Techniques for Precise and Scalable Data Flow Analysis

A thesis submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

by

Komal Pathade
(Roll No. 144057001)

Under the guidance of
Prof. Uday Khedker

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
2021

Thesis Approval

The thesis entitled

Techniques for Precise and Scalable Data Flow Analysis

by

Komal Pathade
(Roll No. 144057001)

is approved for the degree of

Doctor of Philosophy

Digital Signature
Uday P Khedker (i01039)
09-Jun-21 12:12:08 PM

Guide

Subhajit Roy

Examiner

Digital Signature
Amitabha A Sanyal (051046)
09-Jun-21 11:02:45 AM

Examiner

Digital Signature
D Manjunath (i98038)
08-Jun-21 10:45:00 PM

Chairman

Date:
Place: CSE Dept.,
IIT Bombay

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY, INDIA

A Declaration of Academic Honesty and Integrity

I declare that this written submission represents my ideas in my own words and where others ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: Komal Pathade
(Roll No. 144057001)

Acknowledgments

I would thank my adviser Prof. Uday Khedker for his support throughout my PhD journey. His constant guidance and encouragement helped me in continuing my work smoothly in the external setting. The regular weekly meetings helped me to stay focused on my work in times of no progress. I am grateful for his patience and efforts in guiding me in clear thinking and effective writing.

I am grateful to Prof. Amitabh Sanyal and Prof. Supratim Biswas, members of my Research Progress Committee, for many useful insights to improve the overall quality of my work.

From my parent organization, Tata Consultancy Services, I want to thank Shrawan, Ulka, Venky, and Advaita for continuously supporting me during PhD right from the joining phase. Thanks to Tukaram and Divyesh for your suggestions, support, and discussions ranging from technical to non-technical aspects. Thanks to Dr. Sachin Lodha for agreeing to be my external supervisor.

I am thankful to my senior PhD students Pritam, Swati, Vini for guiding me at various occasions. I want to thank fellow PhD student Anshuman for his continuous support, encouragement, , suggestions on technical topics and help in academic activities throughout the PhD.

I did not have to worry about my finances thanks to the support and freedom given by Tata Consultancy Services Limited. It is an honor to be part of this great organization.

I am thankful to my parents, my grandmother, my sister Priyanka, my friends Punit, Rohit, Supriya, Madhav, Arun and rest of my family who provided the emotional strength and support to help me overcome the tough times I had during my PhD. Their contribution to this thesis is gratefully acknowledged.

Finally I would like to thank the almighty God. It was his inspiration that I came to IIT Bombay for my PhD. He made it possible that I am able to finish this thesis. THANK YOU.

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY, INDIA

CERTIFICATE OF COURSE WORK

This is to certify that **Komal Pathade** (Roll No. 144057001) was admitted to the candidacy of Ph.D. degree on July 2014, after successfully completing all the courses required for the Ph.D. programme. The details of the course work done are given below.

S.No	**Course Code**	**Course Name**	**Credits**
1	CS 618	Program Analysis	6
2	CS 420	Program Derivation	6
3	CS 601	Algorithms and Complexity	6
4	CS 613	Design and Implementation of Functional Programming Languages	6
5	CS 738	Concepts, Algorithms, and Tools for Model Checking	6
6	CS 435	Linear Optimization	6
7	CS 602	Applied Algorithms	6
8	CS 691	R & D Project	6
9	CSS 801	Seminar	4
10	HS 791	Communication Skills -I	PP
11	CSS 792	Communication Skills -II	PP
		Total Credits	**52**

IIT Bombay

Date: Dy. Registrar (Academic)

Abstract

The *control flow graph* (CFG) representation of a procedure used by virtually all flow-sensitive program analyses, admits a large number of *infeasible* control flow paths i.e., these paths do not occur in any execution of the program. Hence the information reaching along infeasible paths in an analysis is spurious. This may affect the precision of the conventional *maximum fixed point* (MFP) solution of a data flow analysis, because it includes the information reaching along all control flow paths. The existing approaches for removing this imprecision are either specific to a data flow problem with no straightforward generalization, or involve CFG restructuring which may exponentially blow up the size of the CFG. This thesis makes the following three contributions towards eliminating the effect of infeasible paths from data flow analysis, while overcoming the limitations of the existing approaches.

For the first contribution, we lift the notion of MFP solution to define the notion of *feasible path MFP* (FPMFP) solutions that exclude the data flowing along known infeasible paths. The notion of FPMFP is generic and does not involve CFG restructuring. Instead, it takes externally supplied information about infeasible paths and lifts any data flow analysis to an analysis that maintains the distinctions between different paths where these distinctions are beneficial, and ignores them where they are not. Effectively, it gets the benefit of a path-sensitive analysis without performing a conventional path-sensitive analysis. Hence, an FPMFP solution is more precise than the corresponding MFP solution in most cases; it is guaranteed to be sound in each case.

As the second contribution, we formalize the FPMFP computation for inter-procedural analysis to eliminate the effect of infeasible paths that span across multiple procedures (called *inter-procedural infeasible paths*). In particular, we extend FPMFP to a procedure summary based inter-procedural analysis of bit-vector problems. In this, procedure summaries are computed by ignoring the values reaching along input infeasible paths. Here, apart from intra-procedural infeasible paths, we allow a class of inter-procedural infeasible paths as input. A key feature of this formalization is that it is close to the standard MFP formalization, which we believe will lead to easier adoption of FPMFP in data flow analyses.

As the third contribution, we improve practical applicability of FPMFP by proposing novel optimizations that increase the scalability of a FPMFP computation. In particular, we propose to cluster infeasible paths which reduces the number of distinctions that are maintained considerably. Moreover, we propose an optimization that anticipates the potential precision gain and discards the distinctions that do not lead to any precision gain, on the fly. These optimizations are analysis-independent and have improved the scalability of our approach by the factor of 2. The approach has now scaled on codesets of size 150KLOC.

We have implemented our method of feasible path MFP in an industry strength static program analysis tool called TCS ECA. In this, we performed the reaching definitions analysis and the must-defined variables analysis. We show the impact on the corresponding client analyses which includes reduction in the number of def-use pairs, and reduction in the potentially uninitialized variable alarms. Our measurements show that on 93% of the benchmarks (chosen from Open source, Industry, and SPEC) the FPMFP solution was more precise than the corresponding MFP solution.

Publications based on this Thesis

- Journal paper in writing

  Computing Maximum Fixed-Point Solutions over Feasible Paths in Data Flow Analyses

  Komal Pathade, Uday P. Khedker

- Published conference papers

  1. *Computing Partially Path Sensitive Maximum Fixed-Point Solutions in Data Flow Analyses*

     Komal Pathade, Uday P. Khedker

     27th International Conference on Compiler Construction (CC) 2018

  2. *Path Sensitive Maximum Fixed-Point Solutions in Data Flow Analyses in Presence of Intersecting Infeasible Path Segments*

     Komal Pathade, Uday P. Khedker

     28th International Conference on Compiler Construction (CC) 2019

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The effectiveness of techniques that optimize, verify, and debug a computer program depends on the availability of accurate information describing the program behavior during various executions. To this end, a data flow analysis gathers code level information that could answer either a subset of queries over the program (*demand-driven* analyses [23, 16, 15, 27, 26, 57, 48]) or all possible queries of a particular type over the program (*exhaustive* analyses [9, 43, 29]) without executing the program. In general, a demand driven analysis computes less information hence it is easy to scale and poses less challenges towards achieving precision. On the other hand, a scalable and precise exhaustive data flow analysis is harder to achieve. To this end, this thesis proposes a technique for precise and scalable exhaustive data flow analysis.

The outline of this chapter is as follows. It begins by describing the existing solutions of data flow analysis along with their limitations, and our proposed solution (Section 1.1). Next, it gives an overview of our solution (Section 1.2), followed by the contributions made by the thesis (Section 1.3). We conclude the chapter by presenting the organization of the rest of the thesis (Section 1.4).

1.1 Solutions of Exhaustive Data Flow Analyses

In this section, we briefly describe the existing solutions of a data flow analysis along with their limitations. Next, we describe our solution.

An exhaustive data flow analysis uses the *control flow graph* (CFG) representation of a program to compute a data flow solution. Table 1.1 lists the conventional solutions of an ex-

Solution	Includes information along CFPs			Merges information		Computation is decidable	Precision
	Infeasible	Feasible	Spurious	Across CFPs	Across program points		
FI	Yes	Yes	Yes	Yes	Yes	Yes	Least
MFP	Yes	Yes	No	Yes	No	Yes	↓
MOP	Yes	Yes	No	No	No	No	
MOFP	No	Yes	No	No	No	No	Most

Table 1.1: Solutions of a data flow analysis. As we go down the rows, more entries become "Yes" leading to increased precision and decreased efficiency.

haustive data flow analysis and mentions their computability and (relative) precision. These solutions differ from each other in that they include information reaching at a program point along different classes of *control flow paths* (CFPs) that are observed in the CFG of the program. The CFPs that appear in CFG but do not represent any actual execution of the program are *infeasible*; others are *feasible* [8]. Apart from feasible and infeasible CFPs, a data flow solution could also include data flow values reaching along *spurious* CFPs which are paths that result from an over-approximation of CFPs (meaning spurious CFPs do not appear in the CFG). Inclusion of data flow values reaching along infeasible and spurious CFPs may decrease the precision of the corresponding data flow solutions as described below.

- Flow Insensitive (FI) [31] solution computes a single information that represents a sound-approximation of the information present at all program points. The solution merges information across all program points (ignoring the control flow). Effectively, the FI solution considers an over-approximation of the set of CFPs and hence includes information along feasible, infeasible, and spurious CFPs. Computing FI solution is decidable and efficient.

- Maximum Fixed Point (MFP) [31, 30] solution computes an over-approximation of information reaching a program point along feasible and infeasible CFPs. In particular, the solution is inductively computed by merging information in shared segments of CFPs. Since only one piece of information is stored at a program point regardless of the number of paths passing through it, computing MFP solution is *decidable* [30]. The MFP solution is more precise than FI solution because it avoids spurious CFPs.

- Meet Over Paths (MOP) [31, 30] solution also captures the information reaching a program point along feasible and infeasible CFPs. However, it differs from an MFP solution in that it does not merge the information at the shared segments of CFPs. Hence, a MOP solution is more precise than the corresponding MFP solution. Since a program could have infinitely many CFPs, computing this solution is *undecidable* [30] and less efficient compared to the MFP solution.
- Meet Over Feasible Paths (MOFP) solution (defined in Chapter 5) captures the information reaching a program point along feasible CFPs only, and it does not merge information across CFPs. Computing MOFP solution is also undecidable [30]. The information defined by MOFP precisely represents the possible runtime information at each program point because it discards information reaching along infeasible and spurious CFPs. Naturally, MOFP solution is more precise and less efficient than MOP, MFP, and FI solutions.

Example 1.1. For the program in Figure 1.1, we describe the solutions of a data flow analysis that computes possible values of variable a at each program point below. Observe that the CFP $\sigma : e_0 \rightarrow e_1 \rightarrow e_4$ is infeasible.

Node	FI solution	MFP solution	MOP solution	MOFP solution
1	$a \rightarrow [0, 10000]$	$a \rightarrow [0, 0]$	$a \rightarrow 0$	$a \rightarrow 0$
2	$a \rightarrow [0, 10000]$	$a \rightarrow [0, 10000]$	$a \rightarrow 0, 1, \ldots, 10000$	$a \rightarrow 0, 1, \ldots, 10000$
3	$a \rightarrow [0, 10000]$	$a \rightarrow [0, 9999]$	$a \rightarrow 0, 1, \ldots, 9999$	$a \rightarrow 0, 1, \ldots, 9999$
4	$a \rightarrow [0, 10000]$	$a \rightarrow [0, 10000]$	$a \rightarrow 0, 1, \ldots, 10000$	$a \rightarrow 1, 2, \ldots, 10000$

The FI solution computes a single value range (denoted as [minimum possible value, maximum possible value]) $a \rightarrow [0, 10000]$ at all program points. This represents a sound-approximation of possible values of a over all program points. This includes values along feasible, infeasible and spurious CFPs.

The MFP solution computes different value ranges for a at different program points, obtained by merging the possible values reaching the program point along CFPs present in the CFG. Note that at node 4 the value $a \rightarrow 0$ that is reaching along infeasible path is included in the MFP solution.

The MOP solution computes discrete values of a reaching at each program point along feasible and infeasible CFPs. On the other hand, the MOFP solution computes discrete values of a reaching along feasible CFPs only. Observe that the value $a \rightarrow 0$ at node 4 is discarded in MOFP solution because it reaches along the infeasible CFP σ.

We propose the notion of *feasible path MFP* (FPMFP) solutions in which the computation at each program point is restricted to those data flow values that reach along feasible CFPs (data flow values reaching along *known* infeasible CFPs are excluded, and spurious CFPs are not traversed). Hence, a FPMFP solution is at least as precise as the corresponding MFP and FI solutions. Since FPMFP computation is a type of MFP computation (details in Chapter 2), FPMFP computation is decidable (unlike MOFP and MOP).

Example 1.2. For the program in Figure 1.1, the FPMFP solution contains a precise value range [1,10000] for a at node 4 (because it discards the value 0 reaching along infeasible CFP).

Node	FPMFP solution
1	$a \rightarrow [0, 0]$
2	$a \rightarrow [0, 10000]$
3	$a \rightarrow [0, 9999]$
4	$a \rightarrow [1, 10000]$

1.2 Overview and Distinguishing Features of Feasible Path MFP Solutions

We now give an overview of a FPMFP computation and describe its distinguishing features.

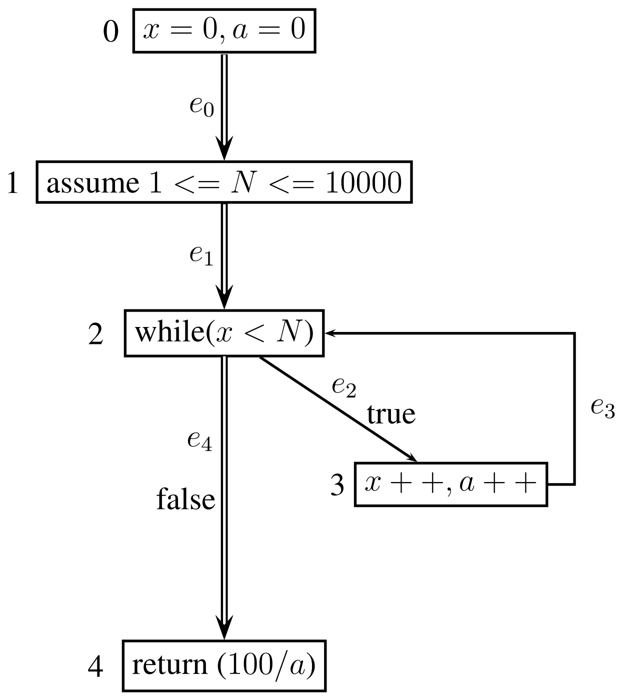


Figure 1.1: Motivating Example: the path $e_0 \to e_1 \to e_4$ (marked with double line arrows) is infeasible hence the value of a is never 0 at node 4.

1.2.1 Overview of FPMFP

A FPMFP computation involves two phases, in which we separate the identification of infeasible CFPs in the CFG of a program, from computation of FPMFP solutions that exclude data flow values reaching along known infeasible CFPs. These two phases are described below.

1. In the first phase, we use the work done by Bodik et.al [8] to detect infeasible CFPs in the CFG of a program. In their work, they identify *minimal infeasible path segments* (MIPS) which are minimal length sub-segments of infeasible CFPs such that the following holds: if a CFP σ contains a MIPS as a sub-segment then σ is infeasible. MIPS[1] captures the infeasibility property of CFPs in a concise form (illustrated in Chapter 2).

   Given a set of MIPS as input, we establish an equivalence relation over them, so that the MIPS that belong to the same equivalence class can be treated as a single unit during FPMFP computation (illustrated in Chapter 4).

[1] We use the acronym MIPS as an irregular noun, whose plural form is same as its singular form.

2. In the second phase, we use the MIPS from the first phase to automatically lift the existing MFP specifications (Chapter 2) of an analysis to its FPMFP specifications. These specifications are then used to compute the FPMFP solution that discards the data flow values reaching along infeasible CFPs.

Example 1.3. The FPMFP computation for the example in Figure 1.1 proceeds as follows. Firstly, Bodik's approach detects $\sigma_1 : e_1 \rightarrow e_4$ as a MIPS (observe that σ_1 is infeasible but no sub-segment of it is infeasible), and gives the MIPS as input to the FPMFP computation. Since only one MIPS is present in the CFG, only one equivalence class of MIPS is created. Secondly, the FPMFP computation discards the value $a \rightarrow [0, 0]$ that reaches along σ_1 at node 4. The final result is as shown in below.

