard of  $st_2$ .

# Chapter 3

## Field Sensitive Transitive Dependences

The goal of this work is to find in a precise and in a scalable manner the field sensitive transitive dependences in the presence of aggregate structures with *Big L-values* (operations where an assignment writes to a whole structure). Traditionally, given a control flow graph of a program, a node  $n$  is *flow dependent* on a node  $m$  if there exists a path in the control flow in which the value assigned in  $m$  is directly used at  $n$ . Similarly,  $n$  is *transitively dependent* on  $m$  if there exists a path in the control flow graph in which the value assigned in  $m$  is indirectly used at  $n$  (of course this is an over approximation since not all CFG paths are feasible). We extend this definition to *field sensitive flow dependent* as follows: an access path  $x.\alpha$  used at node  $n$  is *field sensitive flow dependent* on an access path  $s.\beta$  assigned at node  $m$  if there exists a path in the control flow in which the value assigned in  $m$  to  $s.\beta$  is directly used as the value of  $x.\alpha$  at  $n$ . Thus, the traditional flow-dependent relation is between statements while the field sensitive flow dependent relation is between pairs of access-path and statement.

There are several factors that complicate this definition in real life programming languages: pointers and dynamic allocations (e.g., [HPR89b]), Big L-Values, and procedures (e.g., [HRB90]). We define *Field Sensitive Transitive Dependence* as a transitive dependence which handles precisely aggregate structures and Big L-values (without pointers). Notice that Big L-values make the problem more complex since the flow dependences are not transitive, i.e., it may be that  $m$  is directly flow dependent on  $n$  and  $n$  is directly flow dependent on  $p$  and yet  $m$  is not transitively flow dependent on  $p$ .

Consider the example in Fig. 1.1(a). It is clear that  $l_4$  is flow dependent on  $l_3$  (the use  $t.g$  is flow dependent on the definition of  $t$ ), and  $l_3$  is flow dependent on  $l_2$  (the use of  $x$  and particularly  $x.f$  is flow dependent of the definition of  $x.f$ ), yet their transitive closure yields a spurious dependence of  $t.g$  in  $l_4$  on the definition of  $seed$  in  $l_1$ .

In this chapter, we assume the following:

- Programs are type-safe, i.e., assignments are done only between access paths of matching types.
- Programs do not include guarded commands and *goto* statements.
- We compute only transitive value dependences. i.e., we do not compute computational or control dependences.
- The dependence analysis is intraprocedural, we assume no *call* statements.

```

struct T1 {
    int f1
    int f2
    int f3
}

struct T2 {
    int g1
    int g2
    T1 g3
}

T1 exp1,exp2,s,a
T2 b
int kill,t,x,y

```

Figure 3.1: Declaration of Types and Variables used in the examples.

These assumptions are also used in Chapter 4 and Chapter 5. In addition, in all three chapters we use the type and variable declarations shown in Fig. 3.1 in our examples.

The remainder of this chapter proceeds as follows: first, we define a concrete semantics that given a specific trace of the program enables us to determine which field-sensitive dependences exists between the statement  $st_{seed}$  to  $st_{target}$ . Intuitively, this semantics tracks dependences of *primitive* access-paths, those access-path that are of primitive type and not aggregate. This enables a rather simple and straightforward handling of the complexity of aggregates such as the Big L-value assignment and partial-kills. Then we define a sound vanilla static analysis which computes dependences from  $st_{seed}$  given a path (may be infeasible) on the program's CFG. This vanilla analysis is used only to show that the scalable interval based field sensitive analysis presented in Chapter 5 is complete and as precise as the atomization based analysis presented in Chapter 4.

### 3.1 Concrete Instrumented Semantics

We define the instrumented state  $\sigma^{\natural} = \langle pc, \rho, dep \rangle \in \Sigma^{\natural} = PC \times ENV \times DEP$ , where  $ENV = \mathcal{AP} \rightarrow VAL$  and  $DEP = 2^{\mathcal{AP} \times \mathcal{AP} \times LABEL}$ . The 3-tuple represents for each point in the trace the value of the program counter, the values each of the variables holds and the current field sensitive transitive dependences. A dependency  $d = \langle x.\beta, seed.\alpha, l \rangle \in dep$  implies that the value of  $seed.\alpha$  which was assigned at line  $l$  transitively flows to  $x.\beta$  at the given state. The transformer of a statement  $st$  is the function  $\llbracket st \rrbracket^{\natural} : \Sigma^{\natural} \rightarrow \Sigma^{\natural}$ . The execution of a program starts with the initial empty state  $\sigma_0^{\natural} = \langle 0, \emptyset, \emptyset \rangle$ . This chapter focuses on intraprocedural data flow dependences only, therefore we show the transformer only for assignments statements in our simplified programming language, as only these type of statements can change the data dependences of the program. We generalize for control dependences in Chapter 6. In general the execution of an assignment  $l$  to some primitive access-path  $ap$  removes dependences that were previously on  $ap$  and adapts dependences that are on the used access-path. In addition, a dependency  $\langle ap, ap, l \rangle$  is added in order to track dependences on  $ap$  which is assigned at  $l^1$ .

We use the programs shown in figures 3.2 to 3.5 in order to further explain the transformers. We focus on the dependency part ( $dep$ ) of the concrete state  $\sigma^{\natural}$  as the effect of the transfer

<sup>1</sup>We formalize the analysis for tracking dependences on all final access-path. For the case of tracking dependences on  $seed$  only, a dependency  $\langle ap, ap, l \rangle$  is added only to (partial) assignments to  $seed$

$$\begin{array}{l|l}
l_1: s := \text{exp1} & \begin{array}{l} \text{dep}_1 = \emptyset \\ \text{dep}_2 = \{\langle s.f1, s.f1, l_1 \rangle, \langle s.f2, s.f2, l_1 \rangle, \langle s.f3, s.f3, l_1 \rangle\} \end{array} \\
l_2: s := \text{exp2} & \text{dep}_3 = \{\langle s.f1, s.f1, l_2 \rangle, \langle s.f2, s.f2, l_2 \rangle, \langle s.f3, s.f3, l_2 \rangle\} \\
l_3: a := s & \text{dep}_4 = \{\langle s.f1, s.f1, l_2 \rangle, \langle s.f2, s.f2, l_2 \rangle, \langle s.f3, s.f3, l_2 \rangle, \\
& \quad \langle a.f1, s.f1, l_2 \rangle, \langle a.f2, s.f2, l_2 \rangle, \langle a.f3, s.f3, l_2 \rangle, \\
& \quad \langle a.f1, a.f1, l_3 \rangle, \langle a.f2, a.f2, l_3 \rangle, \langle a.f3, a.f3, l_3 \rangle\}
\end{array}$$

Figure 3.2: Example of the process of the instrumented semantics on statements of type I.

function on the program counter ( $pc$ ) and environment ( $\rho$ ) is straightforward.

### 3.1.1 Identity Statements

Identity statements (type  $I$ ) are assignments of the form  $x := y$ . The transfer function for type  $I$  statements is

$$\begin{aligned}
\llbracket l : x := y \rrbracket^{\sharp}(pc, \rho, dep) &= \langle pc + 1, \rho', dep' \rangle \\
\rho' &= \rho[x.\alpha \mapsto \rho(y.\alpha) \mid \alpha \in \text{pfp}(x)] \\
dep' &= \{ \langle t.\alpha, s.\beta, l' \rangle \in dep \mid t \neq x \} \cup \quad (3.1.1) \\
& \quad \{ \langle x.\alpha, s.\beta, l' \rangle \mid \langle y.\alpha, s.\beta, l' \rangle \in dep \} \cup \quad (3.1.2) \\
& \quad \{ \langle x.\alpha, x.\alpha, l \rangle \mid \alpha \in \text{pfp}(x) \} \quad (3.1.3)
\end{aligned} \tag{3.1}$$

**Term (3.1.1)** copies all the dependences from  $dep$  except for those on some field of  $x$ . Dependences on any (transitive) field of  $x$  are "killed". For instance, consider the example in Fig. 3.2, the application of the transformer of  $l_2$  on  $dep_2$  removes all the existing dependences on  $s$  that are related to  $l_1$ .

**Term (3.1.2)** generates a new dependency on  $x$  for each dependency that already exist on  $y$ . Because the assignment is type-safe, for each field path of  $y$  there is a corresponding field path of  $x$ . In Fig. 3.2, the application of the transformer of  $l_3$  on  $dep_3$  adds a dependency on  $a$  for each dependency that already exist on  $s$ .

**Term (3.1.3)** generates the "self" dependency for each (transitive) primitive field of  $x$ . In Fig. 3.2, the application of the transformer of  $l_1$  on  $dep_1$  adds a self dependency  $\langle ap, ap, l \rangle$  for each primitive field path of  $s$ :  $s.f1$ ,  $s.f2$  and  $s.f3$ .

### 3.1.2 Write Statements

Write statements (type  $W$ ) are assignments of the form  $x.f := y$ . The transfer function for type  $W$  statements is

$$\begin{aligned}
\llbracket l : x.f := y \rrbracket^{\sharp}(pc, \rho, rd) &= \langle pc + 1, \rho', rd' \rangle \\
\rho' &= \rho[x.f.\alpha \mapsto \rho(y.\alpha) \mid \alpha \in \text{pfp}(x.f)] \\
dep' &= \{ \langle t.\alpha, s.\beta, l' \rangle \in dep \mid t \neq x.f \} \cup \quad (3.2.1) \\
& \quad \{ \langle x.f.\alpha, s.\beta, l' \rangle \mid \langle y.\alpha, s.\beta, l' \rangle \in dep \} \cup \quad (3.2.2) \\
& \quad \{ \langle x.f.\alpha, x.f.\alpha, l \rangle \mid \alpha \in \text{pfp}(x) \} \quad (3.2.3)
\end{aligned} \tag{3.2}$$

$$\begin{array}{l|l}
l_1: s := \text{exp1} & dep_1 = \emptyset \\
l_2: s.f2 := \text{kill} & dep_2 = \{\langle s.f1, s.f1, l_1 \rangle, \langle s.f2, s.f2, l_1 \rangle, \langle s.f3, s.f3, l_1 \rangle\} \\
l_3: b.g3 := s & dep_3 = \{\langle s.f1, s.f1, l_1 \rangle, \langle s.f2, s.f2, l_2 \rangle, \langle s.f3, s.f3, l_1 \rangle\} \\
& dep_4 = \{\langle s.f1, s.f1, l_1 \rangle, \langle s.f2, s.f2, l_2 \rangle, \langle s.f3, s.f3, l_1 \rangle, \\
& \quad \langle b.g3.f1, s.f1, l_1 \rangle, \langle b.g3.f2, s.f2, l_2 \rangle, \langle b.g3.f3, s.f3, l_1 \rangle, \\
& \quad \langle b.g3.f1, b.g3.f1, l_3 \rangle, \langle b.g3.f2, b.g3.f2, l_3 \rangle, \langle b.g3.f3, b.g3.f3, l_3 \rangle\}
\end{array}$$

Figure 3.3: Example of the process of the instrumented semantics on statements of type W.

**Term (3.2.1)** copies all the dependences from  $dep$  except for those on some field of  $x.f$ . Dependences on any (transitive) field of  $x.f$  are "killed". For instance, consider the example in Fig. 3.3, the application of the transformer of  $l_2$  on  $dep_2$  removes the existing dependency on  $s.f2$  that is related to  $l_1$ .

**Term (3.2.2)** generates a new dependency on  $x.f$  for each dependency that already exist on  $y$ . In Fig. 3.3, the application of the transformer of  $l_3$  on  $dep_3$  adds a dependency on  $b.g3$  for each dependency that already exist on  $s$  (i.e.,  $b.g3.f1$ ,  $b.g3.f2$  and  $b.g3.f3$ ).

**Term (3.2.3)** generates the "self" dependency for each (transitive) primitive field of  $x.f$ . In Fig. 3.3, the application of the transformer of  $l_3$  on  $dep_3$  adds a self dependency  $\langle ap, ap, l \rangle$  for each primitive field path of  $b.g3$ :  $b.g3.f1$ ,  $b.g3.f2$  and  $b.g3.f3$ .

### 3.1.3 Read Statements

Read statements (type  $R$ ) are assignments of the form  $x := y.f$ . The transfer function for type  $R$  statements is

$$\begin{aligned}
\llbracket l : x := y.f \rrbracket^{\sharp}(pc, \rho, rd) &= \langle pc + 1, \rho', rd' \rangle \\
\rho' &= \rho[x.\alpha \mapsto \rho(y.f.\alpha) | \alpha \in \text{pfp}(x)] \\
dep' &= \{ \langle t.\alpha, s.\beta, l' \rangle \in dep | t \neq x \} \cup \quad (3.3.1) \\
&\quad \{ \langle x.\alpha, s.\beta, l' \rangle | \langle y.f.\alpha, s.\beta, l' \rangle \in dep \} \cup \quad (3.3.2) \\
&\quad \{ \langle x.\alpha, x.f.p, l \rangle | \alpha \in \text{pfp}(x) \} \quad (3.3.3)
\end{aligned} \tag{3.3}$$

**Term (3.3.1)** copies all the dependences from  $dep$  except for those on some field of  $x$  (the same as for type I). Dependences on any (transitive) field of  $x$  are "killed". For instance, consider the example in Fig. 3.4, the application of the transformer of  $l_2$  on  $dep_2$  removes all the existing dependences on  $s$  that are related to  $l_1$ .

**Term (3.3.2)** generates a new dependency on  $x$  for each dependency that already exist on  $y.f$ . In Fig. 3.4, the application of the transformer of  $l_3$  on  $dep_3$  adds a dependency on  $t$  according to the dependency that already exists on  $s.f3$ .

**Term (3.3.3)** generates the "self" dependency for each (transitive) primitive field of  $x$  (the same as for type I). In Fig. 3.4, the application of the transformer of  $l_2$  on  $dep_2$  adds a self dependency  $\langle ap, ap, l \rangle$  for each primitive field path of  $s$ :  $s.f1$ ,  $s.f2$  and  $s.f3$ .

$$\begin{array}{l|l}
l_1: s := \text{exp1} & dep_1 = \emptyset \\
l_2: s := \text{b.g3} & dep_2 = \{\langle s.f1, s.f1, l_1 \rangle, \langle s.f2, s.f2, l_1 \rangle, \langle s.f3, s.f3, l_1 \rangle\} \\
l_3: t := \text{s.f3} & dep_3 = \{\langle s.f1, s.f1, l_2 \rangle, \langle s.f2, s.f2, l_2 \rangle, \langle s.f3, s.f3, l_2 \rangle\} \\
& dep_4 = \{\langle s.f1, s.f1, l_2 \rangle, \langle s.f2, s.f2, l_2 \rangle, \langle s.f3, s.f3, l_2 \rangle, \\
& \quad \langle t, s.f3, l_2 \rangle, \\
& \quad \langle t, t, l_3 \rangle\}
\end{array}$$

Figure 3.4: Example of the process of the instrumented semantics on statements of type R.

$$\begin{array}{l|l}
l_1: s := \text{exp1} & dep_1 = \emptyset \\
l_2: s := \text{inc}(s) & dep_2 = \{\langle s.f1, s.f1, l_1 \rangle, \langle s.f2, s.f2, l_1 \rangle, \langle s.f3, s.f3, l_1 \rangle\} \\
l_3: a := \text{dec}(s) & dep_3 = \{\langle s.f1, s.f1, l_2 \rangle, \langle s.f2, s.f2, l_2 \rangle, \langle s.f3, s.f3, l_2 \rangle\} \\
& dep_4 = \{\langle s.f1, s.f1, l_2 \rangle, \langle s.f2, s.f2, l_2 \rangle, \langle s.f3, s.f3, l_2 \rangle \\
& \quad \langle a.f1, a.f1, l_3 \rangle, \langle a.f2, a.f2, l_3 \rangle, \langle a.f3, a.f3, l_3 \rangle\}
\end{array}$$

Figure 3.5: Example of the process of the instrumented semantics on statements of type C.