Node	FPMFP solution
1	$a \rightarrow [0, 0]$
2	$a \rightarrow [0, 10000]$
3	$a \rightarrow [0, 9999]$
4	$a \rightarrow [1, 10000]$

We define a FPMFP computation in both intra-procedural and inter-procedural manner in Chapter 2 and 3 respectively. In particular, an intra-procedural FPMFP computation discards the data flow values reaching along infeasible paths that remain confined to a single procedure (intra-procedural infeasible paths). On the other hand, an inter-procedural FPMFP computation discards data flow values reaching along a sub-class of infeasible paths that span across multiple procedures (inter-procedural infeasible paths), and intra-procedural infeasible paths.

1.2.2 Distinguishing Features of FPMFP

A FPMFP computation is as generic as a MFP computation and does not involve CFG restructuring. This differs from the existing approaches that attempt to remove the effect of infeasible CFPs from data flow analysis, in that they either involve CFG restructuring which can exponentially blow up the size of the CFG [7, 50, 39, 5, 36] or are analysis specific with no straightforward generalization [11, 12, 18, 53, 13, 14].

Moreover, separating the FPMFP computation in two phases is possible because of the following observation: infeasible paths is a property of programs, and not of any any particular data flow analysis over programs. Hence, we perform the first phase once for each program, and the second phase is performed once for each different data flow analysis. This avoids the costly repetition involved in identifying infeasible CFPs for each different data flow analysis over the same underlying program, which happens in analysis specific approaches [11, 12, 18, 53, 13, 14].

We now list the contributions made in this work.

1.3 Contributions of the Thesis

This thesis makes the following main contributions.

1. We propose FPMFP solutions to eliminate the effect of infeasible paths from the MFP solutions of data flow analyses without using CFG restructuring, and in a generic manner. The formalization of FPMFP is close to the standard MFP formalization. We believe this will ease the adoption of FPMFP in data flow analysis.

2. We prove that FPMFP computes an over-approximation of the MOFP solution of a data flow analysis.

3. We formalize the concept of FPMFP for inter-procedural setting. In particular, we extend it to the *functional approach* [47] of inter-procedural data flow analysis. We evaluate inter-procedural FPMFP computation on an industry strength static program analysis tool called TCS ECA [1]. Inter-procedural static analyses are known to be desirable to software developer community [6].

4. We improve the practical applicability of FPMFP by adding novel optimizations that improve the scalability of a FPMFP computation by the factor of 2 without affecting soundness or precision of the computed solution. The approach has now scaled on the codesets of size 150 KLOC.

1.4 Organization of the Thesis

Chapter 2 defines and describes the FPMFP solutions at intra-procedural level. Chapter 3 extends the FPMFP solutions to inter-procedural level. Chapter 4 explains the optimizations that improve the scalability of FPMFP computations. Chapter 5 proves that a FPMFP solution is a sound approximation of the corresponding MOFP solution of a data flow analysis. Chapter 6 details the experiments and results. Chapter 7 distinguishes FPMFP from related work. Chapter 8 concludes the thesis, and speculates possible directions for future work.

Chapter 2

Computing Feasible Path MFP Solutions at Intra-procedural Level

This chapter details the computation of FPMFP solutions at intra-procedural level in the following way. First, it defines the pre-requisite terminologies (Section 2.1), followed by the explanation of the ideas that enable separation of the data flow values reaching along infeasible CFPs (Section 2.2). Next, it provides an abstract view of a FPMFP computation, followed by the formal definition of FPMFP solution (Section 2.3). Lastly, it describes the FPMFP computations through data flow equations (Section 2.3.4).

2.1 Background

2.1.1 Value Range (Interval) Analysis

An interval data flow analysis [38, 20] computes the range of values that a program variable can take at various program points. A value range is denoted as $[l, h]$ where l, h represent the minimum and maximum possible value at the program point respectively. For brevity of illustration and simplicity of exposition, we restrict our examples to interval analysis that computes the value ranges of a single variable, assuming value ranges for other variables are unknown. The corresponding lattice $\mathcal{L}_v$ where v is the variable of interest is described below. Nevertheless, we formalize FPMFP for all MFP based data flow analyses.

Lattice of Interval Analysis that computes value ranges for variable v:

$$\mathcal{L}_v = \{v \rightarrow [l, h] \mid l, h \in \mathcal{I},\ l \leq h\}$$

where,

$$\mathcal{I} = \{-\infty, ..., -2, -1, 0, 1, 2, ..., +\infty\}$$

The top value ($\top_v$) is $v \rightarrow [+\infty, -\infty]$. Moreover, the partial order ($\sqsubseteq$) between any two elements in $\mathcal{L}_v$ is defined by the inclusion relation between the corresponding intervals i.e.,

$$(v \rightarrow [l_1, h_1]) \sqsubseteq (v \rightarrow [l_2, h_2]) \iff l_1 \leq l_2 \ \wedge\ h_2 \leq h_1$$

2.1.2 Control Flow Paths

A control flow path is a path in the control flow graph representation of a procedure. The start node of a CFP is always the start node of the CFG, however, the end node of a CFP can be any node in the CFG[1]. We use the term "CFPs that reach node n" to refer to all CFPs that have n as the end node. All CFPs referred henceforth are intra-procedural, unless explicitly stated otherwise.

We denote a CFP by $\rho : n_1 \xrightarrow{e_1} n_2 \xrightarrow{e_2} n_3 \xrightarrow{e_3} \ldots \rightarrow n_p \xrightarrow{e_p} n_{p+1}, p \geq 2$, where n_i denotes the node at the i^{th} position in the path, and e_i denotes its out edge. Also, we call e_p as the last edge of the CFP ρ.

2.1.3 Minimal Infeasible Path Segments

We use the following concepts related to infeasible paths [8].

A CFP $\rho : n_1 \xrightarrow{e_1} n_2 \xrightarrow{e_2} n_3 \xrightarrow{e_3} \ldots \rightarrow n_p \xrightarrow{e_p} n_{p+1}, p \geq 2$ is an *infeasible control flow path* if there is some conditional node $n_k,\ k \leq p$ such that the subpath from n_1 to n_k of ρ is a prefix of some execution path but the subpath from n_1 to n_{k+1} is not a prefix of any execution path[2]. A path segment $\mu : n_i \xrightarrow{e_i} \ldots \rightarrow n_k \xrightarrow{e_k} n_{k+1},\ e_i \neq e_k$ of CFP ρ above is a *minimal infeasible*

[1] For convenience, we distinguish between CFP and CFP segment, in that a CFP segment is any sequence of edges that appears in the CFG, whereas a CFP always starts at the start node of the CFG.

[2] Note that the subscripts used in the nodes and edges in a path segment signify positions of the nodes and edges in the path segment and should not be taken as labels. By abuse of notation, we also use n_1, e_2 etc. as labels of nodes and edges in a control flow graph and then juxtapose them when we write path segments.

path segment (MIPS) if μ is not a subpath of any execution path but every subpath of μ is a subpath of some execution path.

Example 2.1. Figure 2.1 illustrates infeasible paths and MIPS. Observe that the CFP ρ : $n_0 \xrightarrow{e_0} n_1 \xrightarrow{e_1} n_3 \xrightarrow{e_3} n_4 \xrightarrow{e_4} n_5$ is infeasible but not minimal . However, its suffix μ_1 is a MIPS, $\mu_1 : n_1 \xrightarrow{e_1} n_3 \xrightarrow{e_3} n_4 \xrightarrow{e_4} n_5$, because μ_1 is infeasible but no sub-segment of μ_1 is infeasible.

For the MIPS $\mu : n_i \xrightarrow{e_i} \ldots \rightarrow n_k \xrightarrow{e_k} n_{k+1}$, we call all nodes $n_j,\ i < j < k+1$ as the *intermediate* nodes of μ. Additionally, we call the edges $e_i,\ e_k$ as the *start* and the *end* edge of μ respectively, while all edges $e_j,\ i < j < k$ are called as the *inner* edges of MIPS μ. If a MIPS has a single edge, we refer the same edge as start and end edge of the MIPS.

As a rule, the end edge of a MIPS is always a conditional edge, and inner edges exist only if a MIPS has at least 3 edges. In a FPMFP computation, we only admit MIPS that do not contain cycles. All MIPS referred henceforth are MIPS without cycles.

2.2 Separating and Blocking the Data Flow Values Reaching along Infeasible CFPs

We define the following notations that allow us to describe how we separate and block the data flow values reaching along infeasible CFPs. For a given MIPS μ, let *start*(μ), *inner*(μ), and *end*(μ) denote the set of start, inner, and end edges of μ respectively. Sets *start*(μ) and *end*(μ) are singleton because each MIPS has exactly one start and one end edge.

First, we describe how a MIPS can be used to separate the data flow values reaching along infeasible CFPs (Section 2.2.1). Later, we explicate the issues in blocking the data flow values reaching along the infeasible CFPs (Section 2.2.2).

2.2.1 Separating the Data Flow Values Reaching along Infeasible CFPs

We observe the following interesting property of MIPS.

Observation 2.1. CFPs that contain a MIPS μ as a sub-segment are infeasible. Consequently, the data flow values computed along CFPs that contain μ are unreachable at the end edge of

Let a be an integer variable in the figure on the right hand side, then the *states* representing possible values of a at each edge over all executions are as follows.

- *states*$(e_0, \{a\}) = \{\{a \mapsto i\} \mid -\infty \leq i \leq \infty\}$
- *states*$(e_1, \{a\}) = \{\{a \mapsto 0\}\}$
- *states*$(e_2, \{a\}) = \{\{a \mapsto i\} \mid -\infty \leq i \leq \infty\}$
- *states*$(e_3, \{a\}) = \{\{a \mapsto i\} \mid -\infty \leq i \leq \infty\}$
- *states*$(e_4, \{a\}) = \{\{a \mapsto i\} \mid 5 < i \leq \infty\}$
- *states*$(e_5, \{a\}) = \{\{a \mapsto i\} \mid -\infty \leq i \leq 5\}$

Path segment $\mu_1 : n_1 \xrightarrow{e_1} n_3 \xrightarrow{e_3} n_4 \xrightarrow{e_4} n_5$ is a MIPS because e_4 cannot be reached from e_1 in any execution although it can be reached from e_3 in some execution i.e., *states*$(e_1, \{a\}) \cap$ *states*$(e_4, \{a\}) = \Phi$, *states*$(e_3, \{a\}) \cap$ *states*$(e_4, \{a\}) = \{\{a \mapsto i\} \mid 5 < i \leq \infty\}$.

The *start*, *inner*, and *end* edges of μ_1 are given by, *start*$(\mu_1) = \{e_1\}$, *inner*$(\mu_1) = \{e_3\}$, *end*$(\mu_1) = \{e_4\}$.

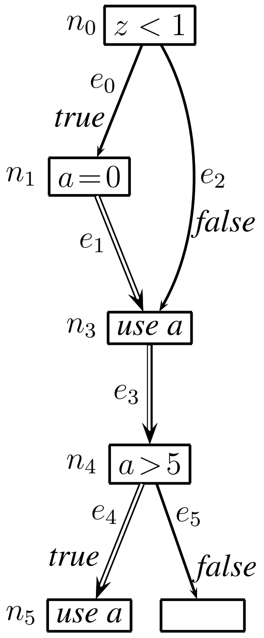


Figure 2.1: Illustrating minimal infeasible path segment (MIPS).

		Data flow values		Meet
		flowing through μ_1	not flowing through μ_1	$\sqcap$
Edges	e_2	$\top$	$z \mapsto [1, \infty]$	$z \mapsto [1, \infty]$
	e_1	$z \mapsto [-\infty, 0]$	$\top$	$z \mapsto [-\infty, 0]$
	e_3	$z \mapsto [-\infty, 0]$	$z \mapsto [1, \infty]$	$z \mapsto [-\infty, \infty]$
	e_4	$\top$ ~~$z \mapsto [-\infty, 0]$~~	$z \mapsto [1, \infty]$	$z \mapsto [1, \infty]$
	e_5	$z \mapsto [-\infty, 0]$	$z \mapsto [1, \infty]$	$z \mapsto [-\infty, \infty]$

Table 2.1: Separating the values of z flowing through MIPS μ_1 for example in Figure 2.1. $\top = z \rightarrow [+\infty, -\infty]$. The last column contains the result of the meet of all data flow values present in a row at each edge.

μ, so these values should be blocked at the end edge. Thus, if we define a data flow analysis that separates the data flow values that are to be blocked, we can eliminate these values thereby avoiding the effect of infeasible paths.

Example 2.2. In Figure 2.1, a CFP $\sigma : n_0 \xrightarrow{e_0} n_1 \xrightarrow{e_1} n_3 \xrightarrow{e_3} n_4 \xrightarrow{e_4} n_5$ is infeasible because it contains MIPS $\mu_1 : n_1 \xrightarrow{e_1} n_3 \xrightarrow{e_3} n_4 \xrightarrow{e_4} n_5$. Hence, the data flow values that reach along σ (e.g., $a \mapsto [0, 0]$, $z \mapsto [-\infty, 0]$) are blocked at e_4 (i.e., at end edge of μ_1).

Observation 2.1 leads us to the following key idea to handle infeasible paths: at each program point, we separate the data flow values reaching along CFPs that contain a MIPS, from the data flow values reaching along CFPs that do not contain any MIPS. This allows us to discard the values reaching along CFPs that contain a MIPS at the end of the MIPS (because these values are unreachable as per Observation 2.1).

For a MIPS μ, the following is a sufficient condition to block the data flow values reaching along CFPs that contain μ.

> $\mathbb{C}$: the data flow values that *flow through* μ (i.e., values that flow along μ —from start edge of μ till the end edge of μ) should be blocked at the end edge of μ.

Example 2.3. Table 2.1 shows the values (of variable z) that flow through MIPS μ_1 of example in Figure 2.1. Here, the value $z \mapsto [-\infty, 0]$ reaches the start edge (e_1) of μ_1, and subsequently flows through μ_1. Hence, we separate $z \mapsto [-\infty, 0]$ from the value that do not flow through μ_1 i.e., $z \mapsto [1, \infty]$. This separation allows us to block $z \mapsto [-\infty, 0]$ at the end edge (e_4) of μ_1.

The final value at each edge, except the end edge, is computed by taking the meet of values that *flow through* μ_1, with values that do not flow through μ_1. In contrast, the final value at the end edge contains only the values that do not flow through μ_1. Thus, we get a precise value range for z at edge e_4 (i.e., $z \mapsto [1, \infty]$) because value $z \mapsto [-\infty, 0]$ is blocked (a MFP solution that includes data flow values along infeasible CFPs gives value range $z \mapsto [-\infty, \infty]$ at edge e_4).

2.2.2 Issues in Blocking the Data Flow Values that Flow Through a MIPS

We now explicate the issues in ensuring that the condition $\mathbb{C}$ (from the previous Section 2.2.1) is satisfied for all MIPS in a program.

A. Modification of Data Flow Values at Intermediate Nodes of MIPS

The data flow values that reach the start edge of a MIPS μ may change or get killed at intermediate nodes or edges when they flow through μ, depending on the type of data flow analysis. In this case, we block the updated data flow value accordingly, as illustrated in example 2.4 below.

Example 2.4. Consider a MIPS μ: $n_1 \xrightarrow{e_1} n_2 \xrightarrow{e_2} n_3 \xrightarrow{e_3} n_4$ with $e_1,\ e_2,\ e_3$ as the start, the inner, and the end edge respectively, as shown in Figure 2.2. The data flow value $x \mapsto [0, 0]$ at the start edge (e_1) of μ changes to $x \mapsto [1, 1]$ as it flows through μ. Hence $x \mapsto [1, 1]$ is blocked at e_3.

B. Presence of Multiple MIPS that Overlap with Each Other

First, we describe the separation of data flow values in a FPMFP computation in presence of a single MIPS in a program. Later, we explain the issue created by presence of multiple overlapping MIPS in the program. We use the example in Figure 2.3a as a running example for explaining FPMFP computation. The two MIPS in Figure 2.3a are $\mu_1 : e_3 \rightarrow e_4 \rightarrow e_5$ and $\mu_2 : e_3 \rightarrow e_4 \rightarrow e_8$ (they are marked with double line arrows). For simplicity of exposition, we restrict the discussion to computation of value range of variable l only.