### 3.1.4 Computational assignment

Computational statements (type  $C$ ) are assignments of the form  $x := \text{exp}(y_1, y_2, \dots, y_n)$ . Unlike transfer function of the other assignment statements, in this case, since we are handling only value flow dependences, we do not adapt existing dependences on  $y_i$  to  $x$ .

$$\begin{aligned}
\llbracket l : x := \text{exp}(y_1, y_2, \dots, y_n) \rrbracket^{\sharp}(pc, \rho, dep) &= \langle pc + 1, \rho', dep' \rangle \\
\rho' &= \rho[x.\alpha \mapsto \rho(t.\alpha) \mid \alpha \in \text{pfp}(x), t = \llbracket \text{exp} \rrbracket(\rho(y_1), \rho(y_2), \dots, \rho(y_n))] \\
dep' &= \{ \langle t.\alpha, s.\beta, l' \rangle \in dep \mid t \neq x \} \cup \{ \langle x.\alpha, x.\alpha, l \rangle \mid \alpha \in \text{pfp}(x) \}
\end{aligned} \tag{3.4}$$

**Term (3.4.1)** copies all the dependences from  $dep$  except for those on some field of  $x$  (the same as for type I). Dependences on any (transitive) field of  $x$  are "killed". For instance, consider the example in Fig. 3.5, the application of the transformer of  $l_2$  on  $dep_2$  removes all the existing dependences on  $s$  that are related to  $l_1$ .

**Term (3.4.2)** generates the "self" dependency for each (transitive) primitive field of  $x$  (the same as for type I). In Fig. 3.5, the application of the transformer of  $l_3$  on  $dep_3$  adds a self dependency  $\langle ap, ap, l \rangle$  for each primitive field path of  $a$ :  $a.f1$ ,  $a.f2$  and  $a.f3$ .

### 3.1.5 Inferring Dependences

Inferring the dependences using the concrete state is straightforward as the concrete state in a certain point on the trace already contains the transitive dependences (in  $dep$ ).

We define the field sensitive flow dependency as a boolean function  $Dep^{\sharp}$  on a given state  $\sigma^{\sharp}$  that arises before the execution of  $l$ .  $Dep^{\sharp}(l_s, s.\beta, l_t, t.\alpha)$  holds if and only if the value of  $t.\alpha$  used at the execution of statement  $l_t$  is flow dependent on the value of  $s.\beta$  assigned at  $l_s$  and  $s.\beta$  and  $t.\alpha$  are primitive access-paths.

Formally:

### Definition 3.1.1

$$Dep^{\sharp}(l_s, s.\beta, l_t, t.\alpha) = \begin{cases} true & \langle t.\alpha, s.\beta, l_s \rangle \in \sigma^{\sharp} \\ false & otherwise \end{cases} \quad (3.5)$$

## 3.2 Vanilla Static Analysis

We define a static analysis algorithm that given a program's CFG path (maybe infeasible) computes the transitive dependences from a statement  $st_{seed}$  to each statement in the path. A single (possible transitive) dependence is represented by a tuple  $d = \langle v.\alpha, seed.\beta, l, k \rangle \in D = \mathcal{AP} \times \mathcal{AP}_{seed} \times Label \times \mathbb{Z}$ . A transitive dependency  $x.\alpha \leftarrow seed.\beta$  where  $seed.\beta$  was assigned in  $l$  is represented by the tuple  $\langle x.\alpha, seed.\beta, l, k \rangle$  where  $k$  is the number of immediate value flow dependences that compose this transitive dependence. The  $k$  member is an instrumented information tracked only for the ease of proving the completeness of the atomization based and interval based algorithms.

The abstract domain is  $\Sigma = 2^D$ : a powerset domain over the set of tuples. The abstract transformer of a statement  $st$  is a function  $\llbracket st \rrbracket : \Sigma \rightarrow \Sigma$ . Similarly to the instrumented semantics, the effect of a statement  $st$  on an abstract state  $\sigma \in \Sigma$  (a set of tuples) is to remove some dependences that are no longer true and to generate new ones. The initial state is the empty state  $\sigma_0 = \emptyset$  and for a path with statements  $\{st_0; st_1; \dots; st_n\}$  we denote

$$\sigma_{i+1} = \llbracket l_i : st_i \rrbracket(\sigma_i)$$

### 3.2.1 Identity Statements

$$\llbracket l : x := y \rrbracket(\sigma) = \{ \langle t.\alpha, s.\beta, l', k \rangle \in \sigma \mid t \neq x \} \cup \quad (3.6.1)$$

$$\{ \langle x.\alpha, s.\beta, l', k+1 \rangle \mid \langle y.\alpha, s.\beta, l', k \rangle \in \sigma \} \cup \quad (3.6.2) \quad (3.6)$$

$$\{ \langle x.\alpha, x.\alpha, l, 0 \rangle \mid \alpha \in \text{pfp}(x) \} \quad (3.6.3)$$

**Term (3.6.1)** copies all the dependences from  $\sigma$  except for those on some field of  $x$ . Dependences on any (transitive) field of  $x$  are "killed". For instance, consider the example in Fig. 3.6, the application of the transformer of  $l_2$  on  $\sigma_2$  removes all the existing dependences on  $s$  that are related to  $l_1$ .

**Term (3.6.2)** generates a new dependency on  $x$  for each dependency that already exist on  $y$ . In addition  $k$ , the number of immediate dependency steps, is increased. In Fig. 3.6, the application of the transformer of  $l_3$  on  $\sigma_3$  adds a dependency on  $a$  for each dependency that already exist on  $s$  (i.e., on  $a.f1, a.f2$  and  $a.f3$ ) and updates the number of dependency steps from 0 to 1.

**Term (3.6.3)** generates the "self" dependency for each (transitive) primitive field of  $x$ . In Fig. 3.6, the application of the transformer of  $l_3$  on  $\sigma_3$  adds a self dependency  $\langle ap, ap, l, 0 \rangle$  for each primitive field path of  $a$ :  $a.f1, a.f2$  and  $a.f3$ .

$$\begin{array}{l|l}
l_1: s := \text{exp1} & \sigma_1 = \emptyset \\
l_2: s := \text{exp2} & \sigma_2 = \{\langle s.f1, s.f1, l_1, 0 \rangle, \langle s.f2, s.f2, l_1, 0 \rangle, \langle s.f3, s.f3, l_1, 0 \rangle\} \\
l_3: a := s & \sigma_3 = \{\langle s.f1, s.f1, l_2, 0 \rangle, \langle s.f2, s.f2, l_2, 0 \rangle, \langle s.f3, s.f3, l_2, 0 \rangle\} \\
& \sigma_4 = \{\langle s.f1, s.f1, l_2, 0 \rangle, \langle s.f2, s.f2, l_2, 0 \rangle, \langle s.f3, s.f3, l_2, 0 \rangle, \\
& \quad \langle a.f1, s.f1, l_2, 1 \rangle, \langle a.f2, s.f2, l_2, 1 \rangle, \langle a.f3, s.f3, l_2, 1 \rangle, \\
& \quad \langle a.f1, a.f1, l_3, 0 \rangle, \langle a.f2, a.f2, l_3, 0 \rangle, \langle a.f3, a.f3, l_3, 0 \rangle\}
\end{array}$$

Figure 3.6: Example of the process of the vanilla static analysis on statements of type I.

$$\begin{array}{l|l}
l_1: s := \text{exp1} & \text{dep}_1 = \emptyset \\
l_2: s.f2 := \text{kill} & \text{dep}_2 = \{\langle s.f1, s.f1, l_1, 0 \rangle, \langle s.f2, s.f2, l_1, 0 \rangle, \langle s.f3, s.f3, l_1, 0 \rangle\} \\
l_3: b.g3 := s & \text{dep}_3 = \{\langle s.f1, s.f1, l_1, 0 \rangle, \langle s.f2, s.f2, l_2, 0 \rangle, \langle s.f3, s.f3, l_1, 0 \rangle\} \\
& \text{dep}_4 = \{\langle s.f1, s.f1, l_1, 0 \rangle, \langle s.f2, s.f2, l_2, 0 \rangle, \langle s.f3, s.f3, l_1, 0 \rangle, \\
& \quad \langle b.g3.f1, s.f1, l_1, 1 \rangle, \langle b.g3.f2, s.f2, l_2, 1 \rangle, \langle b.g3.f3, s.f3, l_1, 1 \rangle, \\
& \quad \langle b.g3.f1, b.g3.f1, l_3, 0 \rangle, \langle b.g3.f2, b.g3.f2, l_3, 0 \rangle, \langle b.g3.f3, b.g3.f3, l_3, 0 \rangle\}
\end{array}$$

Figure 3.7: Example of the process of the vanilla static analysis on statements of type W.

### 3.2.2 Write Statements

$$[[l : x.f := y]](\sigma) = \{\langle t.\alpha, s.\beta, l', k \rangle \in \sigma \mid t \neq x.f\} \cup \quad (3.7.1)$$

$$\{\langle x.f.\alpha, s.\beta, l', k + 1 \rangle \mid \langle y.\alpha, s.\beta, l', k \rangle \in \sigma\} \cup \quad (3.7.2) \quad (3.7)$$

$$\{\langle x.f.\alpha, x.f.\alpha, l, 0 \rangle \mid \alpha \in \text{pfp}(x)\} \quad (3.7.3)$$

**Term (3.7.1)** copies all the dependences from  $\sigma$  except for those on some field of  $x.f$ . Dependences on any (transitive) field of  $x.f$  are "killed". For instance, consider the example in Fig. 3.7, the application of the transformer of  $l_2$  on  $\sigma_2$  removes the existing dependences on  $s.f2$  that is related to  $l_1$ .

**Term (3.7.2)** generates a new dependency on  $x.f$  for each dependency that already exist on  $y$ . In Fig. 3.7, the application of the transformer of  $l_3$  on  $\text{dep}_3$  adds a dependency on  $b.g3$  for each dependency that already exist on  $s$  (i.e.,  $b.g3.f1$ ,  $b.g3.f2$  and  $b.g3.f3$ ). In addition it update the number of immediate dependency steps from 0 to 1.

**Term (3.7.3)** generates the "self" dependency for each (transitive) primitive field of  $x.f$ . In Fig. 3.7, the application of the transformer of  $l_3$  on  $\text{sigma}_3$  adds a self dependency  $\langle ap, ap, l, 0 \rangle$  for each primitive field path of  $b.g3$ :  $b.g3.f1$ ,  $b.g3.f2$  and  $b.g3.f3$ .

### 3.2.3 Read Statements

$$[[l : x := y.f]](\sigma) = \{\langle t.\alpha, s.\beta, l', k \rangle \in \sigma \mid t \neq x\} \cup \quad (3.8.1)$$

$$\{\langle x.\alpha, s.\beta, l', k + 1 \rangle \mid \langle y.f.\alpha, s.\beta, l', k \rangle \in \sigma\} \cup \quad (3.8.2) \quad (3.8)$$

$$\{\langle x.\alpha, x.\alpha, l, 0 \rangle \mid \alpha \in \text{pfp}(x)\} \quad (3.8.3)$$

**Term (3.8.1)** copies all the dependences from  $\sigma$  except for those on some field of  $x$  (the same as for type I). Dependences on any (transitive) field of  $x$  are "killed". For instance, consider

$$\begin{array}{l|l}
l_1: s := \text{exp1} & \sigma_1 = \emptyset \\
l_2: s := \text{b.g3} & \sigma_2 = \{ \langle s.f1, s.f1, l_1, 0 \rangle, \langle s.f2, s.f2, l_1, 0 \rangle, \langle s.f3, s.f3, l_1, 0 \rangle \} \\
l_3: t := \text{s.f3} & \sigma_3 = \{ \langle s.f1, s.f1, l_2, 0 \rangle, \langle s.f2, s.f2, l_2, 0 \rangle, \langle s.f3, s.f3, l_2, 0 \rangle \} \\
& \sigma_4 = \{ \langle s.f1, s.f1, l_2, 0 \rangle, \langle s.f2, s.f2, l_2, 0 \rangle, \langle s.f3, s.f3, l_2, 0 \rangle, \\
& \quad \langle t, s.f3, l_2, 1 \rangle, \\
& \quad \langle t, t, l_3, 0 \rangle \}
\end{array}$$

Figure 3.8: Example of the process of the vanilla static analysis on statements of type R.

the example in Fig. 3.8, the application of the transformer of  $l_2$  on  $\sigma_2$  removes all the existing dependences on  $s$  that are related to  $l_1$ .

**Term (3.8.2)** generates a new dependency on  $x$  for each dependency that already exist on  $y.f$ . In Fig. 3.8, the application of the transformer of  $l_3$  on  $\sigma_3$  adds a dependency on  $t$  according to the dependency that already exists on  $s.f3$  and update the number immediate dependency steps from 0 to 1.

**Term (3.8.3)** generates the "self" dependency for each (transitive) primitive field of  $x$  (the same as for type I). In Fig. 3.8, the application of the transformer of  $l_2$  on  $\sigma_2$  adds a self dependency  $\langle ap, ap, l, 0 \rangle$  for each primitive field path of  $s$ :  $s.f1$ ,  $s.f2$  and  $s.f3$ .

### 3.2.4 Computational Statements

$$\llbracket l : x := \text{exp}(y_1, y_2, \dots, y_n) \rrbracket(\sigma) = \{ \langle t.\alpha, s.\beta, l', k \rangle \in \sigma \mid t \neq x \} \cup \quad (3.9.1) \quad (3.9)$$

$$\{ \langle x.\alpha, x.\alpha, l, 0 \rangle \mid \alpha \in \text{pfp}(x) \} \quad (3.9.2)$$

Unlike transfer function of the other assignment statements, in this case, since we are handling only value flow dependences, we do not adapt existing dependences on  $y_i$  to  $x$ . Basically,  $\llbracket l : x := \text{exp}(y_1, y_2, \dots, y_n) \rrbracket(\sigma)$  updates  $\sigma$  in two steps:

**Term (3.9.1)** copies all the dependences from  $\sigma$  except for those on some field of  $x$  (the same as for type I). Dependences on any (transitive) field of  $x$  are "killed". For instance, consider the example in Fig. 3.9, the application of the transformer of  $l_2$  on  $\sigma_2$  removes all the existing dependences on  $s$  that are related to  $l_1$ .

**Term (3.9.3)** generates the "self" dependency for each (transitive) primitive field of  $x$  (the same as for type I). In Fig. 3.9, the application of the transformer of  $l_3$  on  $\sigma_3$  adds a self dependency  $\langle ap, ap, l, 0 \rangle$  for each primitive field path of  $a$ :  $a.f1$ ,  $a.f2$  and  $a.f3$ .

### 3.2.5 Computing the Dependences

**Lemma 3.2.1** *The static analysis definition of transitive dependences is sound in relation to the concrete semantics.*

$$\begin{array}{l|l}
l_1: s := \text{exp1} & \sigma_1 = \emptyset \\
l_2: s := \text{inc}(s) & \sigma_2 = \{ \langle s.f1, s.f1, l_1, 0 \rangle, \langle s.f2, s.f2, l_1, 0 \rangle, \langle s.f3, s.f3, l_1, 0 \rangle \} \\
l_3: a := \text{dec}(s) & \sigma_3 = \{ \langle s.f1, s.f1, l_2, 0 \rangle, \langle s.f2, s.f2, l_2, 0 \rangle, \langle s.f3, s.f3, l_2, 0 \rangle \} \\
& \sigma_4 = \{ \langle s.f1, s.f1, l_2, 0 \rangle, \langle s.f2, s.f2, l_2, 0 \rangle, \langle s.f3, s.f3, l_2, 0 \rangle \\
& \quad \langle a.f1, a.f1, l_3, 0 \rangle, \langle a.f2, a.f2, l_3, 0 \rangle, \langle a.f3, a.f3, l_3, 0 \rangle \}
\end{array}$$

Figure 3.9: Example of the process of the vanilla static analysis on statements of type C.