In FPMFP computation for the example in Figure 2.3a, we separate the value $l \mapsto [0, 0]$ that flows through μ_1 from the value $l \mapsto [2, 2]$ that does not flow through μ_1 as shown in

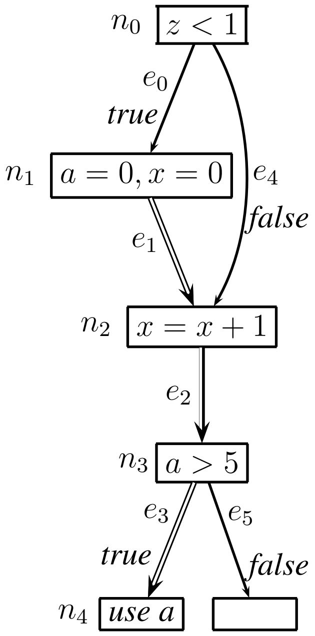
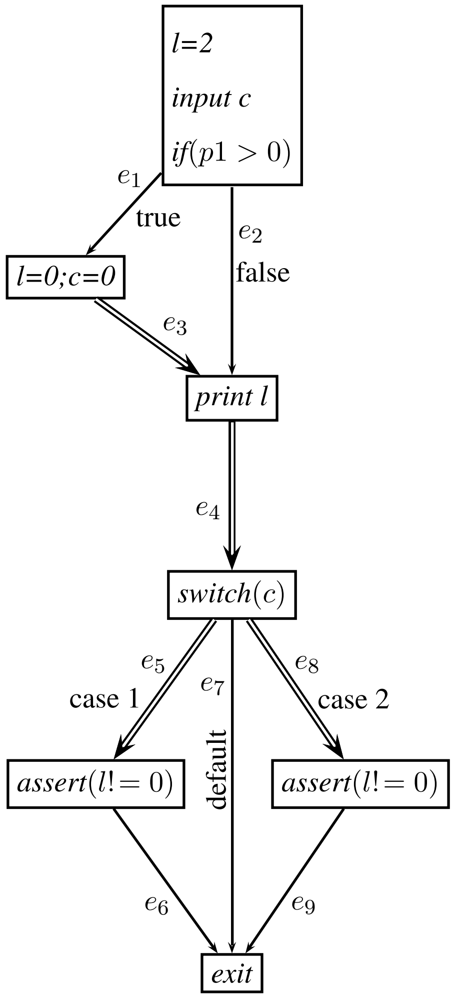


Figure 2.2: Illustrating modification of data flow value at intermediate nodes of a MIPS.

Figure 2.3b. This allows us to block $l \mapsto [0, 0]$ at the end edge (e_5) of μ_1. Thus, the FPMFP solution —computed by taking the meet of data flow values in each row at each edge —contains $l \mapsto [2, 2]$ (a precise value) at edge e_5.

The FPMFP computation explained in the previous paragraph becomes imprecise if we additionally separate the data flow values that flow through MIPS μ_2, as explained below. For MIPS μ_2: $e_3 \rightarrow e_4 \rightarrow e_8$, we separate the values that flow through μ_2 (i.e., $l \mapsto [0, 0]$) as shown in Figure 2.3c. This allows us to block $l \mapsto [0, 0]$ at the end edge (e_8) of μ_2. However, the FPMFP solution (computed in Figure 2.3c) is imprecise because it includes the value $l \mapsto [0, 0]$ at edges e_5 and e_8. In particular, $l \mapsto [0, 0]$ is blocked within μ_1 at e_5, but it is retained in μ_2 because e_5 is not end edge of μ_2. Conversely, at edge e_8, $l \mapsto [0, 0]$ is blocked within μ_2 but is retained in μ_1.

We explicate the reasons for this imprecision below: we separate the data flow values that reach along CFPs that contain a MIPS, however, a MIPS can overlap with other MIPS in a way that both MIPS follow the same CFP resulting in the same data flow values flowing through them. In such a case, at the end edge of a MIPS, the data flow along that MIPS will be blocked but the flow along the other overlapping MIPS will be allowed unless the edge is end edge of both the MIPS. This leads to imprecision. Hence we need to handle the overlapping MIPS



(a)

MIPS details:

- MIPS $\mu_1 : e_3 \rightarrow e_4 \rightarrow e_5$
- MIPS $\mu_2 : e_3 \rightarrow e_4 \rightarrow e_8$
- *start*$(\mu_1) =$ *start*$(\mu_2) = e_3$
- *inner*$(\mu_1) =$ *inner*$(\mu_2) = e_4$
- *end*$(\mu_1) = e_5$, *end*$(\mu_2) = e_8$

		Data flow values		FPMFP
		flowing through μ_1	not flowing through μ_1	$\sqcap$
Edges	e_3	$l \mapsto [0,0]$	$\top$	$l \mapsto [0,0]$
	e_4	$l \mapsto [0,0]$	$l \mapsto [2,2]$	$l \mapsto [0,2]$
	e_5	$\top$	$l \mapsto [2,2]$	$l \mapsto [2,2]$
	e_8	$l \mapsto [0,0]$	$l \mapsto [2,2]$	$l \mapsto [0,2]$

(b)

		Data flow values			FPMFP
		flowing through MIPS		not flowing through MIPS	$\sqcap$
		μ_1	μ_2		
Edges	e_3	$l \mapsto [0,0]$	$l \mapsto [0,0]$	$\top$	$l \mapsto [0,0]$
	e_4	$l \mapsto [0,0]$	$l \mapsto [0,0]$	$l \mapsto [2,2]$	$l \mapsto [0,2]$
	e_5	$\top$	$l \mapsto [0,0]$	$l \mapsto [2,2]$	$l \mapsto [0,2]$
	e_8	$l \mapsto [0,0]$	$\top$	$l \mapsto [2,2]$	$l \mapsto [0,2]$

(c)

Figure 2.3: Issues in blocking the data flow values that flow through overlapping MIPS. The analysis computes value ranges for variable l. $\top = l \rightarrow [+\infty, -\infty]$.

case explicitly. For this purpose, we define *contains-prefix-of* relation between MIPS in below Section 2.3.1.

2.3 The FPMFP Solution

We now formally define a FPMFP solution in the following way. First, we describe the contains-prefix-of relation between MIPS (Section 2.3.1), and describe a partition of set of CFPs created using contains-prefix-of relation (Section 2.3.2). Next, we give a high level view of FPMFP solutions (Section 2.3.3); here, we explain how to separate the values reaching along CFPs that belong to different parts of the partition. Lastly, we define FPMFP solution (Section 2.3.4).

2.3.1 Contains-prefix-of Relation Between MIPS

Definition 2.1. (Contains-prefix-of (CPO)) For a MIPS μ and an edge e in μ, let *prefix*(μ, e) be the sub-segment of μ from *start*(μ) to e. Then, we say that a MIPS μ_1 contains-prefix-of a MIPS μ_2 at e, iff μ_1 contains *prefix*(μ_2, e).

For example in Figure 2.3, the MIPS μ_1 contains-prefix-of MIPS μ_2[3] at edges e_3 and e_4.

In general, for a MIPS μ_1 and an edge e, all MIPS μ_2 that satisfy μ_1 CPO μ_2 at e are obtained as follows: let $\mathcal{U}$ be the set of all MIPS in the CFG of the program

$$\mathit{cpo}_e(\mu_1) = \{\mu_2 \mid \mu_2 \in \mathcal{U},\ \mu_1 \text{ contains } \mathit{prefix}(\mu_2, e)\} \tag{2.1}$$

We observe the following property of $\mathit{cpo}_e(\mu_1)$.

Observation 2.2. Because of the prefix relation, all MIPS in $\mathit{cpo}_e(\mu_1)$ follow the same control flow path till edge e although their start edges may appear at different positions in the path. Next, the values along this path flow through each MIPS in $\mathit{cpo}_e(\mu_1)$. Hence, these values should be blocked at the end edge of every MIPS in $\mathit{cpo}_e(\mu_1)$.

Practically, for an edge sequence $e \to e'$, $\mathit{cpo}_e(\mu)$ may be different from $\mathit{cpo}_{e'}(\mu)$. In particular, all MIPS that reach e may not reach e', and some new MIPS may start at e'. Hence, we define *ext* function below that computes $\mathit{cpo}_{e'}(\mu)$ from $\mathit{cpo}_e(\mu)$.

[3] In this case, μ_2 also contains-prefix-of μ_1 at e_3 and e_4, however in general contains-prefix-of may not be a symmetric relation.

Let *edges*(μ) be the set of all edges in μ, and $\mathcal{S} = cpo_e(\mu)$ then

$$ext(\mathcal{S}, e') = \{\mu \mid \mu \in \mathcal{S},\ e' \in edges(\mu)\} \ \cup\ \{\mu \mid \mu \in \mathcal{U},\ e' \in start(\mu)\} \tag{2.2}$$

In Figure 2.4, the *ext* function captures the change in CPO relation at various edges as follows: at edges e_3 and e_4, the MIPS μ_1 and μ_2 are related with each other with CPO relation i.e., $ext(\{\}, e_3) = \{\mu_1, \mu_2\}$, and at e_4 $ext(\{\mu_1, \mu_2\}, e_4) = \{\mu_1, \mu_2\}$. However, the CPO relation is not present at e_5 i.e., $ext(\{\mu_1, \mu_2\}, e_5) = \{\mu_1\}$. Similarly at e_6 $ext(\{\mu_1, \mu_2\}, e_6) = \{\mu_2\}$.

The benefit of defining the *ext* function is that we can compute the CPO relation between MIPS inductively from one edge to its successor edge without having to worry about entire MIPS. This is analogous to the inductive definitions of MFP solutions, consequently this enables us to derive FPMFP solutions from MFP solutions in Section 2.3.4.

Below, we explain how CPO relation is used to partition the set of control flow paths.

2.3.2 Partitioning the Control Flow Paths

At each edge e, we use the CPO relation to partition the set of CFPs that have e as their last edge. The partition is described as follows.

Let Σ_e be the set of CFPs that have e as the last edge. Moreover, for a CFP σ, let $cpo_e(\sigma)$ represent the set of all MIPS μ such that σ contains $prefix(\mu, e)$. Then, the following relation $\sim$ is an equivalence relation that partitions Σ_e.

We consider two CFPs σ_e and σ'_e in Σ_e as equivalent if $cpo_e(\sigma_e) = cpo_e(\sigma'_e)$ i.e.,

$$\sigma_e \sim \sigma'_e \iff (cpo_e(\sigma_e) = cpo_e(\sigma'_e)) \tag{2.3}$$

It is easy to see that $\sim$ satisfies the properties of an equivalence relation i.e.,

$$\text{Reflexivity:} \quad \sigma_e \sim \sigma_e \tag{2.4}$$

$$\text{Symmetry:} \quad \sigma_e \sim \sigma'_e \implies \sigma'_e \sim \sigma_e \tag{2.5}$$

$$\text{Transitivity:} \quad ((\sigma_e \sim \sigma'_e) \wedge (\sigma'_e \sim \sigma''_e)) \implies (\sigma_e \sim \sigma''_e) \tag{2.6}$$

Two CFPs σ and σ' in Σ_e belong to the same part Π in partition if they are equivalent i.e.,

$$\sigma \in \Pi \wedge \sigma' \in \Pi \implies \sigma \sim \sigma' \tag{2.7}$$

$$\sigma \in \Pi \wedge \sigma' \notin \Pi \implies \sigma \not\sim \sigma' \tag{2.8}$$

At an edge e, each part Π that contains a CFP σ_e can be represented by the set of MIPS $\mathit{cpo}_e(\sigma_e)$. Moreover, MIPS in $\mathit{cpo}_e(\sigma_e)$ are also overlapping with each other i.e., if $\mathit{cpo}_e(\sigma_e) \neq \phi$ then there exists a MIPS $\mu \in \mathit{cpo}_e(\sigma_e)$ such that $\mathit{cpo}_e(\sigma_e) = \mathit{cpo}_e(\mu)$. Therefore, we pair the data flow values flowing through all CFPs in Π with $\mathit{cpo}_e(\mu)$ as explained in the following Section 2.3.3.

2.3.3 High Level View of FPMFP Solution

We now give a high level view of how a FPMFP solution is computed. In a FPMFP computation, at an edge e of MIPS μ, we associate the data flow values that flow through μ with $\mathit{cpo}_e(\mu)$ (instead of μ). These associations are abstractly illustrated in Figure 2.4a and instantiated to an example in Figure 2.4b. We explain each of these below.

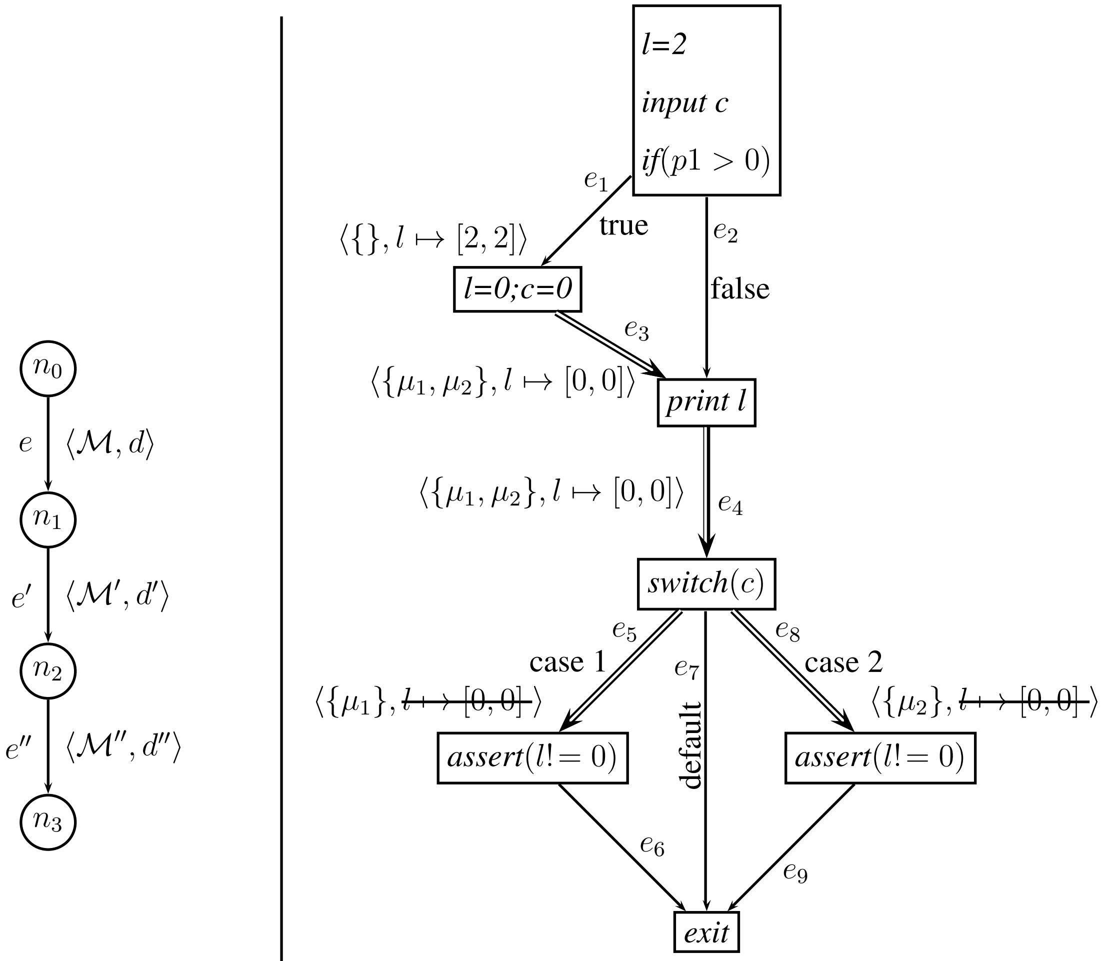
In Figure 2.4a, the data flow values d, d', d'' reach edges e, e', e'' respectively, along the following CFP $\sigma : n_0 \xrightarrow{e} n_1 \xrightarrow{e'} n_2 \xrightarrow{e''} n_3$. The data flow value $d' = f_{n1}(d)$ and $d'' = f_{n2}(d')$ where $f_{ni}(x)$ computes the effect of node n_i on input data flow value x. The corresponding associations with sets of MIPS $\mathcal{M}$, $\mathcal{M}'$, $\mathcal{M}''$ are computed as follows:

$$\mathcal{M} = \mathit{ext}(\{\}, e)$$
$$\mathcal{M}' = \mathit{ext}(\mathcal{M}, e')$$
$$\mathcal{M}'' = \mathit{ext}(\mathcal{M}', e'')$$

The idea is that a pair $\langle \mathcal{M}, d \rangle$ at edge e transforms into another pair $\langle \mathcal{M}', d' \rangle$ at a successor edge e' because

- the CPO relation between MIPS at e' may be different from that at e, and
- d' is computed from d as $d' = f_{n1}(d)$.

In Figure 2.4b, the data flow value $l \mapsto [2, 2]$ at edge e_1 is associated with $\{\}$ because no MIPS goes through e_1. On the other hand, at edges e_3 and e_4 the data flow value $l \mapsto [0, 0]$ is associated with $\{\mu_1, \mu_2\}$ because μ_1 CPO μ_2 at these edges i.e., $\mathit{ext}(\{\}, e_3) = \{\mu_1, \mu_2\}$, $\mathit{ext}(\{\mu_1, \mu_2\}, e_4) = \{\mu_1, \mu_2\}$. Moreover, at the edge e_5, $l \mapsto [0, 0]$ is associated with $\{\mu_1\}$ because $\mathit{ext}(\{\mu_1, \mu_2\}, e_5) = \{\mu_1\}$. Since e_5 is end edge of μ_1, $l \mapsto [0, 0]$ is blocked at e_5.


n_0
e
$\langle \mathcal{M}, d \rangle$
n_1
e'
$\langle \mathcal{M}', d' \rangle$
n_2
e''
$\langle \mathcal{M}'', d'' \rangle$
n_3
l=2
input c
if$(p1 > 0)$
e_1
true
e_2
false
$\langle \{\}, l \mapsto [2,2] \rangle$
l=0;c=0
e_3
$\langle \{\mu_1, \mu_2\}, l \mapsto [0,0] \rangle$
print l
$\langle \{\mu_1, \mu_2\}, l \mapsto [0,0] \rangle$
e_4
switch(c)
e_5
case 1
e_7
e_8
case 2
$\langle \{\mu_1\}, ~~l \mapsto [0,0]~~ \rangle$
$\langle \{\mu_2\}, ~~l \mapsto [0,0]~~ \rangle$
assert$(l != 0)$
default
assert$(l != 0)$
e_6
e_9
exit

(a) Generic view of data flow along a CFP

(b) Illustrating the flow of values for the example in Figure 2.3. For brevity, only values of variable l along CFPs that start at edge e_1 are shown.