This can easily be shown by showing that for each state  $\sigma^{\natural}$  and abstract state  $\sigma$  where  $\sigma$  is a sound representation of  $\sigma^{\natural}$  then for each statement  $st$ :

$$\llbracket st \rrbracket(\sigma) \text{ is a sound representation of } \llbracket st \rrbracket^{\natural}(\sigma^{\natural})$$

■

Based on the results of the static analysis, we define the field sensitive flow dependency as a boolean function  $Dep$  on a given state  $\sigma$  that arises before the execution of  $l$ .  $Dep(l_s, s.\beta, l_t, t.\alpha)$  holds if and only if the value of  $t.\alpha$  used at the execution of statement  $l_t$  is flow dependent on the value of  $s.\beta$  assigned at  $l_s$  and  $s.\beta$  and  $t.\alpha$  are primitive access-paths.

Formally:

### Definition 3.2.2

$$Dep(l_s, s.\beta, l_t, t.\alpha) = \begin{cases} true & \langle t.\alpha, s.\beta, l_s, k \rangle \in \sigma \\ false & otherwise \end{cases} \quad (3.10)$$

*Note that field sensitive flow dependent of non primitive access path can be inferred when the  $Dep$  predicate holds for all possible permutations of inner primitive access paths.*

**Example 3.2.1** Consider the program shown in Fig. 3.10. The query  $Dep(l_1, s.f1, l_3, a.f1)$  returns true because according to (3.10):

$$\langle a.f1, s.f1, l_1, 1 \rangle \in \sigma_3$$

Thus, we conclude that  $a.f1$  in  $l_3$  is (transitive) field sensitive dependent on  $s.f1$  in  $l_1$ .

In a similar way we can show that the query  $Dep(l_1, s.f1, l_4, a.f2)$  returns false, indicating correctly that  $a.f2$  in  $l_4$  is **not** (transitive) field sensitive dependent on  $s.f1$  in  $l_1$ . Note that computing transitive closure of the immediate dependency on this program's PDG (shown in Fig. 3.11) would have concluded falsely that  $a.f2$  in  $l_4$  is transitively flow dependent on  $s.f1$  in  $l_1$ .

## 3.2.6 Complexity

The complexity attributes for the vanilla algorithm are derived in a straightforward way from the definition of the analysis. The algorithm stores at each statement a state representing all the transitive dependences along the path starting from every point on the path (including the seed statements). Each dependency is composed of a primitive access path of the target use and a

$l_1: s.f1 := t$	$\sigma_1 = \emptyset$
$l_2: a := s$	$\sigma_2 = \{\langle s.f1, s.f1, l_1, 0 \rangle\}$
$l_3: x := a.f1$	$\sigma_3 = \{\langle s.f1, s.f1, l_1, 0 \rangle,$ $\langle a.f1, s.f1, l_1, 1 \rangle,$ $\langle a.f1, a.f1, l_2, 0 \rangle, \langle a.f2, a.f2, l_2, 0 \rangle, \langle a.f3, a.f3, l_2, 0 \rangle\}$
$l_4: y := a.f2$	$\sigma_4 = \{\langle s.f1, s.f1, l_1, 0 \rangle, \langle a.f1, s.f1, l_1, 1 \rangle,$ $\langle a.f1, a.f1, l_2, 0 \rangle, \langle a.f2, a.f2, l_2, 0 \rangle, \langle a.f3, a.f3, l_2, 0 \rangle,$ $\langle x, s.f1, l_1, 2 \rangle,$ $\langle x, x, l_3, 0 \rangle\}$
$l_5: y := a.f2$	$\sigma_5 = \{\langle s.f1, s.f1, l_1, 0 \rangle, \langle a.f1, s.f1, l_1, 1 \rangle, \langle x, s.f1, l_1, 2 \rangle,$ $\langle a.f1, a.f1, l_2, 0 \rangle, \langle a.f2, a.f2, l_2, 0 \rangle, \langle a.f3, a.f3, l_2, 0 \rangle,$ $\langle y, a.f2, l_2, 1 \rangle,$ $\langle y, y, l_3, 0 \rangle\}$

Figure 3.10: Example of the *Dep* computation using the vanilla static analysis.

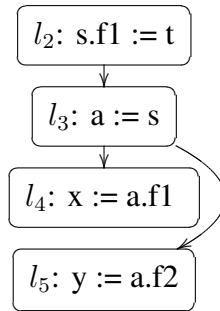


Figure 3.11: The PDG of the example shown in Fig. 3.10.

primitive access path of source def. We denote the size of the group of primitive access paths as  $O(|\mathcal{AP}_{prim}|)$ . We denote the length of the path as  $|p|$  (the number of statements in the CFG path)

Thus the space complexity is:

$$C_{space} = O(|p| \times |\mathcal{AP}_{prim}| \times |\mathcal{AP}_{prim}|)$$

The complexity of computing the dependence is comprised of two parts: (i) the computation of state  $\sigma$  for each statement on the path  $p$  (pre-process) (ii) inferring the field sensitive dependences using the definition of  $Dep$  (query).

When computing the abstract states, each statement is processed once. For each statement, we scan the previous state. Thus, the time complexity of the pre-process:

$$C_{pre} = O(|p| \times |\mathcal{AP}_{prim}| \times |\mathcal{AP}_{prim}|)$$

When computing the  $Dep$  function, the dependences are inferred immediately from the state  $\sigma$  of the given target statement. Thus, The time complexity of the query:

$$C_q = O(|\mathcal{AP}_{prim}| \times |\mathcal{AP}_{prim}|)$$

# Chapter 4

## Atomization Based Field Sensitive Dependences

The vanilla static analysis algorithm defined in Section 3.2 is very naive as the complexity of it is high. A more commonly approach used is based on atomization. In this chapter we define an atomization based static analysis algorithm that computes the field sensitive dependences. The main idea is to perform the dependences calculation on a transformed program that contains only primitive variables and therefore there are no Big L-value assignments nor partial-kill statements. Needless to say, this is feasible only for recursive-free type declarations such as in the case of our programming model. However, as we shall see, this is scalable only for programs with few fields in structure definitions.

Formally, we define a transformation  $Trans(Prog) = Prog^A$ . Program  $Prog^A$  contains the primitive variable  $x_\alpha$  for every variable primitive access-path  $x.\alpha$ . Formally:

$$\{x_\alpha \in \mathcal{V}_{Prog^A} | x \in \mathcal{V}_P, \alpha \in \text{pfp}(x)\}$$

The transformation of the statements is defined as follows:

$$\begin{aligned} Trans(st_1; st_2 : \dots; st_n) &= Trans(st_1); Trans(st_2); \dots; Trans(st_n) \\ Trans(x := y) &= \{x_\alpha = y_\alpha | \alpha \in \text{pfp}(x)\} \\ Trans(x.f := y) &= \{x_{f_\alpha} = y_\alpha | \alpha \in \text{pfp}(x.f)\} \\ Trans(x := y.f) &= \{x_\alpha = y_{f_\alpha} | \alpha \in \text{pfp}(x)\} \\ Trans(x := exp(y_1, y_2, \dots, y_n)) &= \{x_\alpha = exp^\alpha(y_1^\alpha, y_2^\alpha, \dots, y_n^\alpha) | \alpha \in \text{pfp}(x)\} \end{aligned}$$

This transformation disassembles each statement into multiple statements that operate only on primitive variables and not on structures.

Consider the original program's control flow graph  $CFG$  and a path  $p$  on this graph. We define  $CFG^A$  as the control flow graph on the atomized program  $Prog^A$ . The path  $p^A$  which corresponds to the path  $p$  on the original program can be easily constructed by replacing each original node  $v$ , which represents the statement  $st$  in the original program, with the nodes  $v_1, v_2, \dots, v_i$  that represents the corresponding atomized statements  $Trans(st) = st_1; st_2; \dots; st_i$ ;

$$\begin{array}{l|l}
l_1: s\_f1 := \text{exp1\_f1} & \sigma_1^A = \emptyset \\
l_2: s\_f1 := \text{exp2\_f1} & \sigma_2^A = \{s\_f1 \mapsto l_1\} \\
l_3: a\_f1 := s\_f1 & \sigma_3^A = \{s\_f1 \mapsto l_2\} \\
& \sigma_4^A = \{s\_f1 \mapsto l_2, a\_f1 \mapsto l_3\}
\end{array}$$

Figure 4.1: Example of the process of the ATOM static analysis on statements of type I.

## 4.1 Atomization Based Algorithm

Given a path  $p^A$  on  $CFG^A$ , we define a static analysis algorithm that computes flow dependences between program points and show how to infer from this information the field sensitive flow dependences that arise on the equivalent path  $p$  of  $CFG$ . For each program label on the path  $p^A$  we define an abstract state  $\sigma^A \in \Sigma^A = \mathcal{V} \rightarrow LABEL$ . This state represents the reaching-definition that arise at execution of this specific path.

The abstract transformer of a statement  $st$  is a function  $\llbracket st \rrbracket : \Sigma^A \rightarrow \Sigma^A$ . Essentially, the effect of a statement  $st$  on an abstract state  $\sigma^A$  is to remove some reaching definitions that are no longer true and to generate new ones. The initial state is the empty state  $\sigma_0^A = \emptyset$  and for a CFG path with statements  $\{st_0; st_1; \dots; st_n; \}$  we denote

$$\sigma_{i+1}^A = \llbracket l_i : st_i \rrbracket(\sigma_i^A)$$

For convenient reasons, we sometimes refer to statements by their labels and vice-versa.

Note that in the atomized program all the copy assignment statements are reduce to only type I ( $x := y$ ). In addition we show the transformer for computational assignments (type C).

### 4.1.1 Identity Statements

$$\begin{aligned}
\llbracket l : x := y \rrbracket(\sigma^A) &= \sigma^A[t \mapsto \sigma^A(t) | t \neq x] \cup & (4.1.1) \\
&\{x \mapsto l\} & (4.1.2)
\end{aligned} \tag{4.1}$$

**Term (4.1.1)** copies all the reaching definitions from  $\sigma^A$  except for those on  $x$ . For instance, consider the example in Fig. 4.1, the application of the transformer of  $l_2$  on  $\sigma_2^A$  removes the existing reaching definitions on  $s\_f1 \mapsto l_1$ .

**Term (4.1.2)** generates a new reaching definition for  $x$ . In Fig. 4.1, the application of the transformer of  $l_3$  on  $dep_3$  adds the reaching definitions  $a\_f1 \mapsto l_3$ .

### 4.1.2 Computational Statements

$$\begin{aligned}
\llbracket l : x := \text{exp}(y) \rrbracket(\sigma^A) &= \sigma^A[t \mapsto \sigma^A(t) | t \neq x] \cup & (4.2.1) \\
&\{x \mapsto l\} & (4.2.2)
\end{aligned} \tag{4.2}$$

**Term (4.2.1)** copies all the reaching definitions from  $\sigma^A$  except for those on  $x$  (the same as for type I). For instance, consider the example in Fig. 4.2, the application of the transformer of  $l_2$  on  $\sigma_2^A$  removes the existing reaching definitions on  $s\_f1 \mapsto l_1$ .

$$\begin{array}{l|l}
l_1: s\_f1 := \text{exp\_f1} & \sigma_1^A = \emptyset \\
l_2: s\_f1 := \text{inc\_1}(s\_f1) & \sigma_2^A = \{s\_f1 \mapsto l_1\} \\
l_3: a\_f1 := \text{dec\_1}(s\_f1) & \sigma_3^A = \{s\_f1 \mapsto l_2\} \\
& \sigma_4^A = \{s\_f1 \mapsto l_2, a\_f1 \mapsto l_3\}
\end{array}$$

Figure 4.2: Example of the process of the ATOM static analysis on statements of type C.

**Term (4.2.2)** generates a new reaching definition for  $x$  (the same as for type I). In Fig. 4.2, the application of the transformer of  $l_3$  on  $dep_3$  adds the reaching definitions  $a\_f1 \mapsto l_3$ .

### 4.1.3 Computing the Dependences

Given a path  $p$  with statement  $\{st_1; st_2, \dots, st_n\}$  and abstract state  $\sigma^A$  computed for this path, we define the function  $Dep^A : \langle l_s, seed, l_t, target \rangle \rightarrow Boolean$ . The function returns true iff the statement in  $l_t$  is (transitively) field sensitive dependent on the statement in  $l_s$ . Note that this implies that  $def(l_s) = seed$  and  $use(l_t) = target$  since  $Prog^A$  contains only primitive variables.

#### Definition 4.1.1

$$Dep^A(l_s, s, l_t, t) = \begin{cases} true & (l_s = \sigma_{l_t}^A(t)) \\ \vee & \\ (\exists l_i : l_i = \sigma_{l_t}^A(t) \wedge Dep^A(l_1, s, l_i, use(l_i))) & (4.3.2) \\ false & otherwise \end{cases} \quad (4.3)$$

The function  $Dep^A$  returns true in two cases: (i) there is an immediate dependency between  $l_t$  and  $l_s$  (ii) there is a transitive dependency, and therefore there is an immediate dependency between  $l_t$  and  $l_i$  and  $l_i$  is dependent on  $l_s$ . Essentially, this definition of dependences corresponds to a transitive path on the atomized program PDG.

**Example 4.1.1** Consider the program shown in Fig. 4.3. This is the atomized representation of the program shown in Fig. 3.10. The query  $Dep^A(l_1, s\_f1, l_3, a\_f1)$  returns true because according to case (4.3.2):

$$\sigma_3^A(a\_f1) = l_{2.1} \text{ and } Dep^A(l_1, s\_f1, l_{2.1}, s\_f1) = true$$

$Dep^A(l_1, s\_f1, l_{2.1}, s\_f1)$  returns true because according to case (4.3.1):

$$\sigma_{2.1}^A(s\_f1) = l_1$$

Thus we conclude that  $a\_f1$  in  $l_3$  is (transitive) field sensitive dependent on  $s\_f1$  in  $l_1$ .

In a similar way we can show that the query  $Dep^A(l_1, s\_f1, l_4, a\_f2)$  returns false, indicating correctly that  $a\_f2$  in  $l_4$  is **not** (transitive) field sensitive dependent on  $s\_f1$  in  $l_1$ .

**Lemma 4.1.2** The abstract definition of  $Dep^A$  on an atomized program is complete in relation to the vanilla static analysis definition in Section 3.2.

$l_1: s\_f1 := t$	$\sigma_1^A = \emptyset$
$l_{2.1}: a\_f1 := s\_f1$	$\sigma_{2.1}^A = \{s\_f1 \mapsto l_1\}$
$l_{2.2}: a\_f2 := s\_f2$	$\sigma_{2.2}^A = \{s\_f1 \mapsto l_1, a\_f1 \mapsto l_{2.1}\}$
$l_{2.3}: a\_f3 := s\_f3$	$\sigma_{2.3}^A = \{s\_f1 \mapsto l_1, a\_f1 \mapsto l_{2.1}, a\_f2 \mapsto l_{2.2}\}$
$l_3: x := a\_f1$	$\sigma_3^A = \{s\_f1 \mapsto l_1, a\_f1 \mapsto l_{2.1}, a\_f2 \mapsto l_{2.2}, a\_f3 \mapsto l_{2.3}\}$
$l_4: y := a\_f2$	$\sigma_4^A = \{s\_f1 \mapsto l_1, a\_f1 \mapsto l_{2.1}, a\_f2 \mapsto l_{2.2}, a\_f3 \mapsto l_{2.3}, x \mapsto l_3\}$
	$\sigma_5^A = \{s\_f1 \mapsto l_1, a\_f1 \mapsto l_{2.1}, a\_f2 \mapsto l_{2.2}, a\_f3 \mapsto l_{2.3}, x \mapsto l_3, y \mapsto l_4\}$

Figure 4.3: Example of the  $Dep^A$  computation using the ATOM static analysis.