Figure 2.4: Computing FPMFP solution

Edges	Associations ($\langle\mathcal{M}, d\rangle$)	FPMFP	MFP
e_1	$\langle\{\}, l \mapsto [2,2]\rangle$	$l \mapsto [2,2]$	$l \mapsto [2,2]$
e_2	$\langle\{\}, l \mapsto [2,2]\rangle$	$l \mapsto [2,2]$	$l \mapsto [2,2]$
e_3	$\langle\{\mu_1, \mu_2\}, l \mapsto [0,0]\rangle$	$l \mapsto [0,0]$	$l \mapsto [0,0]$
e_4	$\langle\{\mu_1, \mu_2\}, l \mapsto [0,0]\rangle,\ \langle\{\}, l \mapsto [2,2]\rangle$	$l \mapsto [0,2]$	$l \mapsto [0,2]$
e_5	$\langle\{\mu_1\},$ ~~$l \mapsto [0,0]$~~$\rangle, \langle\{\}, l \mapsto [2,2]\rangle$	$l \mapsto [2,2]$	$l \mapsto [0,2]$
e_7	$\langle\{\}, l \mapsto [0,2]\rangle$	$l \mapsto [0,2]$	$l \mapsto [0,2]$
e_8	$\langle\{\mu_2\},$ ~~$l \mapsto [0,0]$~~$\rangle, \langle\{\}, l \mapsto [2,2]\rangle$	$l \mapsto [2,2]$	$l \mapsto [0,2]$

Table 2.2: The FPMFP computation for example in Figure 2.3. A pair $\langle\mathcal{M}, d\rangle$ indicates that the data flow value d flows through all MIPS in the set of MIPS $\mathcal{M}$. The final FPMFP solution is computed by taking the meet of values over all pairs at each edge.

Similarly, $l \mapsto [0,0]$ is blocked at e_8. Rest of the associations at each edge are presented in Table 2.2.

In general, we compute a set of pairs of the form $\langle\mathcal{M}, d\rangle$ at each edge e, where $\mathcal{M}$ is a set of MIPS, and d is the data flow value that flows through all MIPS in $\mathcal{M}$. For completeness, we have a pair $\langle\mathcal{M}', d\rangle$ corresponding to each possible subset $\mathcal{M}'$ of $\mathcal{U}$ at each edge e, where $d = \top$ (top value of data flow lattice) if there is no MIPS μ in $\mathcal{M}'$ such that $cpo_e(\mu) = \mathcal{M}'$.

Moreover, at an edge e, presence of a pair $\langle\mathcal{M}, d\rangle,\ d \neq \top$ indicates that the data flow value d flows through each MIPS contained in $\mathcal{M}$, and for some $\mu \in \mathcal{M},\ \mathcal{M} = cpo_e(\mu)$. Similarly, a pair $\langle\{\}, d'\rangle$ indicates that the data flow value d' does not flow through any MIPS at edge e.

Example 2.5. Figure 2.5 illustrates a FPMFP computation through an example analysis that computes value ranges for variable a. Since μ is the only MIPS in the example, each data flow value d is associated with either $\{\mu\}$ or $\{\}$. The data flow value $a \mapsto [0,0]$ reaches the start edge (e_3) of μ, hence, a pair $\langle\{\mu\}, a \mapsto [0,0]\rangle$ is created at e_3. Similarly, at edge e_4, data flow value $a \mapsto [5,5]$ is associated with $\{\}$ i.e., a pair $\langle\{\}, a \mapsto [5,5]\rangle$ is created.

Edge e_6 is end edge of μ, hence, the data flow value $a \mapsto [0,0]$ that is associated with μ in pair $\langle\{\mu\}, a \mapsto [0,0]\rangle$ is discarded i.e., replaced with $\top : a \to [+\infty, -\infty]$ (the resulting pair $\langle\{\mu\}, \top\rangle$ at e_6 is not shown in figure for brevity).

For a program containing k MIPS, theoretically we need 2^k pairs. at each program point (one pair corresponding to each possible subset of set of all MIPS present in the program).

Nevertheless, we prove in Section 4.2 that at any program point at most k pairs can contain the data flow values other than $\top$. Further, our empirical evaluation shows that the number of such pairs to be much smaller than k. Moreover, ignoring the pairs that contain $\top$ value does not affect the FPMFP solution (details are given in Section 4.1.3). The computed pairs satisfy the following property.

> At each program point, the meet of data flow values over all pairs gives at least as precise information as in MFP solution, and may give more precise information if some pairs contain information reaching from infeasible CFPs because such information is discarded. The precision is achieved only when the discarded information has weaker value or incomparable value compared to rest of the information reaching a node. Let D_n be the data flow value in MFP solution at node n, and $\mathcal{S}_n$ be the final set of pairs present at node n after FPMFP computation then
>
> $$D_n \sqsubseteq \bigsqcap_{\langle \mathcal{M}, d \rangle \in \mathcal{S}_n} d$$

Example 2.6. In Figure 2.5, at the edge e_5 the meet of data flow values over all pairs i.e., $\langle \{\mu\}, a \mapsto [0, 0] \rangle$ and $\langle \{\}, a \mapsto [5, 5] \rangle$ is $a \mapsto [0, 5]$ which is same as given by MFP solution in Figure 2.6; however, at the edge e_6 the meet of data flow values over all pairs i.e. $\langle \{\mu\}, \top \rangle$ and $\langle \{\}, a \mapsto [5, 5] \rangle$ is $a \mapsto [5, 5]$ which is more precise compared to $a \mapsto [0, 5]$ given by MFP solution in Figure 2.6.

2.3.4 Defining the FPMFP Solution

We now explain how the above ideas of FPMFP are incorporated in a data flow analysis. We obtain a FPMFP solution in the following two steps.

1. *Step 1*: we lift a data flow analysis to an analysis that computes separate data flow values for different MIPS (Equation 2.11 and 2.12 define the lifted analysis). The lifted analysis blocks the data flow values associated with each MIPS μ at the end edge of μ.

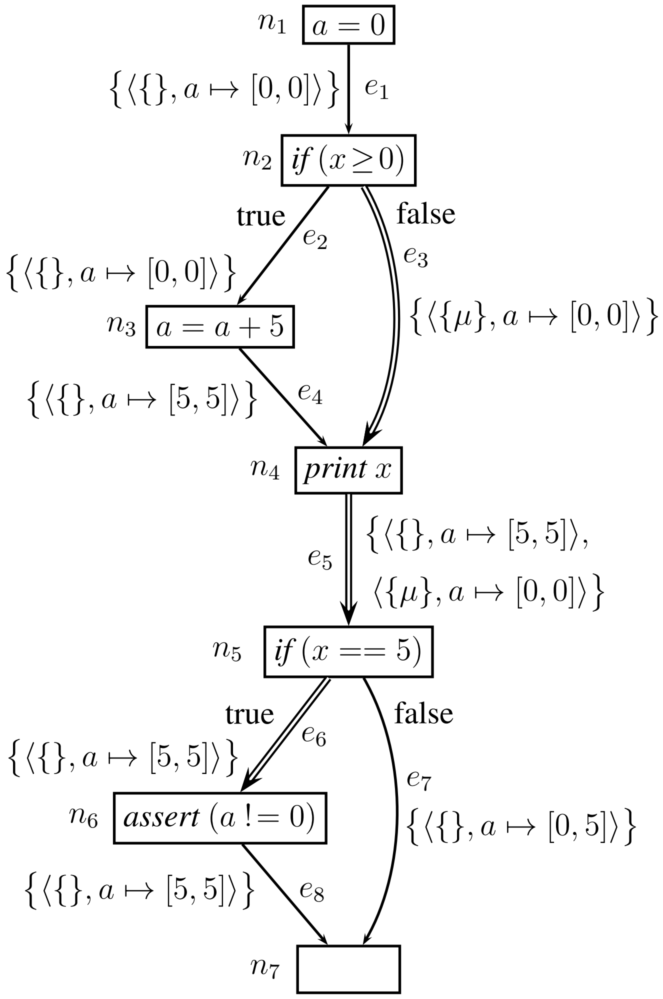
2. *Step 2*: we merge the data flow values that are computed by the lifted analysis (over all pairs at each program point) to obtain the FPMFP solution (Equation 2.18 and 2.19 define the merging of data flow values).

MIPS details:

- MIPS $\mu\colon n_2 \xrightarrow{e_3} n_4 \xrightarrow{e_5} n_5 \xrightarrow{e6} n_6$
- $\mathit{start}(\mu) = \{e_3\}$
- $\mathit{inner}(\mu) = \{e_5\}$
- $\mathit{end}(\mu) = \{e_6\}$

Lattices and data flow values:

- $\mathcal{L} = \big\{a \mapsto [i, j] \mid -\infty \le i \le j \le +\infty\big\}$
- $\top = a \mapsto [+\infty, -\infty]$
- $\mathcal{U} = \{\mu\}$
- $\overline{\mathcal{L}} : \mathcal{P}(\mathcal{U}) \to \mathcal{L}$
- $\overline{\mathit{In}}_{n_6} = \{\langle\{\}, a \mapsto [5,5]\rangle\}$
- $\overline{\overline{\mathit{In}}}_{n_6} = a \mapsto [5,5]$
- $\mathit{In}_{n_6} = a \mapsto [0,5]$ (Figure 2.6)



n_1
$a = 0$
$\{\langle\{\}, a \mapsto [0,0]\rangle\}$
e_1
n_2
$\mathit{if}\,(x \ge 0)$
true
false
e_2
e_3
$\{\langle\{\}, a \mapsto [0,0]\rangle\}$
$\{\langle\{\mu\}, a \mapsto [0,0]\rangle\}$
n_3
$a = a + 5$
$\{\langle\{\}, a \mapsto [5,5]\rangle\}$
e_4
n_4
$\mathit{print}\ x$
e_5
$\{\langle\{\}, a \mapsto [5,5]\rangle,$
$\langle\{\mu\}, a \mapsto [0,0]\rangle\}$
n_5
$\mathit{if}\,(x == 5)$
true
false
e_6
e_7
$\{\langle\{\}, a \mapsto [5,5]\rangle\}$
$\{\langle\{\}, a \mapsto [0,5]\rangle\}$
n_6
$\mathit{assert}\,(a\ != 0)$
$\{\langle\{\}, a \mapsto [5,5]\rangle\}$
e_8
n_7

Figure 2.5: Illustrating a FPMFP solution for an example that computes value ranges for variable a. The lifted data flow analysis computes a set of pairs repesented by $\overline{\mathit{In}}_n/\overline{\mathit{Out}}_n$ at each node n. Since the example contains a single MIPS μ, two pairs are computed (one for $\{\mu\}$ and one for $\{\}$) at each program point. For brevity, we have only shown pairs where $d \neq \top$.

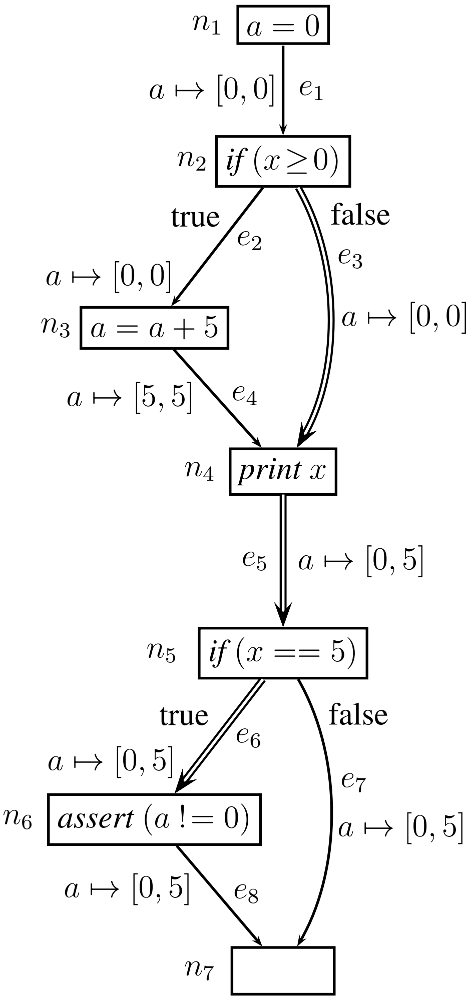


Figure 2.6: MFP solution for example in Figure 2.5. Observe that the value $a \mapsto [0,0]$ is included at edges e_6 and e_8 although it reaches along an infeasible CFP.

We describe each of these steps in detail now.

Step 1. Lifting a MFP Specification to a FPMFP Specification

The data flow equations for computing the FPMFP solution are derived from that of the MFP specifications of a data flow analysis. We explain this lifting by defining the data flow equations, node flow functions, and the meet operator. A key enabler for this lifting is a set of specially crafted edge flow functions[4] (Equation 2.17) that discard data flow values that reach a program point from infeasible CFPs.

[4] The edge flow function for an edge $m \rightarrow n$ uses Out_m to compute In_n in a forward analysis, and vice-versa in a backward analysis.

A. MFP Specifications

Let $\mathit{In}_n, \mathit{Out}_n \in \mathcal{L}$ be the data flow values computed by a data flow analysis at node n, where $\mathcal{L}$ is a meet semi-lattice satisfying the descending chain condition. Additionally, it contains a $\top$ element that represents the top value of the lattice $\mathcal{L}$ —we add an artificial $\top$ value if there is no natural $\top$.

Equations 2.9 and 2.10 (given below) represent the data flow equations for computing MFP solution over the CFG of a procedure, say p. For simplicity, we assume a forward analysis. BI represents the boundary information reaching procedure p from its callers hence BI is used as the value at the In of the start node of p. For all other nodes in p, $\top$ is used as the initial value. The meet operator $\sqcap$ computes the *glb* (greatest lower bound) of elements in $\mathcal{L}$. Function $\mathit{pred}(n)$ returns the predecessor nodes of node n in the CFG of p. The node flow function f_n and edge flow function $g_{m \to n}$ compute the effect of node n and edge $m \to n$ in the CFG respectively. Usually the edge flow functions are identity.

$$\mathit{In}_n = \begin{cases} \mathit{BI} & n = Start_p \\ \displaystyle\bigsqcap_{m \in pred(n)} g_{m \to n}(\mathit{Out}_m) & otherwise \end{cases} \tag{2.9}$$

$$\mathit{Out}_n = f_n(\mathit{In}_n) \tag{2.10}$$

B. FPMFP Specifications

We lift Equations 2.9 and 2.10 to define a data flow analysis that computes data flow variables $\overline{\mathit{In}}_n$ and $\overline{\mathit{Out}}_n$ each of which is a set of pairs of the form $\langle \mathcal{M}, d \rangle$ where $\mathcal{M}$ is a set of MIPS and d is a value in $\mathcal{L}$. Specifically, $\overline{\mathit{In}}_n$ and $\overline{\mathit{Out}}_n$ are values in $\overline{\mathcal{L}}$, where $\overline{\mathcal{L}} : \mathcal{P}(\mathcal{U}) \to \mathcal{L}$ such that $\mathcal{U}$ is the set of all MIPS in the program, and $\mathcal{L}$ is the lattice of values computed by Equations 2.9 and 2.10.

$$\overline{\mathit{In}}_n = \begin{cases} \overline{\mathit{BI}} & n = Start_p \\ \displaystyle\overline{\bigsqcap_{m \in pred(n)}} \overline{g}_{m \to n}(\overline{\mathit{Out}}_m) & otherwise \end{cases} \tag{2.11}$$

$$\overline{\mathit{Out}}_n = \overline{f}_n(\overline{\mathit{In}}_n) \tag{2.12}$$

$$\overline{\mathit{BI}} = \{\langle \{\}, \mathit{BI} \rangle\} \cup \{\langle \mathcal{M}, \top \rangle \mid \mathcal{M} \neq \{\}, \mathcal{M} \subseteq \mathcal{U}\} \tag{2.13}$$

The top value that is used for initialization (i.e., as initial value of $\overline{\mathit{In}}_n/\overline{\mathit{Out}}_n$ when $n \neq Start_p$) is $\{\langle \mathcal{M}, \top \rangle \mid \mathcal{M} \subseteq \mathcal{U}\}$. The node flow function ($\overline{f}_n$) is pointwise application of f_n to the pairs

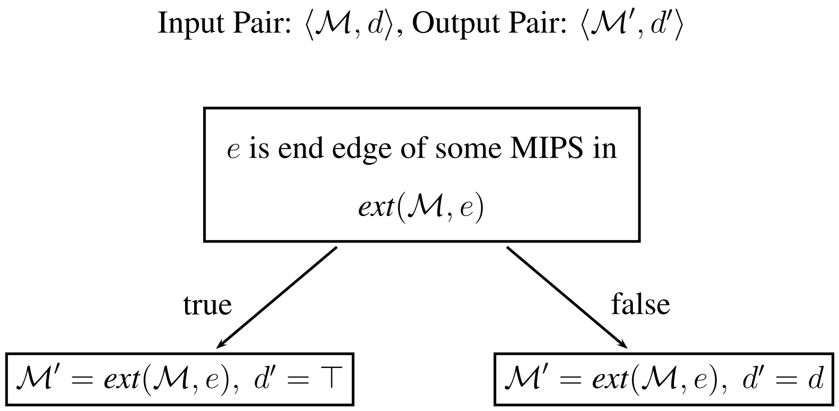

Input Pair: $\langle \mathcal{M}, d \rangle$, Output Pair: $\langle \mathcal{M}', d' \rangle$
e is end edge of some MIPS in $ext(\mathcal{M}, e)$
true
false
$\mathcal{M}' = ext(\mathcal{M}, e),\ d' = \top$
$\mathcal{M}' = ext(\mathcal{M}, e),\ d' = d$

Figure 2.7: Computing output pair $\langle \mathcal{M}', d' \rangle$ from input pair $\langle \mathcal{M}, d \rangle$ in edge flow function $\overline{g}_e$.

in the input set i.e.,

$$\overline{f}_n(\mathcal{S}) = \{\langle \mathcal{M}, f_n(d) \rangle \mid \langle \mathcal{M}, d \rangle \in \mathcal{S}\} \tag{2.14}$$

The meet operator ($\overline{\sqcap}$) is pointwise application of $\sqcap$ to the values in pairs that contain the same set of MIPS i.e.,

$$\mathcal{S} \,\overline{\sqcap}\, \mathcal{S}' = \{\langle \mathcal{M}, d \sqcap d' \rangle \mid \langle \mathcal{M}, d \rangle \in \mathcal{S}, \langle \mathcal{M}, d' \rangle \in \mathcal{S}'\} \tag{2.15}$$

Note that $\overline{\sqcap}$ is defined for all values of $\overline{In}/\overline{Out}$ because $\overline{In}/\overline{Out}$ each contain one pair corresponding to each subset of $\mathcal{U}$.