*Proof:*

Given the following:

$p \in CFG$ of $Prog$	A path in the original program's CFG
$p^A \in CFG$ of $Prog^A$	The corresponding path to $p$ on $Prog^A$ 's CFG
$l_s, l_t \in p$	Seed statement and target statement in $Prog$
$l_s^A, l_t^A \in p^A$	Seed statement and target statement in $Prog^A$
$def(l_s) \supseteq s.\beta$	Statement $l_s$ assigns to final access-path $s.\beta$
$use(l_t) \supseteq t.\alpha$	Statement $l_t$ uses final access-path $t.\alpha$
$l_s^A \in Trans(l_s), def(l_s^A) = s.\beta$	Statement $l_s^A$ is the transformation of $l_s$ and assigns to $s.\beta$
$l_t^A \in Trans(l_t), use(l_t^A) = t.\alpha$	Statement $l_t^A$ is the transformation of $l_t$ and uses $t.\alpha$
$\sigma_i$	Abstract state arising by the vanilla static analysis on path $p$ before statement $i$
$\sigma_i^A$	Abstract state arising by the atom based static analysis on path $p^A$ before statement $i$

we show that

$$Dep(l_s, s.\beta, l_t, t.\alpha) \Leftrightarrow Dep^A(l_s^A, s.\beta, l_t^A, t.\alpha)$$

thus we need to show, according to the definition of  $Dep$  in 3.10 and the definition of  $Dep^A$  in 4.3 that

$$\langle t.\alpha, s.\beta, l_s, k \rangle \in \sigma_{l_t} \Leftrightarrow (l_s^A = \sigma_{l_t}^A(t.\alpha)) \vee (\exists l_i : l_i^A = \sigma_{l_t}^A(t.\alpha) \wedge Dep^A(l_s^A, s.\beta, l_i^A, use(l_i^A)))$$

Note that  $\sigma_{l_t}$  is the abstract state arising before the handling of  $l_t$ , and similar for  $\sigma_{l_t}^A$ . We show this by induction on  $k$  the number of immediate value flow dependences that compose a transitive dependence.

$k = 0$ . Dependences with  $k=0$  are added to the dependences only for assignments to some access path  $ap$  and always as  $\langle ap, ap, l, 0 \rangle$ . Therefore:

$$t.\alpha = s.\beta$$

implying that the statements in this case are (without lost of generality):

$$\begin{array}{ll} l_s : s.\beta' = exp & l_s^A : s.\beta = exp^A \\ l_t : exp' = s.\beta'' & l_t^A : exp'^A = s.\beta \end{array}$$

Note that  $s.\beta' \supseteq s.\beta$  and  $s.\beta'' \supseteq s.\beta$ .

Now we need to show that this is true iff

$$l_s^A = \sigma_{l_t}^A(s\_ \beta)$$

Lets assume on the contrary that this does not hold, i.e.:

$$l_i^A = \sigma_{l_t}^A(s\_ \beta) \wedge l_i^A \neq l_s^A$$

For Statement  $l_i^A$  to be the reaching definition it must have the following properties  $l_i^A \in p^A$  and  $def(l_i^A) = s\_ \beta$ . But, since  $p^A = trans(p)$  there must be  $l_i \in p$  such that  $trans(l_i) \supseteq l_i^A$  and  $def(l_i) \supseteq s.\beta$ . Clearly, from the definition of the transformation and the definition of the vanilla algorithm this is a contradiction to

$$\langle s.\beta, s.\beta, l_s, 0 \rangle \in \sigma_{l_t}$$

**k = n-1.** We assume that for every dependency that composed of n-1 immediate value flow dependences:

$$Dep(l_s, s.\beta, l_t, t.\alpha) \Leftrightarrow Dep^A(l_s^A, s\_ \beta, l_t^A, t\_ \alpha)$$

**k = n.** we need to show that

$$Dep(l_s, s.\beta, l_t, t.\alpha) \Leftrightarrow Dep^A(l_s^A, s\_ \beta, l_t^A, t\_ \alpha)$$

we will show that for  $k > 0$

$$\langle t.\alpha, s.\beta, l_s, k \rangle \in \sigma_{l_t} \Leftrightarrow \exists l_i^A : l_i^A = \sigma_{l_t}^A(t\_ \alpha) \wedge Dep^A(l_s^A, s\_ \beta, l_i^A, use(l_i^A))$$

For such a dependency  $\langle t.\alpha, s.\beta, l_s, k \rangle$  to arise there must be a dependency  $\langle x.\gamma, s.\beta, l_s, k - 1 \rangle \in \sigma_{l_i}$  during the processing of statement  $l_i$  where

$$l_i : t.\alpha' = x.\gamma'$$

and  $t.\alpha' \supseteq t.\alpha$  and  $x.\gamma' \supseteq x.\gamma$  Thus from the definition of the transformation there must exists  $l_i^A$  that is part of the transformation of  $l_i$  and  $l_i^A : t\_ \alpha = x.\gamma$ . From the assumption of the induction we can assume that

$$Dep(l_s, s.\beta, l_i, x.\gamma) \Leftrightarrow Dep^A(l_s^A, s\_ \beta, l_i^A, x.\gamma)$$

In addition, similar to the case of  $k = 0$  we can show that  $l_i^A = \sigma_{l_t}^A(t\_ \alpha)$  Therefore,

$$\langle t.\alpha, s.\beta, l_s, k \rangle \in \sigma_{l_t} \Leftrightarrow \exists l_i^A : l_i^A = \sigma_{l_t}^A(t\_ \alpha) \wedge Dep^A(l_s^A, s\_ \beta, l_i^A, use(l_i^A))$$

■

### 4.1.4 Complexity

The ATOM algorithm stores at each statement a state representing all the reaching definitions that arise along the path. For each statement we store, in the worst case, a mapping from all the (atomized) primitive access path. We denote the size of this group of access paths as  $O(|\mathcal{AP}_{prim}|)$ .

We denote the size of the maximal set of primitive field path of a single type as  $|\mathcal{FP}_{max}|$ . Surely,

$$|\mathcal{FP}_{max}| < |\mathcal{AP}_{prim}|$$

The length of the path  $p^A$  is bounded in the size of the original path  $|p|$  times the maximal atomization:  $O(|p| \times |\mathcal{FP}_{max}|)$  since each statement is transferred to at most  $|\mathcal{FP}_{max}|$  statements.

Thus, the space complexity is:

$$C_{space}^A = O(|p| \times |\mathcal{FP}_{max}| \times |\mathcal{AP}_{prim}|)$$

The complexity of computing the dependence is comprised of two parts: (i) the computation of state  $\sigma^A$  for each statement on the path  $p^A$  (pre-process) (ii) inferring the field sensitive dependences using the definition of  $Dep^A$  (query).

When computing the abstract states, each statement is processed once. For each statement, we scan the previous state. Thus, the time complexity of the pre-process:

$$C_{pre}^A = O(|p| \times |\mathcal{FP}_{max}| \times |\mathcal{AP}_{prim}|)$$

When computing the  $Dep^A$  function, a transitive closure is performed which is limited by the size of the transformed program path times the lookup time in an abstract state. Thus, The time complexity of the query:

$$C_q^A = O(|p| \times |\mathcal{FP}_{max}| \times |\mathcal{AP}_{prim}|)$$

# Chapter 5

## Interval Based Field Sensitive Dependences

In this chapter we define an interval based static analysis algorithm that computes the field sensitive dependences.

For each program label on CFG path  $p$  we define an abstract state  $\sigma^I \in \Sigma^I = \mathcal{V} \times INTERVAL \rightarrow LABEL$  where  $INTERVAL = \mathbb{Z} \times \mathbb{Z}$ . This state represents all possible reaching-definitions that may arise at execution of this specific path. The interval based field path  $\langle start, end \rangle$  corresponds to the memory interval according to the structure type. Essentially,  $\sigma^I$  maps a memory interval to the statement that defines this interval.

The abstract transformer of a statement  $st$  is a function  $\llbracket st \rrbracket : \Sigma^I \rightarrow \Sigma^I$ . Essentially, the effect of a statement  $st$  on an abstract state  $\sigma^I$  is to remove some interval based reaching definitions that are no longer true and to generate new ones. The initial state is the empty state  $\sigma_0^I = \emptyset$  and for a CFG path with statements  $\{st_0; st_1; \dots; st_n; \}$  we denote

$$\sigma_{i+1}^I = \llbracket l_i : st_i \rrbracket(\sigma_i^I)$$

Throughout this chapter we use the interval representation of the structure types defined in Fig. 3.1. Table 5.1 contains the offset details for these types.

### 5.1 Transfer Function

Next, we describe the transformers for the copy assignment statements. For convenience, we use the utility function  $start(ap)$  and  $end(ap)$  to represent the start-offset and the end-offset of an access path  $ap$ . All the offsets are computed according to the access path type  $T(ap)$ . In a similar way we define  $start()$  and  $end()$  for field paths.

#### 5.1.1 Identity Statements

$$\llbracket l : x := y \rrbracket(\sigma^I) = \sigma^I[\langle t, i, j \rangle \mapsto \sigma^I(\langle t, i, j \rangle) | t \neq x] \cup \{\langle x, 0, end(x) \rangle \mapsto l\} \quad (5.1)$$

**Term (5.1.1)** copies all the reaching definitions from  $\sigma^I$  except for those on some interval of  $x$ . Reaching definitions on any interval of  $x$  are "killed". For instance, consider the example

Type/Field	Start offset	End Offset
T1	0	12
f1	0	4
f2	4	8
f3	8	12
T2	0	20
g1	0	4
g2	4	8
g3	8	20
g3.f1	8	12
g3.f2	12	16
g3.f3	16	20

Table 5.1: Interval representation for the example type defined in Fig. 3.1.

$$\begin{array}{l|l}
l_1: s := \text{exp1} & \sigma_1^I = \emptyset \\
l_2: s := \text{exp2} & \sigma_2^I = \{\langle s, 0, 12 \rangle \mapsto l_1\} \\
l_3: a := s & \sigma_3^I = \{\langle s, 0, 12 \rangle \mapsto l_2\} \\
& \sigma_4^I = \{\langle s, 0, 12 \rangle \mapsto l_2, \langle a, 0, 12 \rangle \mapsto l_3\}
\end{array}$$

Figure 5.1: Example of the process of the FSD analysis on statements of type I.

in Fig. 5.1, the application of the transformer of  $l_2$  on  $\sigma_2^I$  removes the existing reaching definition  $\langle s, 0, 12 \rangle \mapsto l_1$ .

**Term (5.1.2)** generates a new reaching definition for  $x$ . For instance, consider the example in Fig. 5.1, the application of the transformer of  $l_3$  on  $\sigma_3^I$  adds the reaching definition  $\langle a, 0, 12 \rangle \mapsto l_3$ .

## 5.1.2 Write Statements

Handling write statement requires some finesse as the definition of  $x.f$  partially kills a prior definition to  $x$ . All the other assignments define the whole structure and thus do not create a "partial kill" definition<sup>1</sup>. Consider the example in Fig. 5.2. The statement  $l_1$  defines the whole structure  $x$  and the statement in  $l_2$  defines only  $x.f2$ . This definition partially kill the previous definitions. Therefore we would like to keep the previous reaching definition for the part of the structure that were not killed and to generate the new reaching definition only to the part that was killed. In order to resolve this issue, we remove the previous reaching definition interval of whole  $x$  and insert three new intervals. The first part pertains to the interval from the start of  $x$  till the start of  $x.f$ , the second is the interval of  $x.f$  and the third pertains to the interval from the end of  $x.f$  till the end of  $x$ .

<sup>1</sup>As we assume that all assignments are type-safe. In Chapter 6 we show that handling weakly type language requires only minor adjustment to the definition of  $\sigma^I$

$$\begin{array}{l|l}
l_1: s := \text{expl} & \sigma_1^I = \emptyset \\
l_2: s.f2 := \text{kill} & \sigma_2^I = \{\langle s, 0, 12 \rangle \mapsto l_1\} \\
l_3: b.g3 := s & \sigma_3^I = \{\langle s, 0, 4 \rangle \mapsto l_1, \langle s, 8, 12 \rangle \mapsto l_1, \langle s, 4, 8 \rangle \mapsto l_2\} \\
& \sigma_4^I = \{\langle s, 0, 4 \rangle \mapsto l_1, \langle s, 8, 12 \rangle \mapsto l_1, \langle s, 4, 8 \rangle \mapsto l_2, \langle h, 0, 12 \rangle \mapsto l_3\}
\end{array}$$

Figure 5.2: Example of the process of the FSD analysis on statements of type W.

$$\sigma^I[\langle t, i, j \rangle \mapsto \sigma^I(\langle t, i, j \rangle) | t \neq x \vee \text{start}(f) > j \vee \text{end}(f) < i] \cup \quad (5.2.1)$$

$$\{\langle x, \text{start}(f), \text{end}(f) \rangle \mapsto l\} \cup \quad (5.2.2)$$

$$\llbracket l : x.f := y \rrbracket(\sigma^I) = \{\langle x, a, \text{start}(f) \rangle \mapsto l' | \forall a, b : \sigma^I(\langle x, a, b \rangle) \mapsto l' \wedge \text{start}(f) > a \wedge \text{end}(f) < b\} \cup \quad (5.2.3)$$

$$\{\langle x, \text{end}(f), b \rangle \mapsto l' | \forall a, b : \sigma^I(\langle x, a, b \rangle) \mapsto l' \wedge \text{start}(f) > a \wedge \text{end}(f) < b\} \quad (5.2.4)$$

$$(5.2)$$

**Term (5.2.1)** copies all the reaching definitions from  $\sigma^I$  except for those on some interval that is contained in  $\widehat{x.f}$  or containing  $\widehat{x.f}$ . Reaching definitions on any interval that is contained in  $\widehat{x.f}$  or containing  $\widehat{x.f}$  are "killed". For instance, consider the example in Fig. 5.2, the application of the transformer of  $l_2$  on  $\sigma_2^I$  removes the existing reaching definition  $\langle s, 0, 12 \rangle \mapsto l_1$  as this interval contains the interval that is defined in  $l_2$ :  $\langle s, 4, 8 \rangle$ .

**Term (5.2.2)** generates a new reaching definition for the interval of  $x.f$ . For instance, consider the example in Fig. 5.2, the application of the transformer of  $l_2$  on  $\sigma_2^I$  adds the reaching definitions  $\langle s, 4, 8 \rangle \mapsto l_2$ .

**Term (5.2.3)** adjusts the prefix of an existing reaching definition on an interval that contains the interval of  $x.f$ . For instance, consider the example in Fig. 5.2, the application of the transformer of  $l_2$  on  $\sigma_2^I$  adds the reaching definitions  $\langle s, 0, 4 \rangle \mapsto l_1$  that corresponds to the prefix of the existing reaching definition on the containing interval  $\langle s, 0, 12 \rangle$ . Note that the reaching definition on  $\langle s, 0, 12 \rangle$  was "killed" via term (5.2.1).

**Term (5.2.4)** adjusts the suffix of an existing reaching definition on an interval that contains the interval of  $x.f$ . For instance, consider the example in Fig. 5.2, the application of the transformer of  $l_2$  on  $\sigma_2^I$  adds the reaching definitions  $\langle s, 8, 12 \rangle \mapsto l_1$  that corresponds to the suffix of the existing reaching definition on the containing interval  $\langle s, 0, 12 \rangle$ . Note that the reaching definition on  $\langle s, 0, 12 \rangle$  was "killed" via term (5.2.1).

### 5.1.3 Read Statements

$$\llbracket l : x := y.f \rrbracket(\sigma^I) = \sigma^I[\langle t, i, j \rangle \mapsto \sigma^I(\langle t, i, j \rangle) | t \neq x] \cup \quad (5.3.1)$$

$$\{\langle x, 0, \text{end}(x) \rangle \mapsto l\} \quad (5.3.2) \quad (5.3)$$

**Term (5.3.1)** copies all the reaching definitions from  $\sigma^I$  except for those on some interval of  $x$  (similar to type I). For instance, consider the example in Fig. 5.3, the application of the transformer of  $l_2$  on  $\sigma_2^I$  removes the existing reaching definitions  $\langle s, 0, 12 \rangle \mapsto l_1$ .

$$\begin{array}{l|l}
l_1: s := \text{exp1} & \sigma_1^I = \emptyset \\
l_2: s := \text{b.g3} & \sigma_2^I = \{\langle s, 0, 12 \rangle \mapsto l_1\} \\
l_3: t := \text{s.f3} & \sigma_3^I = \{\langle s, 0, 12 \rangle \mapsto l_2\} \\
& \sigma_4^I = \{\langle s, 0, 12 \rangle \mapsto l_2, \langle t, 0, 4 \rangle \mapsto l_3\}
\end{array}$$

Figure 5.3: Example of the process of the FSD analysis on statements of type R.

$$\begin{array}{l|l}
l_1: s := \text{exp1} & \sigma_1^I = \emptyset \\
l_2: s := \text{inc}(s) & \sigma_2^I = \{\langle s, 0, 12 \rangle \mapsto l_1\} \\
l_3: a := \text{dec}(s) & \sigma_3^I = \{\langle s, 0, 12 \rangle \mapsto l_2\} \\
& \sigma_4^I = \{\langle s, 0, 12 \rangle \mapsto l_2, \langle a, 0, 12 \rangle \mapsto l_3\}
\end{array}$$

Figure 5.4: Example of the process of the FSD analysis on statements of type C.

**Term (5.3.2)** generates a new reaching definition for  $x$  (similar to type I). For instance, consider the example in Fig. 5.3, the application of the transformer of  $l_3$  on  $dep_3$  adds the reaching definitions  $\langle t, 0, 4 \rangle \mapsto l_3$ .

### 5.1.4 Computational Statements

$$\llbracket l : x := \text{exp}(y_1, y_2, \dots, y_n) \rrbracket(\sigma^I) = \sigma^I[\langle t, i, j \rangle \mapsto \sigma^I(\langle t, i, j \rangle) \mid t \neq x] \cup \{\langle x, 0, \text{end}(x) \rangle \mapsto l\} \quad (5.4)$$

**Term (5.3.1)** copies all the reaching definitions from  $\sigma^I$  except for those on some interval of  $x$  (similar to type I). For instance, consider the example in Fig. 5.4, the application of the transformer of  $l_2$  on  $\sigma_2^I$  removes all the existing reaching definitions on  $s$  that are related to  $l_1$ .

**Term (5.3.2)** generates a new reaching definition for  $x$  (similar to type I). For instance, consider the example in Fig. 5.4, the application of the transformer of  $l_3$  on  $dep_3$  adds the reaching definitions  $\langle a, 0, 12 \rangle \mapsto l_3$ .

## 5.2 Computing The Dependences

This section describes how the field sensitive dependences are inferred using the abstract state  $\sigma^I$  containing the interval based reaching definition. Similarly to the atomization approach the abstract state contains immediate dependences and a transitive function is needed to compute transitive flow dependences. However, unlike the atomization approach where the transitive relation is simple, for the interval-based analysis we need to handle cases where the reaching definition is on a Big L-value. Consider the example of Fig. 5.5, the reaching definition of  $a$  at  $l_3$  and  $l_4$  is on the whole structure. However, when computing the transitive flow dependences from statement  $l_3$  and from statement  $l_4$  to  $l_1$ , the additional reaching definition of  $s.f1$  is taken into account. Clearly, the transitivity of these two reaching definition is valid for  $l_3$  but invalid for  $l_4$ . As we shall see, while tracking the transitive dependence, we keep track of the current

$ival_1$	$ival_2$	Add	Sub	Symbolic
$\langle a, 4, 8 \rangle$	$\langle a, 0, 12 \rangle$		$\langle a, 4, 8 \rangle$	$\widehat{a.f2} - \widehat{a}$
$\langle b, 12, 16 \rangle$	$\langle b, 8, 20 \rangle$		$\langle b, 4, 8 \rangle$	$\widehat{b.g3.f2} - \widehat{b.g3}$
$\langle b, 8, 20 \rangle$	$\langle b, 12, 16 \rangle$		$\langle b, 8, 20 \rangle$	$\widehat{b.g3} - \widehat{b.g3.f2}$ (forward case)
$\langle a, 0, 12 \rangle$	$\langle s, 4, 8 \rangle$	$\langle a, 4, 8 \rangle$		$\widehat{a} + \widehat{s.f2}$
$\langle b, 8, 20 \rangle$	$\langle a, 8, 12 \rangle$	$\langle x, 16, 20 \rangle$		$\widehat{b.g3} + \widehat{a.f3}$
$\langle x, 8, 20 \rangle$	$\langle a, 0, 12 \rangle$	$\langle x, 8, 20 \rangle$		$\widehat{b.g3} + \widehat{a}$

Table 5.2: Examples to the interval basic operations: Add, Sub.

field-path interval that is of interest. Thus, while computing the transitive dependence of  $l_3$  we keep track of  $\widehat{f1}$  as the interesting interval and therefore allow the transitivity with the reaching definition of  $s.f1$ . In addition, since we track  $\widehat{f2}$  as an interval of interest for the transitive dependence of  $l_4$  we filter the reaching definition of  $s.f1$  and avoid this superfluous dependence. This is further complicated when nested access-paths are taken into account.

For this task, two operations are defined on intervals. A subtraction operation computes the part of one interval within another one. For example, the result of  $\widehat{b.g3.f2} - \widehat{b.g3}$  is  $\widehat{f2}$ , the interval of field  $f2$  within the type  $T1$ , the type of  $g3$ . An addition operation computes the inner interval of an interval within another interval, e.g.  $\widehat{b.g3} + \widehat{f3}$  is  $\widehat{x.g3.f3}$ , the interval of  $f3$  within the variable  $b$ . In the rest of this section, we formally define addition and subtraction operations and utilize them in computation of transitive dependences.

### 5.2.1 Basic Interval Operations

We define basic interval operations for *subtracting* intervals ( $Sub()$ ) in order to compute one interval in relation to the other and for *adding* intervals ( $Add()$ ) in order to refine the first interval with an interval "delta". Formally we define the operation as follows:

$$Sub(\langle x, s_1, e_1 \rangle, \langle x, s_2, e_2 \rangle) = \begin{cases} \langle x, s_1 - s_2, (s_1 - s_2) + (e_1 - s_1) \rangle & s_1 \geq s_2 \\ \langle x, 0, (e_1 - s_1) \rangle & s_1 < s_2 \end{cases}$$

The function  $Sub$  constructs a new field interval that represents the first field interval relatively to the second. The start offset is the subtraction of the two and the end offset is defined according to the length of the first.

$$Add(\langle y, s_1, e_1 \rangle, \langle x, s_2, e_2 \rangle) = \langle y, s_1 + s_2, (s_1 + s_2) + (e_2 - s_2) \rangle$$

The function  $Add$  takes the first field interval and adds the "delta" field path. The start offset is the sum of the two and the end offset is defined according to the length of the delta.

We supply Table 5.2 in order to further explain these interval operation via examples. Column 1 and 2 contain the operation parameters, column 3 contains the result in case of  $Add$  and column 4 contains the result in case of  $Sub$ . Column 5 contains a symbolic example ("access-path wise") to the specific operation.

We define the  $Dep^I$  function for the interval based analysis. Note that  $\sigma^I(\langle x, i, j \rangle)$  returns the label that defines this interval ( $\langle x, i, j \rangle$ ) or defines an interval that contains the query interval. This can be done efficiently by using a data structure that supports interval manipulation like *interval tree* [CLRS01].

Essentially,  $Dep^I(l_s, \widehat{s.\beta}, l_n, \widehat{t.\alpha})$  returns true iff  $t.\alpha$  in  $l_n$  is field sensitive dependent on  $s.\beta$  in  $l_s$ .

### Definition 5.2.1

$$Dep^I(l_s, \widehat{s.\beta}, l_t, \widehat{t.\alpha}) = \begin{cases} true & ((l_s = \sigma_{l_t}^I(\widehat{t.\alpha}) \wedge \\ & \widehat{s.\beta} = \widehat{t.\alpha}) \quad (5.5.1) \\ \vee \\ & (\exists l_i : l_i = \sigma_{l_t}^I(\widehat{t.\alpha}) \wedge \\ & Dep^I(l_s, \widehat{s.\beta}, l_i, Add(\widehat{use}(l_i), Sub(\widehat{t.\alpha}, \widehat{def}(l_i)))) \quad (5.5.2) \\ false & otherwise \end{cases} \quad (5.5)$$

The function  $Dep^I$  returns true in two cases: (i) there is an immediate dependency between  $l_t$  and  $l_s$  (ii) there is a transitive dependency, and therefore there is an immediate dependency between  $l_t$  and  $l_i$  and  $l_i$  is dependent on  $l_s$ .

The query access-path calculation in (5.5.2) is executed in two steps: first we compute the query interval  $\widehat{t.\alpha}$  (refinement of the use of  $l_n$ ) relatively to the def  $\widehat{def}(l_i)$  on  $l_i$ . Notice that  $\widehat{def}(l_i) \supseteq \widehat{t.\alpha}$  according to the definition of  $\sigma^I$  (the reaching definition can be interval that is equal or containing the use interval). Then we add this offset field "delta" to the use of  $l_i$   $\widehat{use}(l_i)$ . The result will be the query interval for the recursive call to  $Dep^I$ .

**Example 5.2.1** Consider the program shown in Fig. 5.5. The query  $Dep^I(l_1, \widehat{s.f1}, l_3, \widehat{a.f1})$  returns true because according to case (5.5.2):

$$\sigma_3^I(\widehat{a.f1}) = l_2 \text{ and } Dep^I(l_1, \widehat{s.f1}, l_2, \widehat{s.f1}) = true$$

$Dep^I(l_1, \widehat{s.f1}, l_2, \widehat{s.f1})$  equals true according to case (5.5.1):

$$\sigma_2^I(\widehat{s.f1}) = l_1 \text{ and } \widehat{s.f1} = \widehat{def}(l_1)$$

Thus, we can conclude that  $a.f1$  in  $l_3$  is (transitive) field sensitive dependent on  $s.f1$  in  $l_1$ .

In a similar way we can show that the query  $Dep(l_1, \widehat{s.f1}, l_4, \widehat{a.f2})$  returns false, indicating correctly that  $a.f2$  in  $l_4$  is **not** (transitive) field sensitive dependent on  $s.f1$  in  $l_1$ .

**Lemma 5.2.2** The abstract definition of  $Dep^I$  is complete in relation to the vanilla static analysis definition in Section 3.2.

$$\begin{array}{l|l}
l_1: s.f1 := t & \sigma_1^I = \emptyset \\
l_2: a := s & \sigma_2^I = \{\langle s, 0, 4 \rangle \mapsto l_1\} \\
l_3: x := a.f1 & \sigma_3^I = \{\langle s, 0, 4 \rangle \mapsto l_1, \langle a, 0, 12 \rangle \mapsto l_2\} \\
l_4: y := a.f2 & \sigma_4^I = \{\langle s, 0, 4 \rangle \mapsto l_1, \langle a, 0, 12 \rangle \mapsto l_2, \langle x, 0, 4 \rangle \mapsto l_3\} \\
& \sigma_5^I = \{\langle s, 0, 4 \rangle \mapsto l_1, \langle a, 0, 12 \rangle \mapsto l_2, \langle x, 0, 4 \rangle \mapsto l_3, \langle y, 0, 4 \rangle \mapsto l_4\}
\end{array}$$

Figure 5.5: Example of the  $Dep^I$  computation using the FSD analysis.

*Proof:*

Given the following:

$p \in CFG$ of $Prog$	A path in the original program's CFG
$l_s, l_t \in p$	Seed statement and target statement in $Prog$
$def(l_s) \supseteq s.\beta$	Statement $l_s$ assigns to final access-path $s.\beta$
$use(l_t) \supseteq t.\alpha$	Statement $l_t$ uses final access-path $t.\alpha$
$\sigma_i$	Abstract state arising by the vanilla static analysis on path $p$ before statement $i$
$\sigma_i^I$	Abstract state arising by the interval based static analysis on path $p$ before statement $i$

we show that

$$Dep(l_s, s.\beta, l_t, t.\alpha) \Leftrightarrow Dep^I(l_s, \widehat{s.\beta}, l_t, \widehat{t.\alpha})$$

thus we need to show, according to the definition of  $Dep$  in 3.10 and the definition of  $Dep^I$  in 5.5 that

$$\langle t.\alpha, s.\beta, l_s, k \rangle \in \sigma_{l_t} \Leftrightarrow (l_s = \sigma_{l_t}^I(\widehat{t.\alpha}) \wedge \widehat{s.\beta} = \widehat{t.\alpha}) \vee (\exists l_i : l_i = \sigma_{l_t}^I(\widehat{t.\alpha}) \wedge Dep^I(l_s, \widehat{s.\beta}, l_i, Add(\widehat{use(l_i)}, Sub(\widehat{t.\alpha}, \widehat{def(l_i)}))))$$

Note that  $\sigma_{l_t}$  is the abstract state arising before the handling of  $l_t$ , and similar for  $\sigma_{l_t}^I$ . We show this by induction on  $k$  the number of immediate value flow dependences that compose a transitive dependence.

$k = 0$ . Dependences with  $k=0$  are added to the dependences only for assignments to some access path  $ap$  and always as  $\langle ap, ap, l, 0 \rangle$ . Therefore:

$$t.\alpha = s.\beta \Leftrightarrow \widehat{t.\alpha} = \widehat{s.\beta}$$

implying that the statements in this case are (without lost of generality):

$$\begin{array}{l}
l_s : s.\beta' = exp \\
l_t : exp' = s.\beta''
\end{array}$$

Note that  $s.\beta' \supseteq s.\beta$  and  $s.\beta'' \supseteq s.\beta$ .