We now define the edge flow function $\overline{g}(\mathcal{S})$. The edge flow function performs the following two operations over an input pair $\langle \mathcal{M}, d \rangle$ in $\mathcal{S}$.

1. If e is end edge of some MIPS in $\mathcal{M}$, then the data flow value d is blocked,
2. Otherwise d is associated with a set of MIPS $\mathcal{M}'$ where $\mathcal{M}' = ext(\mathcal{M}, e)$ (described in Section 2.3.1).

In fact, the data flow value d from multiple pairs in $\mathcal{S}$ may get associated with the same set of MIPS $\mathcal{M}'$. Hence in the output pair $\langle \mathcal{M}', d' \rangle$, d' is effectively computed by taking the meet of values of d from individual pairs $\langle \mathcal{M}, d \rangle \in \mathcal{S}$ where $\mathcal{M}' = ext(\mathcal{M}, e)$, as shown in Equation 2.17. Let the function $endof(\mathcal{M}, e)$ be a boolean function that denotes whether e is end edge of some MIPS in $\mathcal{M}$ or not i.e.,

$$endof(\mathcal{M}, e) = (\exists \mu \in \mathcal{M}.\, e \in end(\mu)) \tag{2.16}$$

The first term in the right hand side of the following edge flow function corresponds to the Item (1) in the list above, and the second term corresponds to the Item (2) of the list.

$$
\begin{aligned}
\overline{g}_e(\mathcal{S}) = & \left\{ \langle \mathcal{M}', \top \rangle \mid \mathcal{M}' \subseteq \mathcal{U},\ \textit{endof}(\mathcal{M}', e) \right\} \\
& \bigcup \\
& \left\{ \langle \mathcal{M}', d' \rangle \mid \mathcal{M}' \subseteq \mathcal{U},\ \neg\textit{endof}(\mathcal{M}', e),\ d' = \bigsqcap_{\langle \mathcal{M}, d \rangle \in \mathcal{S},\ \textit{ext}(\mathcal{M}, e) = \mathcal{M}'} d \right\}
\end{aligned}
\tag{2.17}
$$

Step 2. Merging Data Flow Values Across Pairs to Obtain FPMFP Solution

The FPMFP solution is a set of values in $\mathcal{L}$ (from Step 1), represented by the data flow variables $\overline{\overline{\mathit{In}}}_n/\overline{\overline{\mathit{Out}}}_n$ (Equation 2.18 and 2.19) for every node n in the CFG of the program. They are computed from $\overline{\mathit{In}}_n/\overline{\mathit{Out}}_n$ using the following operations.

$$\overline{\overline{\mathit{In}}}_n = \textit{fold}_{\sqcap}(\overline{\mathit{In}}_n) \tag{2.18}$$

$$\overline{\overline{\mathit{Out}}}_n = \textit{fold}_{\sqcap}(\overline{\mathit{Out}}_n) \tag{2.19}$$

$$where,\ \textit{fold}_{\sqcap}(\mathcal{S}) = \bigsqcap_{\langle \mathcal{M}, d \rangle \in \mathcal{S}} d \tag{2.20}$$

The $\overline{\overline{\mathit{In}}}_n/\overline{\overline{\mathit{Out}}}_n$ values represent the FPMFP solution which contains the data flow information reaching node n along all CFPs excluding the ones that contain a MIPS because they are infeasible CFPs.

2.4 Chapter Summary

In this chapter, we defined the feasible path MFP solutions for an intra-procedural data flow analysis. At first, we introduced the contains-prefix-of relation between MIPS that allowed us to cluster infeasible path segments in a useful way to handle overlapping MIPS. Next, the specially crafted edge flow functions allowed us to separate and block the data flow values flowing through MIPS.

In the next chapter (Chapter 3), we extend the FPMFP computation to inter-procedural level, and in Chapter 4 we describe optimizations that improve the scalability of FPMFP computation.

Chapter 3

Computing FPMFP Solutions at Inter-procedural Level

In this chapter, we explain the FPMFP computation for *inter-procedural data flow analysis* [47] by describing the call node handling. Specifically, we formalize the FPMFP computation for the *functional approach* [47] of inter-procedural data flow analysis.

We organize the chapter as follows. First, we briefly explain the functional approach of inter-procedural analysis (Section 3.1.1). Next, we describe the class of *inter-procedural MIPS* (MIPS that span across multiple procedures) that we admit for the FPMFP computation (Section 3.1.2). Next, we explain how FPMFP handles call nodes (Section 3.2 and 3.3). Lastly, we explain the limitations, and challenges in FPMFP computation at inter-procedural level 3.4).

3.1 Overview

3.1.1 The Functional Approach of Inter-procedural Analysis

The functional approach consists of two phases. The first phase computes summaries of procedures in a program by traversing the callgraph in a bottom-up order (i.e., summaries for callee procedures are computed before computing the summaries of the caller procedures). In this phase, a non-recursive procedure is analyzed once, while a recursive procedure is analyzed multiple times until a fix-point of procedure summaries is reached. In the second phase, a top-down traversal of the call-graph is performed to propagate the data flow values from callers to

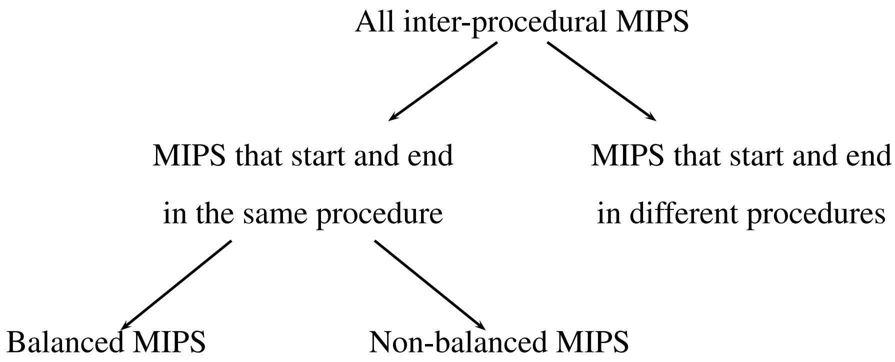

All inter-procedural MIPS
MIPS that start and end in the same procedure
MIPS that start and end in different procedures
Balanced MIPS
Non-balanced MIPS

Figure 3.1: Types of Inter-procedural MIPS

callees. In this phase, a procedure is analyzed in an intra-procedural manner because the effect of call nodes is represented by the summaries of the corresponding callee procedures. Therefore, explaining call node handling and procedure summary computation is sufficient to extend the FPMFP computation at the inter-procedural level using the functional approach.

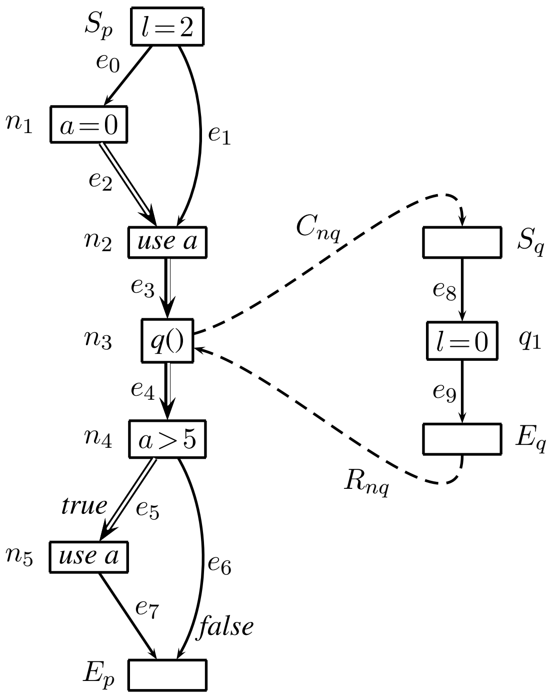
We now state the types of inter-procedural MIPS that we admit (Section 3.1.2), followed by defining the effect of a caller procedure on its callee procedure (Section 3.2), and vice-versa (Section 3.3).

3.1.2 The Class of Inter-procedural MIPS Allowed in FPMFP

An inter-procedural MIPS can span across multiple procedures, unlike an intra-procedural MIPS that remains inside a single procedure. We allow a specific class of inter-procedural MIPS in the inter-procedural FPMFP computation (along with intra-procedural MIPS). We describe this class of inter-procedural MIPS below.

We categorize inter-procedural MIPS as shown in Figure 3.1. Here, we distinguish between the two types of inter-procedural MIPS depending on whether they start and end in the same procedure or not. We admit a subclass of MIPS that belong to the former type; this subclass is defined below. We refer these MIPS as *balanced inter-procedural MIPS*, because each *call edge* is matched by the corresponding *call-return edge*.

Definition 3.1. (Balanced Inter-procedural MIPS) Let a MIPS μ be an inter-procedural MIPS



(a)

		Sets of MIPS		FPMFP
		$\{\}$	$\{\mu_1\}$	$\sqcap$
Edges	e_3	$l \mapsto [2,2]$	$l \mapsto [2,2]$	$l \mapsto [2,2]$
	e_4	$l \mapsto [0,0]$	$l \mapsto [0,0]$	$l \mapsto [0,0]$
	e_8	$l \mapsto [2,2]$		$l \mapsto [2,2]$
	e_9	$l \mapsto [0,0]$		$l \mapsto [0,0]$

The FPMFP solution: path segment $\mu_1 : e_2 \to e_3 \to e_4 \to e_5$ is a MIPS. For brevity, value range of only l is shown.

(b)

Figure 3.2: Computing FPMFP solution using the functional approach of inter-procedural analysis: S, E represent the start and end nodes of a procedure respectively. C_{nq} represents the transfer of control from n^{th} callsite of procedure q to entry of q, and R_{nq} represents the return of control from exit of q to n^{th} callsite of q.

and $\mathcal{V}$ be the set of variables present in the condition on the end edge of μ[1]. Then, we say μ is a balanced inter-procedural MIPS if it starts and ends in the same procedure in a non-recursive manner (i.e., for each call edge there is a corresponding call return edge in the MIPS), and the variables in $\mathcal{V}$ are not modified inside the procedures called through the intermediate nodes of μ[2].

For example in Figure 3.2, MIPS $\mu_1 : n_1 \xrightarrow{e_2} n_2 \xrightarrow{e_3} n_3 \xrightarrow{e_4} n_4 \xrightarrow{e_5} n_5$ is a balanced inter-procedural MIPS.

[1] Recall that end edge of a MIPS is always a conditional edge (Section 2.1)

[2] In general, this may not hold for all inter-procedural MIPS that start and end in the same procedure. Specifically, there could exist an inter-procedural MIPS μ_1 such that the variables in the condition on the end edge of μ_1 are modified along some paths inside the procedures called from intermediate nodes of the MIPS.

A balanced MIPS has the following interesting property.

> For a balanced MIPS μ that starts and ends in a procedure p, the control flow inside the procedures called from intermediate nodes of μ can be abstracted out during the FPMFP computation of p.

Example 3.1. In Figure 3.2, MIPS $\mu_1 : n_1 \xrightarrow{e_2} n_2 \xrightarrow{e_3} n_3 \xrightarrow{e_4} n_4 \xrightarrow{e_5} n_5$ goes through the call node n_3 that calls procedure q. However, the control flow inside q is abstracted out during a FPMFP computation of the caller procedure p. In particular, the edges in procedure q are not marked as inner edges of μ_1 (even though the call $q()$ is part of μ_1).

The above property of a balanced MIPS allows us to eliminate the data flow values flowing through the MIPS without having to worry about the control flow inside the functions called from intermediate nodes of the MIPS, thereby avoiding the necessity of tracking caller-callee sequences (call chains). We describe the challenges in handling non-balanced inter-procedural MIPS in Section 3.4.

Henceforth, we assume all inter-procedural MIPS that are input to our analysis are balanced. We now define how the effect of caller and callee procedures are computed in FPMFP for bit-vector problems. The formalization can be easily extended to any data flow problem where function summaries can be computed in calling context independent manner.

3.2 Defining the Effect of a Caller Procedure on its Callee Procedure

We now describe how the data flow values from a caller procedure are propagated to a callee procedure in a FPMFP computation. Specifically, we explain the computation of the boundary information of a callee procedure from its caller procedures.

Figure 3.3 shows an abstract view of the handling of a call node in a FPMFP computation. The boundary information ($\overline{\mathit{BI}}_q$) for a callee procedure q is computed by taking the meet of data flow values present at all call sites of q. If C_q is the set of all call sites of q, and $\mathcal{U}_q$ is the set containing 1) intra-procedural MIPS of q, and 2) balanced inter-procedural MIPS that start and

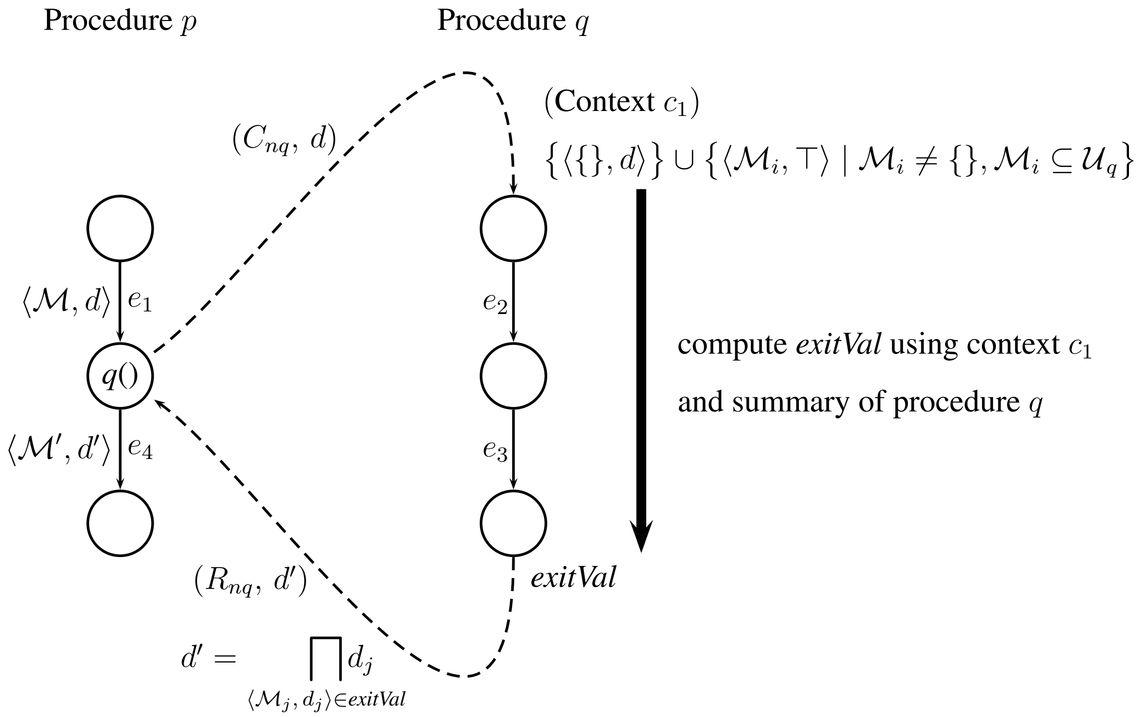

Procedure p
Procedure q
$(C_{nq},\, d)$
(Context c_1)
$\{\langle\{\}, d\rangle\} \cup \{\langle\mathcal{M}_i, \top\rangle \mid \mathcal{M}_i \neq \{\}, \mathcal{M}_i \subseteq \mathcal{U}_q\}$
$\langle\mathcal{M}, d\rangle$ e_1
e_2
$q()$
compute *exitVal* using context c_1
and summary of procedure q
$\langle\mathcal{M}', d'\rangle$ e_4
e_3
$(R_{nq},\, d')$
exitVal
$d' = \bigsqcap_{\langle\mathcal{M}_j, d_j\rangle \in exitVal} d_j$

Figure 3.3: Generic view of call node handling in a FPMFP computation. C_{nq} represents the transfer of control from n^{th} callsite of procedure q to entry of q, and R_{nq} represents the return of control from exit of q to n^{th} callsite of q. $\mathcal{U}_q$ is the set containing all i) intra-procedural MIPS of q, and ii) balanced inter-procedural MIPS that start and end in q.

end in q then $\overline{\mathit{BI}}_q$ is computed as follows:

$$\overline{\mathit{BI}}_q = \{\langle\{\}, x\rangle\} \cup \{\langle\mathcal{M}, \top\rangle \mid \mathcal{M} \subseteq \mathcal{U}_q, \mathcal{M} \neq \{\}\} \text{ ,where}$$
$$x = \bigsqcap_{n \in C_q} \mathit{fold}_{\sqcap}(\overline{\mathit{In}}_n)$$

In Figure 3.2, the data flow information in $\overline{\mathit{In}}_{n3}$ is used to compute the boundary information ($\overline{\mathit{BI}}_q$) for procedure q i.e., $\overline{\mathit{BI}}_q = \{\langle\{\}, l \mapsto [2,2]\rangle\}$.