Now we need to show that this is true iff

$$l_s = \sigma_{l_t}^I(\widehat{s.\beta})$$

Lets assume on the contrary that this does not hold, i.e.:

$$l_i = \sigma_{l_t}^I(\widehat{s.\beta}) \wedge l_i \neq l_s$$

For statement  $l_i$  to be in the reaching definition it must have the following properties  $l_i \in p$  and  $def(l_i) = s.\beta'''$  where  $def(l_i) \supseteq s.\beta$ . Thus according to the definition of the vanilla algorithm exists

$$\langle s.\beta, s.\beta, l_i, 0 \rangle \in \sigma_{l_t}$$

Clearly, this is a contradiction to

$$\langle s.\beta, s.\beta, l_s, 0 \rangle \in \sigma_{l_t}$$

**k = n-1.** We assume that for every dependency that composed of n-1 immediate value flow dependences:

$$Dep(l_s, s.\beta, l_t, t.\alpha) \Leftrightarrow Dep^I(l_s, \widehat{s.\beta}, l_t, \widehat{t.\alpha})$$

**k = n.** we need to show that

$$Dep(l_s, s.\beta, l_t, t.\alpha) \Leftrightarrow Dep^I(l_s, \widehat{s.\beta}, l_t, \widehat{t.\alpha})$$

we will show that for  $k > 0$

$$\langle t.\alpha, s.\beta, l_s, k \rangle \in \sigma_{l_t} \Leftrightarrow \exists l_i : l_i = \sigma_{l_t}^I(\widehat{t.\alpha}) \wedge Dep^I(l_s, \widehat{s.\beta}, l_i, Add(\widehat{use}(l_i), Sub(\widehat{t.\alpha}, \widehat{def}(l_i))))$$

For such a dependency  $\langle t.\alpha, s.\beta, l_s, k \rangle$  to arise there must be a dependency  $\langle x.\gamma'.\alpha'', s.\beta, l_s, k-1 \rangle \in \sigma_{l_i}$  during the processing of statement  $l_i$  where

$$l_i : t.\alpha' = x.\gamma'$$

and  $t.\alpha' \supseteq t.\alpha$  and  $x.\gamma' \supseteq x.\gamma'.\alpha''$ . In addition,  $t.\alpha = t.\alpha'.\alpha''$ .

Thus from the definition of  $Add()$  and  $Sub()$  exists

$$Add(\widehat{use}(l_i), Sub(\widehat{t.\alpha}, \widehat{dep}(l_i))) = Add(\widehat{x.\gamma'}, Sub(\widehat{t.\alpha}, \widehat{t.\alpha'})) = \widehat{x.\gamma'.\alpha''}$$

From the assumption of the induction we can assume that

$$Dep(l_s, s.\beta, l_i, x.\gamma'.\alpha'') \Leftrightarrow Dep^I(l_s, \widehat{s.\beta}, l_i, \widehat{x.\gamma'.\alpha''})$$

In addition, similar to the case of  $k = 0$  we can show that  $l_i = \sigma_{l_t}^I(\widehat{t.\alpha})$  Therefore,

$$\langle t.\alpha, s.\beta, l_s, k \rangle \in \sigma_{l_t} \Leftrightarrow \exists l_i : l_i = \sigma_{l_t}^I(\widehat{t.\alpha}) \wedge Dep^I(l_s, \widehat{s.\beta}, l_i, Add(\widehat{use}(l_i), Sub(\widehat{t.\alpha}, \widehat{def}(l_i))))$$

■

## 5.2.2 Complexity

The interval based algorithm stores at each statement a state representing all the interval reaching definitions that arise along the path. In the worst case the number of intervals can be as the number of primitive access path. However, we assume that in practice the number of intervals is much lower. We denote *Intervals* as the set of intervals that are actually mapped in some abstract state. The *Interval* set contains at least all the variables in the program (assuming all variables have some reaching definition)

$$|\mathcal{V}_P| \leq |\text{Intervals}| \leq |\mathcal{AP}_{prim}|$$

Thus, the space complexity is:

$$C_{space}^I = O(|p| \times |\text{Intervals}|)$$

The complexity of computing the dependence is comprised of two parts: (i) the computation of state  $\sigma^I$  for each statement on the path  $p$  (pre-process) (ii) inferring the field sensitive dependences using the definition of  $Dep^I$  (query).

The time complexity of the pre-process:

$$C_{pre}^I = O(|p| \times |\text{Intervals}|)$$

The time complexity of the query is bounded by the size of the path times the lookup time in an abstract state:

$$C_q^A = O(|p| \times |\text{Intervals}|)$$

Thus, the interval algorithm improves the space complexity and the time complexity (pre-process and query) in at least a factor of  $O(|\mathcal{FP}_{max}|)$  over the ATOM algorithm. In practice, we can expect a much better improvement as for the programs we analyze

$$|\text{Intervals}| \ll |\mathcal{AP}_{prim}|$$

# Chapter 6

## Analyzing Large Scale Programs

In this chapter we present extensions to our interval based algorithm FSD which enable us to analyze large scale programs. In the previous chapters we described the algorithm in a primitive settings in order to simplify our presentation. We defined it in details while focusing only on the computation of intraprocedural value dependences. We now extend our discussion to include more complex attributes of real-life programs.

The additional challenges of the interval based algorithm include:

**Forward Analysis** Our impact analysis application requires computing the dependences from a certain seed statement to all the dependent statement in the program. As a result, instead of performing a query on a given source statement and target statement (backward analysis over reaching definition data on the CFG) we modify our algorithm to support also a query on a single source statement (forward analysis over def-use edges on the PDG).

**Generalizing Seed Access Path** We generalize our dependence queries to support non-primitive access path. Whereas the backward query requires primitive access path for both the target-use and the source-def, the forward query permits non-primitive access paths for the source-def. This enables to use the actual  $def(st_{seed})$  in the query. In many cases we benefit from tracking the whole def interval as multiple fields of *seed* act in a similar manner. However, this calls for tracking the source-def interval in addition to tracking the target-use interval which enables filtering of superfluous dependences (as in the backward query).

**Control Dependences** We extend the dependence analysis to include computation of control dependences in addition to value dependences. We use the PDG which contains immediate data and control dependency edges and compute the transitive field sensitive dependences.

**Weakly Typed Programming Languages** In this type of languages, assignments between structures of different types is acceptable and in some cases even common. It is crucial that any field sensitive dependence analysis should compute precise dependences despite the present of such assignments.

**Interprocedural Analysis** Handling procedure in an efficient way is one of the key factors in the scalability of the algorithm to large scale programs. Methods like inline are usually not feasible due the significant size of the programs. We adjust known procedure summary algorithms in order maintain the field sensitivity property.

Next, we describe in details how the interval based field sensitive dependence analysis meets these additional challenges. The remainder of this chapter proceeds as follows: Section 6.1 presents forward intraprocedural analysis over a labeled PDG; Section 6.2 tackles weakly typed languages and Section 6.3 describes the interprocedural analysis.

## 6.1 Field Sensitive Dependence Analysis on the PDG

### 6.1.1 Constructing Interval Label PDG

The program dependence graph represents immediate data and control dependences in a procedure. The nodes of the graph are the statements. The edges are the def-use edges, which are based on the reaching definitions, and control flow edges which are based on the CFG. We adjust our algorithm to use the def-use edges instead of the interval based reaching definition that we computed in  $\sigma^I$ .

Using the interval based reaching definition  $\sigma^I$  (for details see Chapter 5) we add labels on the def-use edges that represent the interval that is actually in use. This enables handling of destructive updates (i.e., partial kill assignments) accurately. For example in Fig. 6.1 the two edges from  $l_2$  to  $l_4$  are marked with the intervals of field  $a.f1$  and  $a.f3$  because of statement  $l_3$  that kills  $a.f2$ . In addition the algorithm uses the control dependence edges to infer transitive field sensitive control dependences.

### 6.1.2 The Abstract State

The dependence analysis performs chaotic iterations over the interval labeled PDG. Let  $use(st)$  be the variable used (or which one of its fields is being used) at  $st$ . The analysis allows to conservatively infer the field-sensitive dependency of  $use(st)$  at  $st$  on  $def(st')$  at  $st'$  that arise at any path between  $st'$  and  $st$ .

The analysis computes for every statement  $st$  a set of *field sensitive states*. Every field sensitive state  $s \in S = (\mathcal{V} \times INTERVAL) \times (\mathcal{V} \times INTERVAL) \times \{true, false\}$  represents which part of the used variable at  $st$  is dependent on which part of the variable that is defined at the seed statement. We assume without loss of generality that  $def(st_{seed}) = sd$ . Computing the state  $s = \langle \widehat{t.\alpha}, \widehat{sd.\beta}, vd \rangle$  at  $st$  represents that the interval  $\widehat{t.\alpha}$  of  $use(st)$  may be dependent on the interval  $\widehat{sd.\beta}$  of  $def(st_{seed})$ . The dependency is a value dependency if  $vd$  is true and a computational dependency otherwise. Value dependency means that the value assigned to the interval  $\widehat{\beta}$  of  $sd$ , is read from the interval  $\widehat{\alpha}$  of  $use(st)$ .

Consider the example in Fig. 6.1. The state that is attached to a node  $n$  is a result of the processing of its incoming edges. For the statement in  $l_7$  we conclude that the  $\widehat{c.g2}$  interval (relative to the use  $c$ ) is field sensitive value dependent on the  $\widehat{sd.f1}$  interval (relative to the def  $sd$ ) in line  $l_1$ . For statement  $l_5$  exists the dependence  $\langle \langle b, 0, length(b.f1) \rangle, \widehat{sd.f1}, true \rangle$ , representing that the whole use  $b.f1$  (there is no symbolic representation to this interval so we use the explicit interval:  $\langle b, 0, length(b.f2) \rangle$ ) is value dependent on the  $\widehat{sd.f1}$  interval (relative to the def  $sd$ ) in line  $l_1$ .

We refer to the first interval in the state  $\widehat{t.\alpha}$  as the *target-use interval refinement* as it refines the actual use, and determines which part of it is dependent on the seed statement. As in Chapter 5,

we use this interval to track which part of the use is of interest and to filter out false dependences. We refer to the second interval in the state  $\widehat{sd.\beta}$  as the *source-def interval refinement* as it refines the source def, and determines on which part of the seed exists this dependency. Note, that we need to track this interval because we allow non-primitive access path queries.

The analysis starts by setting the trivial field sensitive state  $\langle \widehat{sd}, \widehat{sd}, true \rangle$  at the seed statement. Note that for a variable  $v$ :  $\widehat{v} = \langle v, 0, length(T(v)) \rangle$ . It then continues propagating field sensitive states along the edges of the PDG until a fixpoint is reached.

When propagating a field sensitive state over an edge  $(st, st')$  in the PDG, the analysis determines how the dependences on  $use(st)$  at  $st$  contribute to the dependences on  $use(st')$  at  $st'$  according to the state in  $st$ , the relation between  $def(st)$  and  $use(st')$  and whether this edge in the PDG is control dependency edge or data dependency edge (in case the edge is a data edge the interval label  $u$  is also taken into consideration).

Technically, the analysis uses interval calculation that resemble the definition of  $Dep^I$  in Section 5.2 to determine how to propagate the field sensitive states.

### 6.1.3 The Transfer Function

Given a statement  $st$ , a field sensitive state  $s = \langle \widehat{t.\alpha}, \widehat{sd.\beta}, vd \rangle$ , a PDG data dependence edge  $e$  with the interval label  $u$  that connects to the statement  $st'$ , we compute the new state  $s'$  as follows:

**Copy Assignments.** In case  $st$  is a copy assignment statement we need first to transform the dependency from the use of  $st$  to the def of  $st$  before continuing on the def-use edge and reaching the use of  $st'$ . In order to transform the field sensitive dependency to the left hand side of  $st$ , we define the interval  $d$  (*def dependency*) which represents which part of the  $def(st)$  is dependent on the seed :

$$d = Add(\widehat{def(st)}, \widehat{t.\alpha})$$

In addition, we define the interval  $b$  (*use base*) which represents the offset difference between the edge label  $u$  (the actual interval that is used in  $st$  considering partial kill statements on the way), and the syntactic use in  $st'$   $use(st')$ . The purpose of this interval is to adjust the new use-interval refinement that was computed according to the edge label  $u$ , to the syntactic use of  $st'$ .

$$b = Sub(u, \widehat{use(st')})$$

The new state  $s'$  is defined as follows:

$$s' = \begin{cases} \langle Add(Sub(d, u), b), Add(\widehat{sd.\beta}, Sub(u, d)), vd \rangle & (u \supseteq d \vee u \subset d) \wedge vd & (6.1.1) \\ \langle Add(Sub(d, u), b), \widehat{sd.\beta}, vd \rangle & (u \supseteq d \vee u \subset d) \wedge \neg vd & (6.1.2) \\ \emptyset & \text{otherwise} & (6.1.3) \end{cases} \quad (6.1)$$

**Computational Assignments.** In case the from statement  $st$  is of the form  $x := exp(y)$  we need to process the outgoing data dependency edge  $e$  in a delicate manner. Although there might be a target-use dependency refinement on the use  $y$  (i.e.,  $\langle \widehat{y.\alpha}, \widehat{sd.\beta}, vd \rangle$ ) this refinement does

not pass to definition of  $x$  due to the computational function  $exp()$ . Therefore, in this case we define the def dependency interval  $d$

$$d = \widehat{def}(st)$$

Moreover, when computing the new state of the target statement, we should set the  $vd$  flag to false, representing that from this point on (the dependency path), this is not a "pure" value dependence path.

The definition of  $b$  remains the same as for copy assignments. The new state  $s'$  is defined as follows:

$$s' = \begin{cases} \langle Add(Sub(d, u), b), \widehat{sd}.\beta, false \rangle & (u \supseteq d \vee u \subset d) & (6.2.1) \\ \emptyset & \text{otherwise} & (6.2.2) \end{cases} \quad (6.2)$$

**Control Assignments.** In case the edge  $e$  is control edge the new state is:

$$s' = \langle \widehat{use}(st'), \widehat{sd}.\beta, false \rangle \quad (6.3)$$

Using the states that the FSD algorithm computed for each statement we can directly infer the transitive field sensitive dependency between each statement's use to the seed statement's ( $l_1$ ) def.

### 6.1.4 Exploring the Transfer Function

The example in Fig. 6.1 is used to further explain the transfer function. The example contains the FSD results when computing the dependences on  $l_1$ . For each node in the PDG the field sensitive dependency state is attached according the definition above.

**Handling Partial Kill.** Consider transition  $l_2$  to  $l_4$  which includes the processing of two edges due to the destructive update in  $l_3$ . For each outgoing edge from  $l_2$  a new state is computed for  $l_4$  according to 6.1.1. Note that when computing the state of  $l_4$  as the edge label  $u \neq use(l_4)$ , the use-base interval is not empty and it is added to the target-use refinement interval (according to the definition:  $Add(Sub(d, u), b)$ ). The def-dependency interval is  $d = Add(\widehat{a}, \widehat{sd}) = \widehat{a}$ . For the edge label  $u = \widehat{a.f1}$  the use-base interval is  $b = Sub(\widehat{a.f1}, \widehat{a}) = \widehat{a.f1}$ . Thus, the target-use refinement interval is  $Add(\widehat{a}, \widehat{a.f1}) = \widehat{a.f1}$ . In a similar way we compute the state for the edge label  $u = \widehat{a.f3}$ .

**Filtering Unused Intervals.** When processing the transition from  $l_4$  to  $l_5$ , the source state  $\langle \widehat{a.f3}, \widehat{sd.f3}, true \rangle$  leads to a "dead end" as  $d_{l_4} = \widehat{b.f3}$  and the use label  $u$  (that equals the syntactic  $use(l_5)$ ) equals to  $\widehat{b.f1}$ . Clearly, these two intervals are disjoint and thus according to 6.1.3 there is no new dependency (empty state).

However, for the source state  $\langle \widehat{a.f1}, \widehat{sd.f1}, true \rangle$  these same intervals ( $d$  and  $u$ ) are equal and therefore according to 6.1.1 the new state  $\langle \langle b, 0, length(b.f1) \rangle, \widehat{sd.f1}, true \rangle$  is constructed for  $l_5$ . Note that while in  $l_4$  the field sensitive state adds the refinement that only  $\widehat{a.f1}$  part of the use ( $a$ ) is dependent on the seed, on  $l_5$  this additional refinement is not required as the use interval

is  $\widehat{b.f1}$ . The interval  $\langle b, 0, \text{length}(b.f1) \rangle$  is a result of the substraction  $\text{Sub}(\widehat{b.f1}, \widehat{b.f1})$  and it represents that the whole use  $(b.f1)$  is dependent on the seed. We use the explicit form of the interval as there is no symbolic way to present this interval.

**Handling Big L-value Assignments.** When processing the edge from  $l_6$  to  $l_7$  we compute that the use-interval of the new state (for  $l_7$ ) is  $\widehat{c.g2}$ . This data enables us to filter out the spurious dependency when processing the edge from the Big L-value assignment at  $l_7$  to  $l_8$ . Despite the obvious immediate dependency between  $l_7$  and  $l_8$ , the transitive dependency from  $st_{seed}$  does not include  $l_8$  as it is filtered out by the FSD algorithm. The algorithm computes the empty state for  $l_8$  because the def-dependency interval of  $l_7$  ( $d = \widehat{d.g2}$ ) and the use label interval ( $u = \widehat{d.g1}$ ) are disjoint. Thus, according to 6.1.3 we can conclude that  $d.g1$  in  $l_8$  is not transitively field sensitive dependent on  $l_1$  which defines  $sd$ .

For the edge from  $l_7$  to  $l_9$ , the use label interval equals the def-dependency interval of  $l_7$  ( $d = u$ ). Thus, according to 6.1.1, we construct a new state and set the target-use refinement interval to be on the whole use of  $l_8$ :  $\langle d, 0, \text{length}(d.g2) \rangle$ .

**Handling Computational Assignments.** The def variable  $p$  of the assignment in  $l_9$  does not copy the value of  $d.g2$  but rather a computation on this use. In this case a refinement interval on the use (if existed) is not transferred to the def (according to 6.2.1). In addition, from this statement on the dependency path (starting from  $l_1$ ), the dependences are no longer "pure" value dependences (i.e., a value that is assigned to the def interval in  $l_s$  is read in  $l_t$ ). Thus, for the state of  $l_{10.1}$  we set  $vd = false$ .

**Handling Control Dependence Edges.** Consider the guarded command statement in  $l_{10}$ . In the PDG this statement was split into two nodes: one for the predicate  $p$  (node  $l_{10.1}$ ) and one for the guarded statement itself  $i := q$  (node  $l_{10.2}$ ). The outgoing edge of  $l_{10.1}$  in the PDG is a control dependence edge. The new state of  $l_{10.2}$  is constructed according to 6.3.

**Handling non Value Dependences.** When processing the edge between  $l_{10.2}$  and  $l_{11}$ , the edge label  $u = \widehat{i.g1}$  is a sub interval of  $d = \widehat{i}$  resulting that  $\text{Sub}(u, d) = \widehat{i.g1}$ . However, according to (6.1.2) we do not add this interval to the source-def interval (the refinement of seed,  $sd.\beta$ ) of the new state, as the source state contains  $vd = false$  (which represent a non-value dependency). The transition from  $l_{12}$  to  $l_{13}$  concludes the dependency path and adds the target-use interval refinement  $\widehat{j.g2}$  to the new state of the Big L-value assignment in  $l_{13}$  (also according to 6.1.2).

## 6.2 Weakly Typed Languages

One of the main difficulties when handling weakly typed languages is handling assignments where the right hand side and the left hand side are from different structure types. This pattern is quite common when using multipurpose buffers to store structured data at certain points in the program and then to convert it back to a structure type. Another common use case is when a full structure is copied to a small structure that is actually a prefix of the full structure (e.g., the small structure contains only the functional key fields that exist in the beginning of the full structure). In both cases it is crucial to keep the field sensitivity in order to avoid over

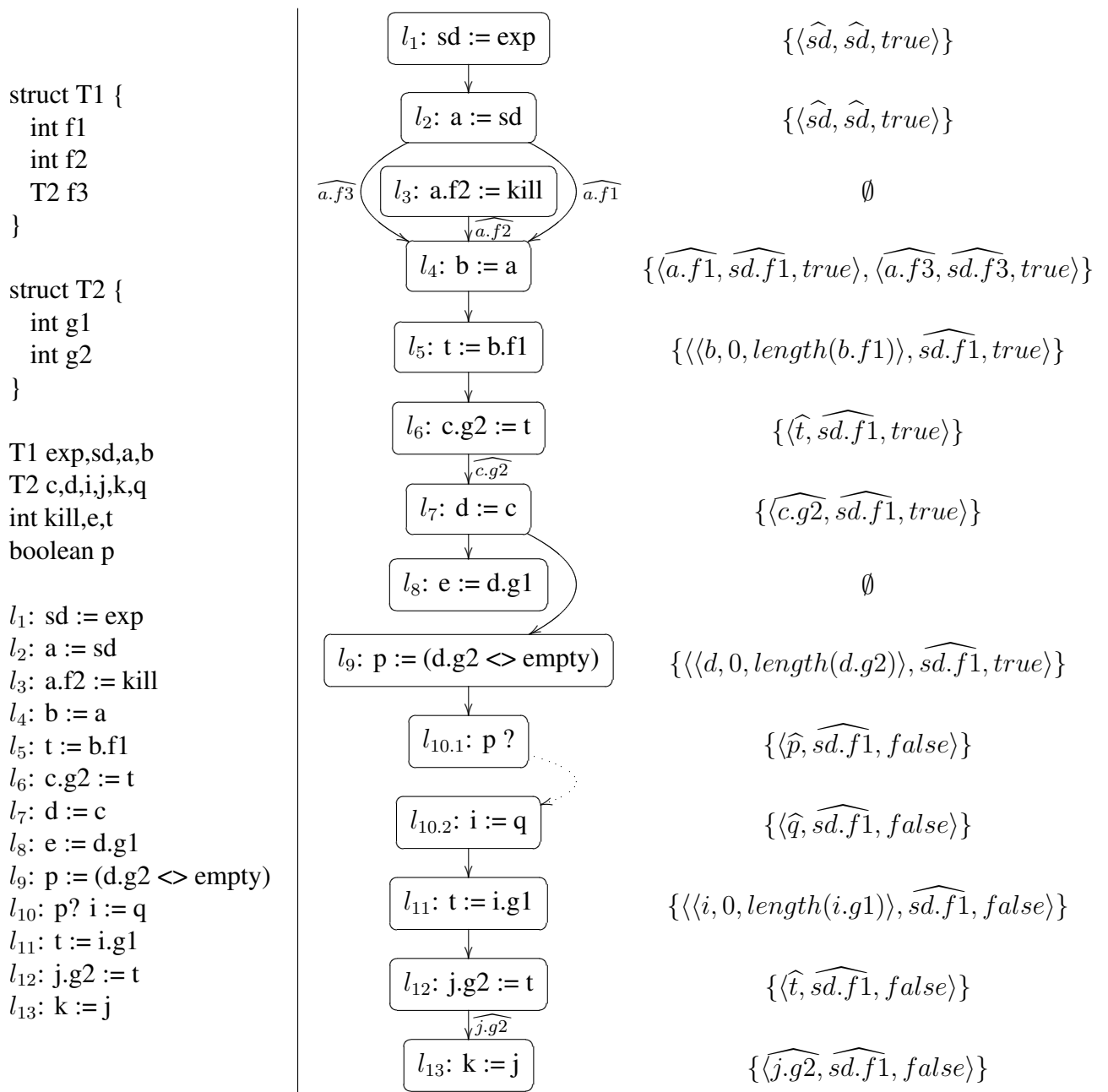


Figure 6.1: FSD algorithm run example on PDG. The state next to each statement is a result of processing the incoming edges. Edge labels that were equal to the target's use were omitted for clarity. Guarded commands were split in order to accommodate control dependence edges.

approximated results. In our study on a preliminary version of the FSD which handled weakly type assignments conservatively as computational assignments, we witnessed many cases where such over approximation changed the results from focusing on a single structure field to hundreds of fields, thus producing unusable results.

In this circumstances even the standard atomization approach may not succeed to keep the field sensitivity property. Consider the case of assigning a structure variable into a multipurpose memory buffer (untyped) and then back to a structure variable. In this scenario even if we disassembled the structure variable into its atomic fields, we may still loose the focus on a certain field as the buffer is not atomized. A different approach may include to also disassemble the buffer, but because these buffers are usually multipurpose and are used to more than one structure type, this can result in decomposing the buffer into byte fragments. This massive atomization does not come without a price, and surely this method has a significant affect on the performance of the analysis.

Clearly, supporting these scenarios may be a very complex task, however, as our algorithm uses intervals to represent access paths we gain support for weakly typed assignments with only a minor adjustment to the computation of the interval based reaching definition. We generalize the transfer function to all copy assignment statement of the form  $x.\alpha = y.\beta$  as follows: For convenience we define the minimum length  $len = \min((end(\alpha) - start(\alpha)), (end(\beta) - start(\beta)))$  and the def's end-offset  $e = start(\alpha) + len$ .

$$\sigma^I[\langle t, i, j \rangle \mapsto \sigma^I(\langle t, i, j \rangle) | t \neq x \vee start(\alpha) > j \vee e < i] \cup \quad (6.4.1)$$

$$\{\langle x, start(\alpha), e \rangle \mapsto l\} \cup \quad (6.4.2)$$

$$\llbracket l : x.\alpha := y.\beta \rrbracket(\sigma^I) = \{\langle x, a, start(\alpha) \rangle \mapsto l' | \forall a, b : \sigma^I(\langle x, a, b \rangle) \mapsto l' \wedge \quad (6.4.3)$$

$$start(\alpha) > a \wedge start(\alpha) < b\} \cup$$

$$\{\langle x, e, b \rangle \mapsto l' | \forall a, b : \sigma^I(\langle x, a, b \rangle) \mapsto l' \wedge \quad (6.4.4)$$

$$e > a \wedge e < b\} \quad (6.4)$$

The adjustment is based on two observations:

- The right hand side of the assignment may be of a smaller size than the left hand side, resulting with a partial assignment to the left hand side. For example, if the length of the right hand side is 4 and the length of the left hand side is 8, only the first 4 bytes are actually defined and the last 4 bytes remains unchanged.
- When handling an interval definition  $\widehat{x.\gamma}$ , there might be a prior reaching definition in  $\sigma^I$  that only partially contains this interval. It may contains only a prefix of the interval  $\widehat{x.\gamma}$  or a suffix of the interval.

The main idea when handling a weakly type language, is that every assignment may "behave" as type W as only part of the left hand side is defined (because the right hand side variable is smaller). Therefore, we handle all the assignments as type W (note that  $\alpha$  and  $\beta$  may be empty). Moreover, the end-offset of the left hand side is determined according to the minimum length between the right hand side and the left hand side ( $e = start(\alpha) + len$ ).

**Term (6.4.1)** copies all the reaching definitions from  $\sigma^I$  except for those on some interval that is contained in  $\langle x, start(\alpha), e \rangle$  or (partially) containing it.

**Term (6.4.2)** generates a new reaching definition for the interval  $\langle x, start(\alpha), e \rangle$ .

**Term (6.4.3)** adjusts the prefix of an existing reaching definition on an interval that partially contains the interval  $\langle x, start(\alpha), e \rangle$ .

**Term (6.4.4)** adjusts the suffix of an existing reaching definition on an interval that partially contains the interval  $\langle x, start(\alpha), e \rangle$ .

Using the adjusted interval based reaching definitions, our labeled PDG contains def-use edges that are precise despite the presence of weakly typed assignment statements. Thus, for both weakly type related use cases - the multipurpose buffer and the substructure assignment (prefix structure) - the FSD algorithm produces precise results in an efficient way.

## 6.3 Inter-procedural Analysis

In order to compute the inter-procedural field sensitive dependences efficiently we use procedural summary information. Thus, during our analysis when we reach a procedure call-site or more specific an actual-in parameter of the call, an FSD summary information is utilized to avoid reanalyzing the callee and continue the dependences analysis in the caller. In addition, we analyze the dependences in the callee's PDG but there is no need to exit the callee as we already continued the dependences analysis after the call using the summary information.

The inter-procedural dependences analysis can be described as a two phase process: In the first phase we summarize each transitive dependency between a procedure's formal-in and formal-out as a *dependency statement* and install it in all of the procedure's call sites; In the second phase we compute the field sensitive dependences across procedures using the FSD algorithm. Specifically, the *dependency statement* are processed as any other statement when computing the field sensitive dependences.

We modify an existing summary algorithm [HRB90]. The summary information construction phase is comprised from two steps. In the first step intra-procedural dependences between formal-in to formal-out nodes are computed via the interval approach. Given a dependency  $d = \langle \alpha, \beta, vd \rangle$  from  $st_1$  (that defines the variable  $f_{in}$ ) to  $st_2$  (that uses  $f_{out}$ ), we construct a *dependency statement*  $ds(d)$ , which represents this dependency, as follows:

$$ds(d) = \begin{cases} f_{out}.\alpha := f_{in}.\beta & \text{if } vd = true \\ f_{out}.\alpha := exp(f_{in}.\beta) & \text{if } vd = false \end{cases}$$

Note that  $ds(d)$  is not necessary in a simplified form. Clearly, it can be transformed to the form defined in Section 2.1.

The next part is an iterative process in which dependency statements of callees are used at call sites to compute the dependency statements of the caller. This is processed until no more dependency statements are computed.

A similar modification can be applied on a more efficient summary algorithm [HRSR94], thus improving the running time of the inter-procedural *field sensitive* dependences analysis.

# Chapter 7

## Empirical Results

This chapter presents the empirical study of our field sensitive dependences algorithm. The implementation of the inter-procedural offset based FSD algorithm is part of an impact analysis tool - PanayaIA - a customization impact analysis tool for Enterprise Resource Planning (ERP) systems. The tool automatically identifies impact of customization changes, i.e. how changes effect the software behavior of SAP, a leading ERP vendor.

### 7.1 The PanayaIA Tool

One of the main advantages of an ERP system is its flexibility, which is achieved by customization tables that drive the program functionality. Customization tables can control the code functionality, for example, whether a report can be saved or the conditions taken into account in the calculation of a price. Therefore, new and changed business requirements can be quickly implemented and tested in the system. The customization values are stored in the system's database and the application reads this data and operates accordingly. Each customization property is represented by a field (a column in the database), and customization attributes from the same functional category are aggregated into a structure (a table in the database). When an ERP Professional wants to update the system behavior, he updates the proper customization attribute which translates to an update of a specific table and column in the system's database.

In order to understand the effect of the user's change, the PanayaIA tool locates all the points in the code where values are read from the customization tables, usually `select` statements. It then computes the transitive dependences using the `select` statement as the seed and reports the impact of the change in non-pragmatically terms, i.e. which dialog screens are effected, which database manipulation performed, which external invocation performed and other output identifiers of the program that explain the impact of the change to the user. We say that a customization pair (table, column) impacts a program if some *output* statement are dependent on customization values read statements (the seed).

The key challenge in analyzing real world ERP system is its significant size (thousands of programs, each may contain over million LOCs), therefore scalability is one of the main concerns. Another factor is the importance of structure variables. ERP systems contains large scale data processing applications that manipulate the business entities. These entities are often large structures with hundreds of fields, and are stored/fetched in the system's database. A common practice when manipulating a business entity is to fetch the whole structure from the

database, even if only few fields are needed for the current computation. This emphasizes the need to handle structure variables in an efficient and precise way.

## 7.2 The Benchmark

The experiments presented in this section are on a selected benchmark of 12 programs which vary in size, complexity, and associated SAP component. Time is measured in minutes. Experiments performed on a computer grid comprised of five Intel servers each with two Dual Intel Xeon 5355 processors and 16GB memory running Windows XP operating system (64-bit) with Java 5.0. In our experiments we study the differences between three transitive dependences algorithms with respect to the structure fields problem:

1. The pre-step atomization approach - disassembling each structure to its components (fields) and the statements that operate on it before starting the data flow analysis (ATOM).
2. The "whole structure" approach - treating each def/use of a structure field as if the whole structure is affected (WS).
3. Our field sensitive dependences approach - using the Field Sensitive dependences analysis (FSD).