Moreover, a single boundary is maintained for each procedure. If there is a change in the data flow information at any call site of a procedure q then the new boundary $\overline{\mathit{BI}}'_q$ is merged with the existing boundary information of q (i.e.,$\overline{\mathit{BI}}_q$). If this leads to a change in the resulting boundary information of q, the procedure q is added to the worklist for solving. Later, the procedure q is solved in an intra-procedural manner using the new boundary information using the worklist algorithm [47].

3.3 Defining the Effect of a Callee Procedure on its Caller Procedure

3.3.1 Overview

The MFP computation (specified by Equations 2.9 and 2.10 from Section 2.3) defines the node flow functions (f_n) for all nodes —including nodes that call other procedures. In the functional approach of inter-procedural analysis [47], the call node flow functions are defined using summaries of the corresponding callee procedures. We use these node flow functions to compute the effect of a procedure call on the caller procedure using Equation 2.14. For example, in Figure 3.2 an analysis for computing MFP solution defines the node flow function f_{n3} which incorporates the effect of call $q()$ i.e., q replaces the original value of l by 0. The FPMFP computation uses f_{n3} to compute $\overline{f}_{n3}$ using Equation 2.14 as follows:

$$\begin{aligned}\overline{f}_{n3}\big(\{\langle\{\}, l \mapsto [2,2]\rangle, \langle\{\mu_1\}, l \mapsto [2,2]\rangle\}\big) &= \{\langle\{\}, f_{n3}(l \mapsto [2,2])\rangle, \langle\{\mu_1\}, f_{n3}(l \mapsto [2,2])\rangle\} \\ &= \{\langle\{\}, l \mapsto [0,0]\rangle, \langle\{\mu_1\}, l \mapsto [0,0]\rangle\}\end{aligned}$$

We elaborate the procedure summary computation for bit-vector frameworks in Section 3.3.2 and substitution of these summaries in node flow functions in Section 3.3.3.

3.3.2 Defining GEN and KILL Summaries over Feasible Paths for Bit-vector Frameworks

In this section, we state the MFP specifications used for the computation of procedure summaries for bit-vector frameworks [31]. Next, we lift these specifications to the corresponding FPMFP specifications.

We use the standard notions of a procedure summary computation for bit-vector frameworks from [31]. In these frameworks, the node flow functions are of the following form $f_n(X) = (X - \text{KILL}_n) \cup \text{GEN}_n$, where GEN_n and KILL_n are independent of input information X. Hence the procedure summaries can be constructed by composing individual node flow functions, independent of the calling context information.

The meet operator in bit-vector frameworks is either the set-union or set-intersection. For simplicity of exposition, we explain the procedure summary computation assuming the meet operator is union, henceforth we refer these data flow problems as *union problems*. We explain the computation for forward data flow analysis, a dual modeling exists for backward analysis (like live variables analysis [31]).

Let GEN_n, KILL_n be the constant GEN and KILL information at a non-call node n respectively. Then GEN and KILL summaries for procedures are computed by solving corresponding GEN and KILL data flow problems as described below. Since the GEN Summaries are defined using the results of KILL Summaries, first we explain the KILL data flow problem.

Solving the KILL Data Flow Problem

The kill summary for union problems is computed using $\cap$ as the meet operator, indicating that a value is added to the kill summary of a procedure iff the value is killed along all CFPs that reach the exit of the procedure. Initially, kill summary of all procedures is set to $\top$ which is the top value of the kill data flow lattice, and $\{\}$ is used as the boundary value. The MFP specifications of a KILL data flow problem for a procedure p are as follows.

$$\text{KIn}_n = \begin{cases} \{\} & n = Start_p \\ \bigcap\limits_{m \in pred(n)} \text{KOut}_m & otherwise \end{cases} \tag{3.1}$$

$$\text{KOut}_n = \text{KIn}_n \cup \textit{nodekill}(n) \tag{3.2}$$

$$\textit{nodekill}(n) = \begin{cases} \text{KILL}_n & \textit{n is not a call node} \\ \text{KOut}_{\textit{Exit}_q} & \textit{n calls a procequre q} \end{cases} \tag{3.3}$$

nodekill(n) represents the kill information for the node n; the information is constant if n is not a call node. Otherwise, *nodekill*(n) is the information at the exit the procedure called from n. For simplicity of exposition, we assume at most one procedure can be called from a node (in practice, more than one procedures can be called from a node through function pointers, in which case the kill summary of the node is computed by doing intersection of kill information at exit of all procedures that can be called from n).

Equations 3.1 to 3.3 are solved for each procedure chosen in a bottom up traversal of the call graph of a program. In this, each non-recursive procedure is solved once, while a recursive procedure is solved multiple times until the data flow values at all points saturate.

The final value in KOut at the exit of a procedure p is the KILL summary for procedure p, as given by the MFP solution. This summary may include the KILL information reaching from infeasible CFPs. Hence, to get a more precise summary, we lift the MFP specifications to the FPMFP specifications for each procedure p in the program as follows. Let $\mathcal{U}_p$ be the set containing 1) intra-procedural MIPS of p, and 2) balanced inter-procedural MIPS that start and end in procedure p.

The FPMFP specifications (obtained by lifting MFP specifications from equations 3.1 to 3.3):

$$\overline{\text{KIn}}_n = \begin{cases} \overline{\text{KILLBI}} & n = Start_p \\ \\ \overline{\bigsqcap}_{m \in pred(n)} \overline{g}_{m \to n}(\overline{\text{KOut}}_m) & otherwise \end{cases} \tag{3.4}$$

$$\overline{\text{KILLBI}} = \{\langle \{\}, \{\} \rangle\} \cup \{\langle \mathcal{M}, \top \rangle \mid \mathcal{M} \subseteq \mathcal{U}_p,\ \mathcal{M} \neq \{\}\} \tag{3.5}$$

$$\overline{\text{KOut}}_n = \{\langle \mathcal{M}, \mathcal{D} \cup \overline{nodekill}(n) \rangle \mid \langle \mathcal{M}, \mathcal{D} \rangle \in \overline{\text{KIn}}_n\} \tag{3.6}$$

$$\overline{nodekill}(n) = \begin{cases} \text{KILL}_n & \textit{n is not a call node} \\ \\ \bigcap_{\langle \mathcal{M}, \mathcal{D} \rangle \in \overline{\text{KOut}}_{Exit_q}} \mathcal{D} & \textit{n calls a procequre q} \end{cases} \tag{3.7}$$

The $\overline{\sqcap}$ does a point-wise intersection of the kill data flow values in the input set of pairs as follows.

$$\mathcal{S}_1 \overline{\sqcap} \mathcal{S}_2 = \{\langle \mathcal{M}, \mathcal{D}_1 \cap \mathcal{D}_2 \rangle \mid \langle \mathcal{M}, \mathcal{D}_1 \rangle \in \mathcal{S}_1,\ \langle \mathcal{M}, \mathcal{D}_2 \rangle \in \mathcal{S}_2\} \tag{3.8}$$

The final kill summary of a procedure p is computed by doing intersection of the data flow values at the exit of p i.e.,

$$\text{KSUM}_p = \bigcap_{\langle \mathcal{M}, \mathcal{D} \rangle \in \overline{\text{KOut}}_{Exit_p}} \mathcal{D} \tag{3.9}$$

The edge flow function $\overline{g}_{m \to n}$ is same as defined by Equation 2.17. We now explain the GEN summary computation of a procedure below.

Solving the GEN Data Flow Problem

The GEN summary for union problems is computed using $\cup$ as the meet operator, indicating that a data flow value is added to GEN summary if it is generated along at least one CFP that reaches the exit of the procedure. Next, $\{\}$ is used as the boundary value. The corresponding MFP specifications are as follows.

The MFP specifications:

$$\text{GIn}_n = \begin{cases} \{\} & n = Start_p \\ \\ \bigcup\limits_{m \in pred(n)} \text{GOut}_m & otherwise \end{cases} \tag{3.10}$$

$$\text{GOut}_n = (\text{GIn}_n - \textit{nodekill}(n)) \cup \textit{nodegen}(n) \tag{3.11}$$

$$\textit{nodegen}(n) = \begin{cases} \text{GEN}_n & \textit{n is not a call node} \\ \text{GOut}_{\textit{Exit}_q} & \textit{n calls a procedure q} \end{cases} \tag{3.12}$$

Based on these MFP specifications, we derive the FPMFP specifications for each procedure p in the program as follows. Let $\mathcal{U}_p$ be the set containing 1) intra-procedural MIPS of p, and 2) balanced inter-procedural MIPS that start and end in p.

The FPMFP specifications:

$$\overline{\text{GIn}}_n = \begin{cases} \big\{\langle \mathcal{M}, \{\}\rangle \mid \mathcal{M} \subseteq \mathcal{U}_p\big\} & n = Start_p \\ \\ \overline{\bigsqcap}'_{m \in pred(n)} \overline{g}_{m \to n}(\overline{\text{GOut}}_m) & otherwise \end{cases} \tag{3.13}$$

$$\overline{\text{GOut}}_n = \big\{\langle \mathcal{M}, (\mathcal{D} - \overline{\textit{nodekill}}(n)) \cup \overline{\textit{nodegen}}(n)\rangle \mid \langle \mathcal{M}, \mathcal{D}\rangle \in \overline{\text{GIn}}_n\big\} \tag{3.14}$$

$$\overline{\textit{nodegen}}(n) = \begin{cases} \text{GEN}_n & \textit{n is not a call node} \\ \\ \bigcup\limits_{\langle \mathcal{M}, \mathcal{D}\rangle \in \overline{\text{GOut}}_{Exit_q}} \mathcal{D} & \textit{n calls a procedure q} \end{cases} \tag{3.15}$$

$\overline{\sqcap}'$ does a point-wise union of data flow values in the set of input pairs as follows.

$$\mathcal{S}_1 \overline{\sqcap}' \mathcal{S}_2 = \big\{\langle \mathcal{M}, \mathcal{D}_1 \cup \mathcal{D}_2\rangle \mid \langle \mathcal{M}, \mathcal{D}_1\rangle \in \mathcal{S}_1,\ \langle \mathcal{M}, \mathcal{D}_2\rangle \in \mathcal{S}_2\big\} \tag{3.16}$$

The final gen summary of a procedure p is computed by doing the union of data flow values at the exit of the procedure p i.e.,

$$\text{GSUM}_p = \bigcup_{\langle \mathcal{M}, \mathcal{D} \rangle \in \overline{\text{GOut}}_{Exit_p}} \mathcal{D} \tag{3.17}$$

3.3.3 Substituting GEN and KILL Summaries in FPMFP

The GEN and KILL summaries are computed for all procedures using the corresponding constant boundary values. We call this summary computation phase. Later, these GEN and KILL summaries are used to compute the effect of procedure p at all callsites of p during the actual FPMFP computation (using actual boundary values) as follows: Let $n : p()$ be a callsite of p then

$$\overline{f}_n(\mathcal{S}) = \{\langle \mathcal{M}, f_n(\mathcal{D}) \rangle \mid \langle \mathcal{M}, \mathcal{D} \rangle \in \mathcal{S}\} \tag{3.18}$$

$$\text{where, } f_n(X) = (X - \text{KSUM}_p) \cup \text{GSUM}_p \tag{3.19}$$

3.4 Limitations and Challenges in FPMFP Computation at Inter-procedural Level

Currently, we have not defined FPMFP computation for context sensitive data flow analyses. We leave this part for future work. Consequently, we have not defined the FPMFP computation for non-balanced inter-procedural MIPS, because they may necessitate a context sensitive analysis as described below.

A non-balanced inter-procedural MIPS can be of three types:

1. MIPS that starts in a procedure and ends in its callee procedure (or a transitive callee down the call chain), or

2. MIPS that starts in a callee procedure and ends in one of its caller procedure (or a transitive caller up the call chain).

3. MIPS that starts and ends in the same procedure, but the control flow inside the callee procedures called from intermediate nodes of the MIPS cannot be abstracted out.

Intuitively, to eliminate the data flow values along such MIPS, the data flow values along the corresponding inter-procedural control flow paths must be maintained separately across procedure boundaries. We believe this necessitates a context sensitive analysis. Moreover, we believe achieving scalability for such an analysis will be challenging as argued below.

A FPMFP computation inherently trades efficiency for achieving precision because it maintains and processes more information compared to a MFP computation. In Section 4.1, we describe a series of optimizations that we employ to make the FPMFP computation efficient and scalable. Additionally, if we have to handle all inter-procedural MIPS, then the maximum number of pairs at a program point will be in terms of total number of inter-procedural MIPS in the entire program plus the maximum number of intra-procedural MIPS in a procedure (unlike maximum number of MIPS in a procedure only as current FPMFP). We believe achieving scalability for such analysis to be more challenging than technically incorporating context sensitivity in FPMFP computation. We leave the possibility of such an analysis as part of future work.

3.5 Chapter Summary

In this chapter, we defined FPMFP computation for inter-procedural analysis using the functional approach. Here, restricting inter-procedural MIPS to balanced ones allowed us to treat them as intra-procedural MIPS by abstracting the control flow inside the procedures called from intermediate nodes of the MIPS. These MIPS are also detected by Bodik's approach. Lastly, we described the handling of call nodes in FPMFP computation. In the next chapter (Chapter 4), we explain optimizations that improve scalability of FPMFP computations.

Chapter 4

Optimizations and Complexity for Computing FPMFP Solutions

This chapter describes 3 optimizations that improve the scalability of a FPMFP computation (Section 4.1), followed by the description of the complexity of FPMFP computation (Section 4.2).

4.1 Optimizations for Scalability of FPMFP Solutions

First, we explicate the relation between efficiency of a FPMFP computation and the number of pairs in the FPMFP computation. Next, we propose ideas to reduce the number of pairs. Our empirical data shows the reduction obtained is significant.

As evident from equations 2.11 to 2.17, the efficiency of a FPMFP computation is inversely proportional to the number of pairs. Theoretically, the number of pairs needed is exponential in the number of MIPS in $\mathcal{U}$ (set of all MIPS) because a different pair is created corresponding to each subset of $\mathcal{U}$. Nevertheless, at any program point the number of pairs that are evaluated is bounded by $|\mathcal{U}| + 1$ (explained in Section 4.2). Below, we describe the ideas to reduce the number of pairs by merging them (Sections 4.1.1 and 4.1.2), and ignoring the ones that have no effect on the final FPMFP solution (Section 4.1.3). These optimizations are shown in Figure 4.1.

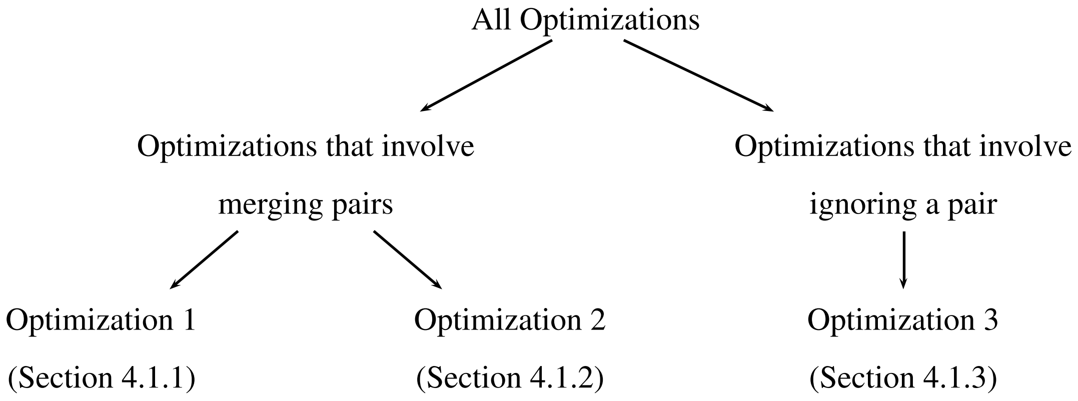


Figure 4.1: Optimizations for scalability of a FPMFP computation. The leaf nodes label a particular optimization, and rest of the nodes describe the type of the optimization.

4.1.1 Optimization 1: Merging Pairs Containing MIPS with Same End Edges

We now describe an optimization that allows us to merge some of the pairs present at a program point.

We observed that a large class of MIPS found in our benchmarks were satisfying the following property.

> $\mathcal{P}$: the condition on the end edge of a MIPS μ evaluates to *false* at the start edge of μ, and the variables present in the condition are not modified at the intermediate nodes of μ.

We observe the following about MIPS that satisfy $\mathcal{P}$.

Observation 4.1. In a FPMFP computation, we do not need to distinguish between two MIPS if both satisfy $\mathcal{P}$ and have the same end edge, because the condition on the end edge does not hold at the starts of these MIPS and data flow values associated with them are blocked at the same end edge. Hence, the data flow values associated with them can be merged.

We support Observation 4.1 with a proof in Section 5.4. Observation 4.1 allows us to merge pairs at a program point as described below.

Optimization 1. We merge two pairs $\langle \mathcal{M}, d \rangle$ and $\langle \mathcal{M}', d' \rangle$ into a single pair $\langle \mathcal{M} \cup \mathcal{M}', d \sqcap d' \rangle$

		Sets of MIPS				FPMFP
		$\{\}$	$\{\mu_1\}$	$\{\mu_2\}$	$\{\mu_1, \mu_2\}$	$\sqcap$
Edges	e_3	$z \mapsto [1,1]$	$\top$	$\top$	$\top$	$z \mapsto [1,1]$
	e_5	$\top$	$z \mapsto [0,0]$	$\top$	$\top$	$z \mapsto [0,0]$
	e_6	$\top$	$\top$	$z \mapsto [2,2]$	$\top$	$z \mapsto [2,2]$
	e_7	$z \mapsto [1,1]$	$z \mapsto [0,0]$	$z \mapsto [2,2]$	$\top$	$z \mapsto [0,2]$
	e_9	$z \mapsto [1,1]$	$\top$	$\top$	$\top$	$z \mapsto [1,1]$

Table 4.1: The FPMFP solution for example in Figure 4.2a. The solution computes value ranges for variable z. $\top = z \rightarrow [+\infty, -\infty]$.

iff the sets $\mathcal{M}$ and $\mathcal{M}'$ are *end edge equivalent* (denoted by $\overset{\textit{end}}{=}$) as defined below:

$$(\mathcal{M} \overset{\textit{end}}{=} \mathcal{M}') \Leftrightarrow (\bigcup_{\mu \in \mathcal{M}} \textit{end}(\mu) = \bigcup_{\mu' \in \mathcal{M}'} \textit{end}(\mu')) \tag{4.1}$$

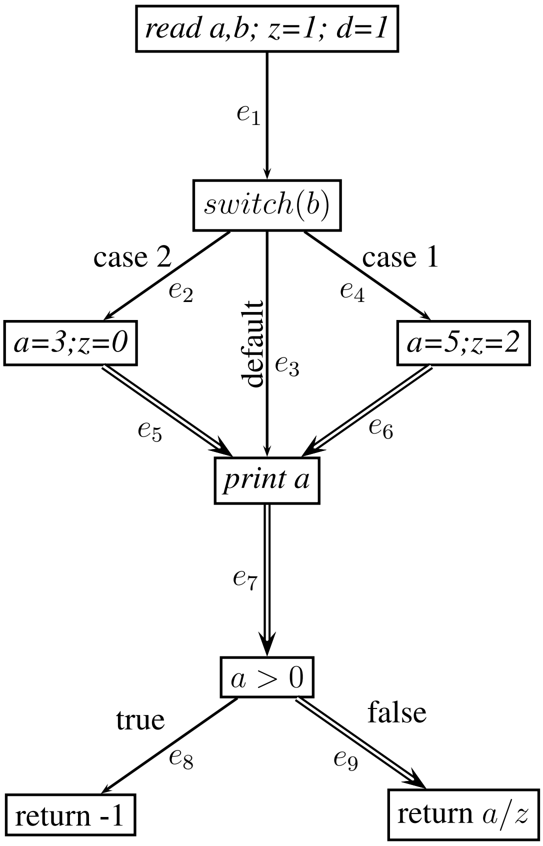
The FPMFP computation for the example in Figure 4.2 is shown in Table 4.1. Here, four pairs are computed at each edge corresponding to each subset of $\{\mu_1, \mu_2\}$. Observe that MIPS μ_1, μ_2 have the same end edge e_9 and hence the following sets are end edge equivalent: $\{\mu_1\}$, $\{\mu_2\}$, $\{\mu_1, \mu_2\}$. Hence, we optimize the FPMFP computation by merging these pairs as shown in Figure 4.2b, where only two pairs are maintained at each edge. Note that the corresponding FPMFP solution is identical to that in Table 4.1.

4.1.2 Optimization 2: Removing Duplicate Data Flow Values Across Pairs

In this section, we explain how the same data flow values can appear in more than one pairs at a program point. Further, we propose an idea to remove this duplication of values while retaining the precision and soundness of the FPMFP solution. This optimization has helped us to improve the scalability of our approach to 150KLOC when the timeout was kept at 10K seconds, whereas without this optimization scalability was 73KLOC with same timeout.

We observe the following property of a data flow analysis.

Observation 4.2. A data flow value d may reach a program point from more than one CFPs. In such case, if one of the CFPs is a feasible CFP and other CFPs are infeasible then blocking the copies of d reaching along infeasible CFPs does not lead to precision improvement.



(a)

Paths and MIPS:

- MIPS $\mu_1 : e_5 \rightarrow e_7 \rightarrow e_9$
- MIPS $\mu_2 : e_6 \rightarrow e_7 \rightarrow e_9$
- $\{\mu_1\} \overset{end}{=} \{\mu_2\} \overset{end}{=} \{\mu_1, \mu_2\}$

The FPMFP solution: the sets $\{\mu_1\}$, $\{\mu_2\}$, and $\{\mu_1, \mu_2\}$ are end edge equivalent hence the corresponding data flow values are associated with $\{\mu_1, \mu_2\}$ only. The solution computes value ranges for z. $\top = z \rightarrow [+\infty, -\infty]$.

		Sets of MIPS		FPMFP
		$\{\}$	$\{\mu_1, \mu_2\}$	$\sqcap$
Edges	e_3	$z \mapsto [1,1]$	$\top$	$z \mapsto [1,1]$
	e_5	$\top$	$z \mapsto [0,0]$	$z \mapsto [0,0]$
	e_6	$\top$	$z \mapsto [2,2]$	$z \mapsto [2,2]$
	e_7	$z \mapsto [1,1]$	$z \mapsto [0,2]$	$z \mapsto [0,2]$
	e_9	$z \mapsto [1,1]$	$\top$	$z \mapsto [1,1]$

(b)

Figure 4.2: Illustrating MIPS that have same end edge

		Sets of MIPS		FPMFP
		$\{\}$	$\{\mu_1\}$	$\sqcap$
Edges	e_3	$\top$	$d \mapsto [1,1]$	$d \mapsto [1,1]$
	e_5	$d \mapsto [1,1]$	$\top$	$d \mapsto [1,1]$
	e_6	$d \mapsto [1,1]$	$\top$	$d \mapsto [1,1]$
	e_7	$d \mapsto [1,1]$	$\top$	$d \mapsto [1,1]$

Table 4.2: Removing duplication of data flow values across pairs. $\top = d \rightarrow [+\infty, -\infty]$.

In line with observation 1, in a FPMFP computation at an edge e, two pairs can contain the same data flow value d if d reaches e along two or more CFPs (because in this case d may flow through two different MIPS which are not related by CPO relation). In such case, if one of the CFP is a feasible CFP then the two pairs should be merged into one (because keeping them separate does not increase the precision but increases the analysis cost). We explain this in Example 4.1 below.

Example 4.1. For example in Figure 4.3, the data flow value $d \mapsto [1,1]$ flows through MIPS μ hence it is associated with $\{\mu\}$ at edges in μ i.e., e_3, e_5, e_7. Next, the value associated with $\{\mu\}$ is blocked at the end edge e_7 of μ, because this value is reaching along the infeasible CFP that contains μ i.e., $e_0 \rightarrow e_1 \rightarrow e_3 \rightarrow e_5 \rightarrow e_7$.

On the other hand, $d \mapsto [1,1]$ also reaches e_7 along the following CFP that is not infeasible $\sigma : e_0 \rightarrow e_2 \rightarrow e_4 \rightarrow e_6 \rightarrow e_7$, hence $d \mapsto [1,1]$ is also associated with $\{\}$. Effectively, $d \mapsto [1,1]$ is included at e_7 in the FPMFP solution.

In such case, we can get the same FPMFP solution without associating $d \mapsto [1,1]$ with $\{\mu\}$ at e_5, e_7 as shown in Table 4.2. Observe that the corresponding FPMFP solution is identical to that in Figure 4.3.

We use the following general criteria to treat the pairs that contain the same data flow value at a program point.

> If a data flow value d reaches the end edge of a MIPS μ along a feasible CFP then d need not be associated with μ, because even if d is blocked within μ, it reaches along the feasible CFP and hence its association with μ is redundant.

To find out if a data flow value reaches the end edge of a MIPS along a feasible CFP, we define the *contains-suffix-of* relation below.

Definition 4.1. (Contains-suffix-of (CSO)) For a MIPS μ and an edge e in μ, let $\textit{suffix}(\mu, e)$ be the sub-segment of μ from e to $\textit{end}(\mu)$. Then, we say that a MIPS μ_1 contains-suffix-of a MIPS μ_2 at e, iff μ_1 contains $\textit{suffix}(\mu_2, e)$.

$$\textit{cso}_e(\mu_1) = \{\mu_2 \mid \mu_2 \in \mathcal{U},\ \mu_1 \text{ contains } \textit{suffix}(\mu_2, e)\} \tag{4.2}$$

Observe that CSO is a reflexive relation, and is a dual of the CPO relation defined in Section 2.3.1 in the following sense: at an edge e, μ_1 CPO μ_2 means μ_1 and μ_2 follow the same CFP till edge e; whereas, μ_1 CSO μ_2 means μ_1 and μ_2 follow the same CFP after edge e (they may or may not follow the same CFP before e).

In a FPMFP computation, the MIPS that are related by CSO relation satisfy the following properties. Let two pairs $p_1 : \langle\{\mu\}, d\rangle$ and $p_2 : \langle\{\mu'\}, d\rangle$ contain the same data flow value d at an edge e, and d is not killed along MIPS μ and μ' then

- if μ CSO μ' at e, then d is blocked at or before reaching the end edge of μ in both the pairs p_1 and p_2 (because after e, μ and μ' follow the same CFP on which μ' ends at or before the end of μ).

Example 4.2. For example in Figure 4.4a, μ_1 CSO μ_2 at e_5. Consequently, in the FPMFP computation, $l \mapsto [0, 0]$ is blocked at e_6 in pair $\langle\{\mu_2\}, l \mapsto [0, 0]\rangle$, and is blocked at e_7 in pair $\langle\{\mu_1\}, l \mapsto [0, 0]\rangle$. Thus $l \mapsto [0, 0]$ does not reach e_7.

- if μ does not contain suffix of μ' at e then d reaches the end edge of μ along at least one feasible CFP. In particular, this feasible CFP is obtained by extending the feasible path —along which pair p_2 reaches edge e —on suffix(μ,e) (proof of this is given in Section 5.4). Hence the association of d with μ can be discarded.

Example 4.3. For example in Figure 4.4a, μ_2 does not contain suffix of μ_1. Further, $l \mapsto [0, 0]$ flows through μ_1 and μ_2 and is blocked at the corresponding end edges as shown in Figure 4.4b. However, $l \mapsto [0, 0]$ is included in the FPMFP solution at the end edge e_6 of μ_2 because it reaches along the following feasible CFP: $e_0 \rightarrow e_3 \rightarrow$

$e_5 \to e_6 \to e_9$.

In general, if two pairs contain the same data flow value d, then the association of d with a MIPS μ in one pair can be discarded if there is no MIPS μ' in other pair such that μ CSO μ'. Based on this, we now formally describe the following criteria to merge two pairs that have the same data flow value.

Optimization 2. At an edge e, we shift the common data flow value d from two pairs $\langle \mathcal{M}_1, d\rangle$ and $\langle \mathcal{M}_2, d\rangle$ to a single pair $\langle \mathcal{M}_3, d\rangle$ where $\mathcal{M}_3$ contains only 1) those MIPS in $\mathcal{M}_1$ that contains suffix of at least one MIPS in $\mathcal{M}_2$, and 2) those MIPS in $\mathcal{M}_2$ that contain suffix of at least one MIPS in $\mathcal{M}_1$ i.e.,

$$\begin{aligned}\mathcal{M}_3 =&\{\mu_1 \mid \mu_1 \in \mathcal{M}_1,\ \exists \mu_2 \in \mathcal{M}_2.\, \mu_1 \text{ CSO } \mu_2 \text{ at } e\}\\ &\cup\\ &\{\mu_2 \mid \mu_2 \in \mathcal{M}_2,\ \exists \mu_1 \in \mathcal{M}_1.\, \mu_2 \text{ CSO } \mu_1 \text{ at } e\}\end{aligned}$$

Example 4.4. For example in Figure 4.4a, at edges e_5 and e_6 μ_1 CSO μ_2, hence we associate $l \mapsto [0,0]$ with μ_1 only as shown in Figure 4.4c. Observe the corresponding FPMFP solution is identical to the one in Figure 4.4b.

Example 4.5. For example in Figure 4.5a, the data flow value $l \mapsto [0,0]$ flows though two MIPS μ_1 and μ_2. Further neither μ_1 CSO μ_2 nor μ_2 CSO μ_1 is true, hence $l \mapsto [0,0]$ reaches *end*(μ_1) and *end*(μ_2) along the following feasible CFPs σ_1 and σ_2 respectively, $\sigma_1 : e_0 \to e_1 \to e_3 \to e_4 \to e_7$, $\sigma_2 : e_0 \to e_2 \to e_4 \to e_5$. Hence, $l \mapsto [0,0]$ is included in the FPMFP solution at edges e_7 and e_5 as shown in Figure 4.5b.

We obtain the same FPMFP solution without associating $l \mapsto [0,0]$ with μ_1 or μ_2 as shown in Figure 4.5c.

4.1.3 Optimization 3: Removing Pairs Containing $\top$ Data Flow Value

In a pair $\langle \mathcal{M}, d\rangle$ the data flow value d remains $\top$ at edges that are outside a MIPS in $\mathcal{M}$. We use the following observation to eliminate such pairs.