Note that all three algorithms compute the transitive inter-procedural dependences over the System Dependence Graph (SDG) using procedural summary dependences. We focused on two comparison criteria : the accuracy of the dependences in terms of number of false-positive (spurious dependences) and the scalability of the analysis approach in terms of time and memory. The results of the atomization approach (ATOM) served as the most precise results and we compared the other two algorithms' results to it. We anticipated that the WS algorithm has a high false positive ratio due to its over approximated nature, and that the FSD algorithm achieves the most precise results.

Table 7.1 compares the accuracy of the dependences for the three algorithms. For each program the table contains the number of customization (table, column) pairs that have an impact on the program according to each algorithm. Column 2 lists the size of the program, in lines of code. Column 3 contains the ATOM algorithm results which also serves as a reference for the other two algorithm. Columns 4-5 contains the WS algorithm results and the percentage of the false positives of the results compered to the base results. The results of the FSD algorithm are specified in columns 6-7. For the WS algorithm, we can see that the percentage of the false positives is relatively high as expected — an average of 62%, and a peak of 500% for a specific program. For the FSD algorithm, there is low false positive percentage of only 1% on average. This is as a result of the implementation of the PanayaIA tool that uses some heuristics through the analysis that consider among other things the size of the program which is quite different between the original program and the atomized program (in some cases the atomized program has 4 times more statements then the original program).

Facing the high number of ERP programs that need to be analyzed and the significant size of the programs, the ability to scale the algorithms has been a key element in the success of the tool. Table 7.2 lists a comparison of the three algorithms according to the execution time of the analysis for each program. As before, the ATOM algorithm results serve as the base for the WS and FSD algorithms. Obviously, the WS algorithm has a large decrease of the execution

Program	KLOC	ATOM	WS		FSD	
		# Pairs	# Pairs	FP	# Pairs	FP
SAPMF02B	3	4	5	25%	4	0%
SAPF110V	8	3	18	500%	3	0%
SAPMV10A	23	5	12	140%	5	0%
SAPMA01B	42	40	120	200%	40	0%
SAPMM07R	65	36	72	100%	36	0%
SDBILLDL	115	23	46	100%	22	-4%
SAPLAMDP	178	168	308	83%	175	4%
SAPMV60A	333	275	492	79%	273	-1%
SAPMV50A	212	374	697	86%	391	4%
SAPLAIST	211	138	264	91%	144	4%
SAPMF02D	226	837	1054	26%	837	0%
SAPMM06B	419	-	560	-	456	-
Average	129	173	281	62%	175	1%

Table 7.1: The number of customization (table, column) pairs that impact the benchmark program and FP, the amount of false positives. The number of lines of code (LOCs) is in thousands.

time of 54% on average with a peak of 72% on a specific program. Very interestingly, the FSD is not far behind with a decrease of 43% on average with a peak of 65%. This is without taking into consideration the time saved for the analysis done prior to the actual computation of the transitive dependences (building the CFG, Constant propagation, etc). Although the WS algorithm performs fewer actions on each step than the FSD algorithm (do not need to calculate and compare field sensitive properties), due to its over approximated nature, it performs more steps than the FSD algorithm. Overall, the improvements of both algorithms in execution time is quite high around the 50%. Whereas the WS algorithm includes a high portion of false-positives 62%, the FSD algorithm achieves this performance boost without hurting the accuracy of the dependences computation (only 1% of false-positives due to heuristics of the tool).

Table 7.3 lists the memory usage of the three algorithms for each benchmark program. As expected the less precise WS algorithm (columns 3-4) has a 35% lower memory consumption than the base result of the ATOM algorithm (column 2). Columns 5-6 contains the results of the FSD algorithm. We can see that the FSD algorithm was able to decrease the memory consumption by 31% on average. FSD successfully analyzed the largest program with a memory usage of 5.7GB while the ATOM algorithm has failed to analyze the program within the available heap size (15GB).

### 7.3 Targeting More Precise Atomization

We studied if a more efficient atomization algorithm in the style of [RFT99], which have shown great improvement for Cobol programs, can reduce the performance penalty of the ATOM algorithm on our case of ERP programs. The main idea here is to perform the atomization only for those structure fields that actually used/refered in the program explicitly and not according to the type information. In worst case of course, this approach may result in full atomization, but

Program	Atom	WS		FSD	
	Time	Time	Imp.	Time	Imp.
SAPMF02B	0.59	0.57	-3%	0.76	30%
SAPF110V	1.43	2.24	56%	1.26	-12%
SAPMV10A	3.99	4.63	16%	3.23	-19%
SAPMA01B	11.20	12.68	13%	10.65	-5%
SAPMM07R	10.11	9.27	-8%	8.24	-18%
SDBILLDL	24.97	20.88	-16%	21.41	-14%
SAPLAMDP	40.86	30.79	-25%	35.81	-12%
SAPMV60A	359.31	99.23	-72%	125.25	-65%
SAPMV50A	43.42	32.53	-25%	31.56	-27%
SAPLAIST	59.13	41.39	-30%	51.08	-14%
SAPMF02D	123.00	56.64	-54%	96.71	-21%
SAPMM06B	-	110.10	-	151.74	-
Average	61.64	28.26	-54%	35.09	-43%

Table 7.2: Elapsed time in minutes and the performance improvement (Imp.).

Prog	ATOM	WS		FSD	
	Mem	Mem	Imp.	Mem	Imp.
SAPMF02B	405	397	-2%	401	-1%
SAPF110V	414	397	-4%	401	-3%
SAPMV10A	644	494	-23%	522	-19%
SAPMA01B	1,284	867	-32%	985	-23%
SAPMM07R	1,399	1,069	-24%	998	-29%
SDBILLDL	2,524	1,574	-38%	1,747	-31%
SAPLAMDP	2,480	1,770	-29%	1,884	-24%
SAPMV60A	8,023	5,044	-37%	5,058	-37%
SAPMV50A	5,538	3,305	-40%	3,556	-36%
SAPLAIST	3,051	1,993	-35%	2,313	-24%
SAPMF02D	5,393	3,345	-38%	3,532	-35%
SAPMM06B	-	4,831	-	5,755	-
Average	2,596	1,688	-35%	1,783	-31%

Table 7.3: The average memory usage (in MB) and the performance improvement.

Program	# Fields	# Unused Fields	% of unused
SAPMF02B	555	288	52%
SAPF110V	2,208	228	10%
SAPMV10A	9,029	1,024	11%
SAPMA01B	29,986	17,398	58%
SAPMM07R	22,600	4,491	20%
SDBILLDL	53,925	11,380	21%
SAPLAMDP	42,445	6,418	15%
SAPMV60A	69,399	16,711	24%
SAPMV50A	69,672	11,909	17%
SAPLAIST	59,936	21,151	35%
SAPMF02D	32,706	2,406	7%
SAPMM06B	82,490	11,355	14%
Average	39,579	8,730	22%

Table 7.4: Results of a study on the number and percentage of unused structure fields.

in some cases the atomization can involve only a small portion of the structure fields. In order to do so, a flow-insensitive algorithm is used to group structure fields to equivalence classes. Two structure fields are in the same equivalence class if there is a direct or indirect assignment between them. These fields are considered from the same type (for type inference) and are disassembled (for atomization) if at least one of the fields is explicitly used.

For ERP programs, we can state that due to the algorithm's flow-insensitive nature and the vast use of large structure variables, the resulting equivalence classes are large and the number of unused structure fields is quite low, which induce atomization of large portion of the structure fields (i.e the optimization unable to improve performance).

Table 7.4 contains the total number of structure fields (column 2) and the number of unused structure fields (column 3) and its ratio (column 4) for each program in our benchmark. We can see that on average approximately 20% of the fields are unused, i.e. the atomization needs to include most of the structure field (over 80%). Therefore, we conclude that this approach does not provide a good optimization to the full atomization approach.

# Chapter 8

## Related Work

Many large applications are written with a heavy use of aggregated types. In some cases, such as ERP systems, a single structure type may contain hundreds of fields. It is not trivial to statically analyze code with heavy use of aggregated structure variables. In the domain of pointer analysis, many techniques were suggested to precisely handle aggregates [YHR99, Hin01, PKH04].

In the field of program dependences many of the research works focused on pointers, recursive data structure and procedures. [HPR89b] present an efficient work to compute flow dependences in the presence of heap pointers and recursive data structure. An efficient solutions for handling procedures and especially recursive calls are presented in works using the System Dependence Graph [HRB90, HRSR94].

However, few techniques for handling aggregates exist in the scope of program dependences. The two common approaches are: the whole structure approach [OO84, Lyl84, Muc97a] which handles a definition point or a use point of a structure field as if it may define or use the whole structure, respectively; and the atomization approach [RFT99, Muc97b] which performs a pre-step of atomization that disassembles the structure to its primitive components.

In our case, the need for the dependences is an industrial impact analysis tool, which needs to analyzes thousands of large programs with extensive use of global and large aggregate structures. Thus using the common approaches for handling aggregates is not feasible.

In [RFT99] a fine grained atomization is presented. The main idea is a flow-insensitive analysis that decide which intervals are needed to be atomized, However, our study presented in Chapter 7, shows that this approach is not effective for the type of program we analyze: large enterprise applications with hundreds of global variables of large structure types.

# Chapter 9

## Further Work and Conclusion

Our empirical results show that with the FSD algorithm we can enjoy both worlds, getting the performance in time and memory of the less accurate WS algorithm (improvement of 43% in time and 31% in memory consumption) with the accuracy of the ATOM algorithm. In addition, FSD successfully analyzed large program which the ATOM algorithm has failed on. This combination enabled us to overcome the big challenge of scalability when analyzing a real life large scale systems.

In our algorithm, we concentrated in the programming language in hand and we decided to tackle the problem of field sensitive dependence analysis which did not have any scalable solution. Some programming language features were not integrated in our solution and thus leave room for further work.

Computing dependences on pointers and recursive data structure were handled precisely in the scope of immediate dependences [HPR89b] but may cause false transitive dependences when facing big L-value assignments. Using our technique to accurately construct the transitive dependences over the existing immediate dependences may be the solution for this problem.

Another area that can be addressed is the computation of field sensitive dependences on arrays. The complexity of handling arrays arises from the need to track the index. This issue can be solved by combining an integer domain within the dependency analysis. Integrating known methods of handling arrays with our algorithm may also extend our solution.

# Bibliography

- [BH93] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Symp. on Principles of Prog. Lang.*, 1993.
- [Bin92] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Conf. on Soft. Maintenance*, 1992.
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.
- [DLAL<sup>+</sup>08] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, and D. Weiss. Customization change impact analysis for ERP professionals via program slicing. In *Int. Symp. on Soft. Testing and Analysis*, 2008.
- [Gal90] K.B. Gallagher. *Using program slicing in software maintenance*. PhD thesis, Comp. Sci. Dept., Univ. of Maryland, Baltimore Campus, 1990. Tech. Rep. CS-90-05.
- [GL91] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Trans. on Soft. Eng.*, 1991.
- [Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet. In *Work. on Prog. Analysis for Soft. Tools and Eng.*, 2001.
- [Hor90] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Conf. on Prog. Lang. Design and Impl.*, 1990.
- [HPR89a] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *Trans. on Prog. Lang. and Syst.*, 1989.
- [HPR89b] Susan Horwitz, Phil Pfeiffer, and Thomas W. Reps. Dependence analysis for pointer variables. In *PLDI*, 1989.
- [HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 1990.
- [HRSR94] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *Symp. on the Foundations of Soft. Eng.*, 1994.
- [LW86] J. Lyle and M. Weiser. Experiments on slicing-based debugging tools. In *Conf. on Empirical Studies of Programming*, June 1986.

- [Lyl84] J. R. Lyle. *Evaluating variations on program slicing for debugging*. PhD thesis, University of Maryland, 1984.
- [Muc97a] S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 8.12. Morgan Kaufmann, 1997.
- [Muc97b] S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 12.2. Morgan Kaufmann, 1997.
- [NEK94] J.Q. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Commun. ACM*, 1994.
- [OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Symp. on Practical Soft. Development Environments*, 1984.
- [Pan09] Panaya Inc. <http://www.panayainc.com>. 2009.
- [PKH04] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for c. In *Work. on Prog. Analysis for Soft. Tools and Eng.*, 2004.
- [RFT99] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Symp. on Principles of Prog. Lang.*, 1999.
- [RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Principles of Prog. Lang.*, pages 49–61, New York, NY, 1995. ACM.
- [SFB07] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Conf. on Prog. Lang. Design and Impl.*, pages 112–122. ACM, 2007.
- [SRH96] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167:131–170, 1996.
- [Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
- [Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4), July 1984.
- [YHR99] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Conf. on Prog. Lang. Design and Impl.*, 1999.

# List of Figures

1.1	(a) A program using user-defined type $T1$ and (b) its PDG. (c) The program after atomization. (d) The atomized PDG. . . . .	8
1.2	(a) a program manipulating user-defined type $T1$ , defined in Fig. 1.1, and (b) its PDG annotated with the abstract states of the interval based algorithm at every statement. . . . .	9
1.3	Dependences computed by vanilla static analysis over the CFG, $\sigma_1$ is the initial dependences set and $\sigma_i$ is the set of dependences computed before processing statement $l_i$ . . . . .	9
1.4	FSD analysis of the program in Fig. 1.1 filters out the spurious dependency of $t.g$ on $seed$ . Type $T1$ is defined in Fig. 1.1. . . . .	11
3.1	Declaration of Types and Variables used in the examples. . . . .	17
3.2	Example of the process of the instrumented semantics on statements of type I. . . . .	18
3.3	Example of the process of the instrumented semantics on statements of type W. . . . .	19
3.4	Example of the process of the instrumented semantics on statements of type R. . . . .	20
3.5	Example of the process of the instrumented semantics on statements of type C. . . . .	20
3.6	Example of the process of the vanilla static analysis on statements of type I. . . . .	22
3.7	Example of the process of the vanilla static analysis on statements of type W. . . . .	22
3.8	Example of the process of the vanilla static analysis on statements of type R. . . . .	23
3.9	Example of the process of the vanilla static analysis on statements of type C. . . . .	24
3.10	Example of the $Dep$ computation using the vanilla static analysis. . . . .	25
3.11	The PDG of the example shown in Fig. 3.10. . . . .	25
4.1	Example of the process of the ATOM static analysis on statements of type I. . . . .	28
4.2	Example of the process of the ATOM static analysis on statements of type C. . . . .	29
4.3	Example of the $Dep^A$ computation using the ATOM static analysis. . . . .	30
5.1	Example of the process of the FSD analysis on statements of type I. . . . .	34
5.2	Example of the process of the FSD analysis on statements of type W. . . . .	35
5.3	Example of the process of the FSD analysis on statements of type R. . . . .	36
5.4	Example of the process of the FSD analysis on statements of type C. . . . .	36
5.5	Example of the $Dep^I$ computation using the FSD analysis. . . . .	39

6.1 FSD algorithm run example on PDG. The state next to each statement is a result of processing the incoming edges. Edge labels that were equal to the target's use were omitted for clarity. Guarded commands were split in order to accommodate control dependence edges. . . . . 47

# List of Tables

5.1	Interval representation for the example type defined in Fig. 3.1. . . . .	34
5.2	Examples to the interval basic operations: Add, Sub. . . . .	37
7.1	The number of customization (table, column) pairs that impact the benchmark program and FP, the amount of false positives. The number of lines of code (LOCs) is in thousands. . . . .	52
7.2	Elapsed time in minutes and the performance improvement (Imp.). . . . .	53
7.3	The average memory usage (in MB) and the performance improvement. . . . .	53
7.4	Results of a study on the number and percentage of unused structure fields. . . . .	54