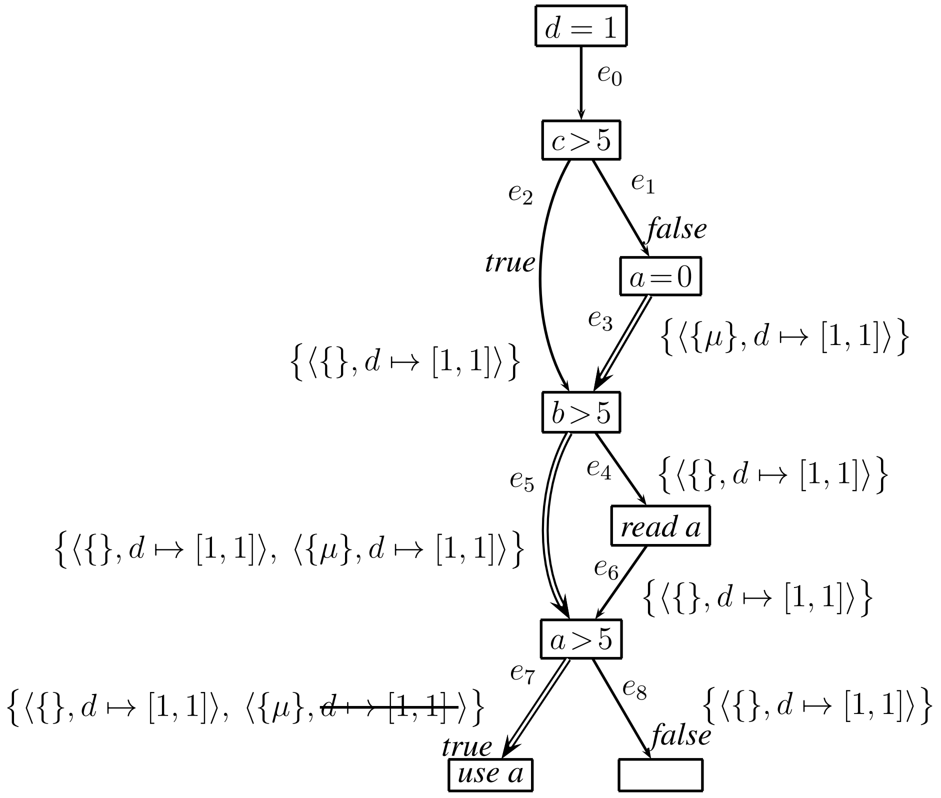

$d = 1$
e_0
$c > 5$
e_2
e_1
false
true
$a = 0$
e_3
$\{\langle\{\mu\}, d \mapsto [1,1]\rangle\}$
$\{\langle\{\}, d \mapsto [1,1]\rangle\}$
$b > 5$
e_5
e_4
$\{\langle\{\}, d \mapsto [1,1]\rangle\}$
read a
$\{\langle\{\}, d \mapsto [1,1]\rangle,\ \langle\{\mu\}, d \mapsto [1,1]\rangle\}$
e_6
$\{\langle\{\}, d \mapsto [1,1]\rangle\}$
$a > 5$
e_7
e_8
$\{\langle\{\}, d \mapsto [1,1]\rangle,\ \langle\{\mu\}, \cancel{d \mapsto [1,1]}\rangle\}$
$\{\langle\{\}, d \mapsto [1,1]\rangle\}$
true
false
use a

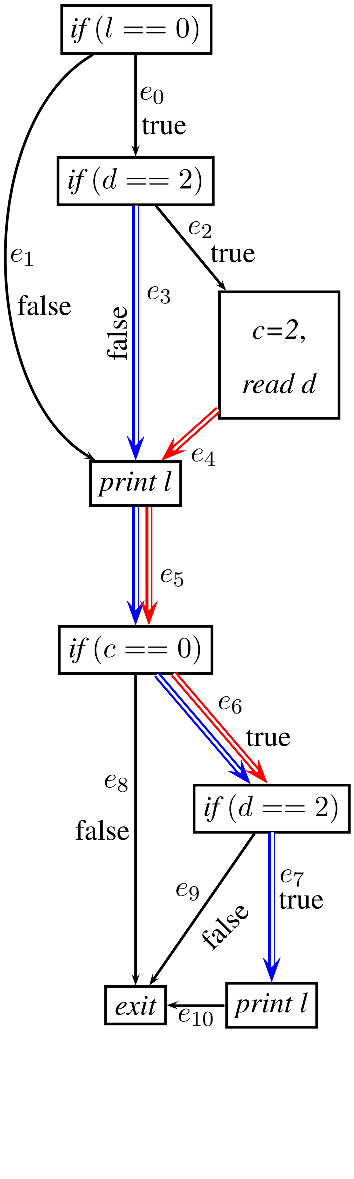
Figure 4.3: An example illustrating a FPMFP computation. The path segment $\mu : e_3 \rightarrow e_5 \rightarrow e_7$ is a MIPS. The analysis computes value ranges for variable d. For brevity, we have not shown pairs $\langle \mathcal{M}, x \rangle$ where $x = \top = d \rightarrow [+\infty, -\infty]$.

Observation 4.3. Ignoring the pairs that contain $\top$ as data flow value does not affect the precision or the soundness of the FPMFP solution, because the FPMFP solution is computed by taking meet of the data flow values in each pair at a program point.

Optimization 3. We eliminate the pairs that have $\top$ as the associated data flow value, if the total number of remaining pairs at the program point is more than one.

For example in Figure 4.2b, the pairs corresponding to $\{\}$ at edges e_5 and e_6 contain $\top$ value hence these pairs can be eliminated. Similarly, pairs corresponding to $\{\mu_1, \mu_2\}$ at edges e_3 and e_9 can be eliminated.

After Optimization 3 the node and edge flow functions (Equations 2.14 and 2.17) remain same except that the meet operator (Equation 2.15) now assumes absence of a pair corresponding to some $\mathcal{M} \subseteq \mathcal{U}$ in the input sets as presence of a pair $\langle \mathcal{M}, \top \rangle$.



(a)

Paths and MIPS:

- MIPS $\mu_1 : e_3 \to e_5 \to e_6 \to e_7$(marked in blue)
- *start*$(\mu_1) = \{e_3\}$, *inner*$(\mu_1) = \{e_5, e_6\}$, *end*$(\mu_1) = \{e_7\}$
- MIPS $\mu_2 : e_4 \to e_5 \to e_6$(marked in red)
- *start*$(\mu_2) = \{e_4\}$, *inner*$(\mu_2) = \{e_5\}$, *end*$(\mu_2) = \{e_6\}$
- At e_5 and e_6, μ_1 CSO μ_2

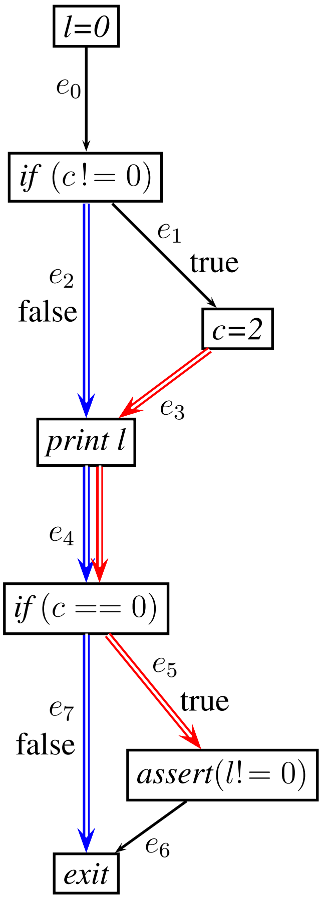
		Sets of MIPS				FPMFP
		$\{\}$	$\{\mu_1\}$	$\{\mu_2\}$	$\{\mu_1, \mu_2\}$	$\sqcap$
Edges	e_1	$l \mapsto [1, \infty]$	$\top$	$\top$	$\top$	$l \mapsto [1, \infty]$
	e_3	$\top$	$l \mapsto [0, 0]$	$\top$	$\top$	$l \mapsto [0, 0]$
	e_4	$\top$	$\top$	$l \mapsto [0, 0]$	$\top$	$l \mapsto [0, 0]$
	e_5	$l \mapsto [1, \infty]$	$l \mapsto [0, 0]$	$l \mapsto [0, 0]$	$\top$	$l \mapsto [0, \infty]$
	e_6	$l \mapsto [1, \infty]$	$l \mapsto [0, 0]$	~~$l \mapsto [0, 0]$~~	$\top$	$l \mapsto [0, \infty]$
	e_7	$l \mapsto [1, \infty]$	~~$l \mapsto [0, 0]$~~	$\top$	$\top$	$l \mapsto [1, \infty]$

(b)

		Sets of MIPS				FPMFP
		$\{\}$	$\{\mu_1\}$	$\{\mu_2\}$	$\{\mu_1, \mu_2\}$	$\sqcap$
Edges	e_1	$l \mapsto [1, \infty]$	$\top$	$\top$	$\top$	$l \mapsto [1, \infty]$
	e_3	$\top$	$l \mapsto [0, 0]$	$\top$	$\top$	$l \mapsto [0, 0]$
	e_4	$\top$	$\top$	$l \mapsto [0, 0]$	$\top$	$l \mapsto [0, 0]$
	e_5	$l \mapsto [1, \infty]$	$l \mapsto [0, 0]$	$\top$	$\top$	$l \mapsto [0, \infty]$
	e_6	$l \mapsto [1, \infty]$	$l \mapsto [0, 0]$	$\top$	$\top$	$l \mapsto [0, \infty]$
	e_7	$l \mapsto [1, \infty]$	~~$l \mapsto [0, 0]$~~	$\top$	$\top$	$l \mapsto [1, \infty]$

(c)

Figure 4.4: Illustrating pairs that have the same data flow value and have contains-suffix-of relation between constituent MIPS. The analysis computes value ranges for unsigned int variable l. $\top = l \to [+\infty, -\infty]$.



(a)

Paths and MIPS:

- MIPS $\mu_1 : e_2 \rightarrow e_4 \rightarrow e_7$ (marked in blue)
- MIPS $\mu_2 : e_3 \rightarrow e_4 \rightarrow e_5$ (marked in red)
- $\mathit{start}(\mu_1) = e_2$
- $\mathit{start}(\mu_2) = e_3$
- $\mathit{inner}(\mu_1) = \mathit{inner}(\mu_2) = e_4$
- $\mathit{end}(\mu_1) = e_7, \mathit{end}(\mu_2) = e_5$

		Sets of MIPS				FPMFP
		$\{\}$	$\{\mu_1\}$	$\{\mu_2\}$	$\{\mu_1, \mu_2\}$	$\sqcap$
Edges	e_2	$\top$	$l \mapsto [0,0]$	$\top$	$\top$	$l \mapsto [0,0]$
	e_3	$\top$	$\top$	$l \mapsto [0,0]$	$\top$	$l \mapsto [0,0]$
	e_4	$\top$	$l \mapsto [0,0]$	$l \mapsto [0,0]$	$\top$	$l \mapsto [0,0]$
	e_5	$l \mapsto [0,0]$	$\top$	~~$l \mapsto [0,0]$~~	$\top$	$l \mapsto [0,0]$
	e_7	$l \mapsto [0,0]$	~~$l \mapsto [0,0]$~~	$\top$	$\top$	$l \mapsto [0,0]$

(b)

		Sets of MIPS				FPMFP
		$\{\}$	$\{\mu_1\}$	$\{\mu_2\}$	$\{\mu_1, \mu_2\}$	$\sqcap$
Edges	e_2	$l \mapsto [0,0]$	$\top$	$\top$	$\top$	$l \mapsto [0,0]$
	e_3	$l \mapsto [0,0]$	$\top$	$\top$	$\top$	$l \mapsto [0,0]$
	e_4	$l \mapsto [0,0]$	$\top$	$\top$	$\top$	$l \mapsto [0,0]$
	e_5	$l \mapsto [0,0]$	$\top$	$\top$	$\top$	$l \mapsto [0,0]$
	e_7	$l \mapsto [0,0]$	$\top$	$\top$	$\top$	$l \mapsto [0,0]$

(c)

Figure 4.5: Illustrating pairs that have the same data flow value but do not have CSO relation between constituent MIPS. $l \mapsto [0,0]$ reaches $\mathit{end}(\mu_1)$ and $\mathit{end}(\mu_2)$ along the following feasible CFPs σ_1 and σ_2 respectively, $\sigma_1 : e_0 \rightarrow e_1 \rightarrow e_3 \rightarrow e_4 \rightarrow e_7$, $\sigma_2 : e_0 \rightarrow e_2 \rightarrow e_4 \rightarrow e_5$. The analysis computes value ranges for variable l. $\top = l \rightarrow [+\infty, -\infty]$.

4.2 Complexity Analysis of FPMFP

The cost of the FPMFP computation is equivalent to computing $k + 1$ parallel MFP solutions, where k is bounded by the maximum number of pairs at a program point that do not contain $\top$ as the associated data flow value.

At an edge e, the maximum number of pairs that do not contain $\top$ value is same as the maximum number of distinct sets given by $\mathit{cpo}_e(\mu)$ over all MIPS μ, plus 1 for the pair corresponding to $\{\}$. In worst case, $\mathit{cpo}_e(\mu)$ set for each μ is distinct. Hence the total number of such pairs (i.e., k value from previous paragraph) is bounded by the number of MIPS in the CFG. In practice, we found the k value to be much smaller where the FPMFP computation time was $2.9\times$ of the MFP computation time on average.

4.3 Chapter Summary

In this chapter, we improved the practical applicability of FPMFP solutions by proposing three optimizations that improve the scalability of FPMFP solutions. These optimizations arise from observations that anticipate the potential of precision gain in FPMFP and discard the distinctions that do not lead to any gain or merge the pairs that do not affect the outcome.

In the next chapter (Chapter 5), we prove that FPMFP solution is a sound approximation of the corresponding MOFP solution of a data flow analysis. Additionally, we prove that the optimizations for scalability of FPMFP solutions retain the soundness and precision of FPMFP solutions.

Chapter 5

Proof of Soundness of FPMFP Solutions

This chapter proves the soundness of FPMFP solutions in the following way. First we formally define the MOFP solution of a data flow analysis (Section 5.2), and prove that FPMFP solution is a sound approximation of the corresponding MOFP solution. Next, we prove that the optimizations introduced in Chapter 4 for scalability of FPMFP solutions retain the soundness and precision of the solution.

5.1 Overview of the Soundness Proof

It has been proved that a MFP solution is a sound-approximation of the MOP solution [31]. The proof showed that a data flow value computed by a MFP solution at program point p is a sound-approximation of the data flow values reaching p along all CFPs. On similar lines, we first define the meet over feasible paths solution of a data flow problem and prove that FPMFP solution is a sound-approximation of the MOFP solution.

More specifically, we show that the data flow values associated with each pair at a program point p in FPMFP computation represents a sound approximation of data flow values reaching along a unique subset of feasible CFPs at p. Additionally, the meet of data flow values across all pairs at p gives a sound approximation of the data flow values reaching along all feasible CFPs at p.

5.2 Definitions

5.2.1 Data Flow Value at an Edge

Definition 5.1. For an edge $e : m \to n$, we refer the data flow value $\overline{g}_e(\overline{\mathit{Out}}_m)$ obtained by applying the edge flow function $\overline{g}_e$ to the data flow value $\overline{\mathit{Out}}_m$ as "the data flow value at e".

5.2.2 CFPs that Reach a Node

Definition 5.2. Recall that all CFPs referred in our analysis are intra-procedural. We refer CFPs that begin at start node of a CFG and end at a particular node n as "CFPs that reach n".

Let $\mathcal{U}$ be the set of all MIPS that are input to a FPMFP computation, then we create the following classes of CFPs that reach a node n; these CFPs are illustrated in Figure 5.1.

- *paths*(n) is the set of all CFPs that reach node n. This includes feasible as well as infeasible CFPs that reach n.

- *mpaths*(n) is the set of all CFPs that reach node n and also contain at least one MIPS from $\mathcal{U}$ (*mpaths*(n) is a subset of *paths*(n)). All CFPs in *mpaths*(n) are infeasible CFPs because they contain a MIPS.

- *mpaths*$^c(n)$ is the set of all CFPs that reach node n and do not contain any MIPS from $\mathcal{U}$. *mpaths*$^c(n)$ is the set complement of *mpaths*(n) i.e., *mpaths*$^c(n) =$ *paths*$(n) -$ *mpaths*(n).

- *fpaths*(n) is the set of all feasible CFPs that reach node n. *fpaths*(n) is a subset of *mpaths*$^c(n)$.

5.2.3 CFPs that Reach an Edge

Definition 5.3. We refer CFPs that begin at start node of a CFG and end at a particular edge e as "CFPs that reach e". These are the same CFPs that reach the destination node of e and have e as the last edge.

We create the following classes of CFPs that reach an edge e namely *paths*(e), *mpaths*(e), *mpaths*$^c(e)$, *fpaths*(e). These classes are defined similar to the corresponding classes of CFPs

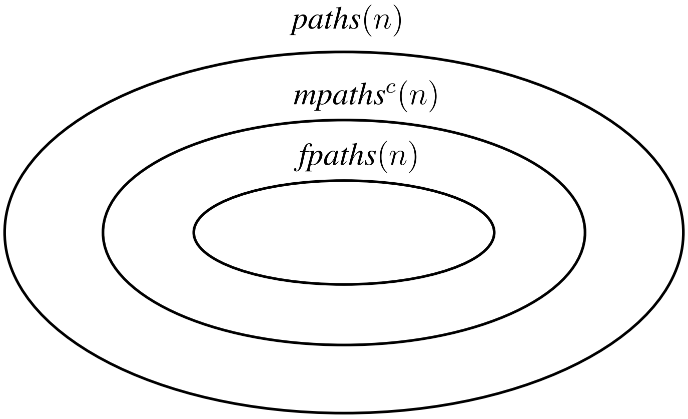


Figure 5.1: Paths that reach node n: $\mathit{paths}(n) \supseteq \mathit{mpaths}^c(n) \supseteq \mathit{fpaths}(n)$

that reach a node n. Also, if n is the destination node of an edge e then the following holds.

$$\begin{aligned} \mathit{paths}(e) &\subseteq \mathit{paths}(n) \\ \mathit{mpaths}(e) &\subseteq \mathit{mpaths}(n) \\ \mathit{mpaths}^c(e) &\subseteq \mathit{mpaths}^c(n) \\ \mathit{fpaths}(e) &\subseteq \mathit{fpaths}(n) \end{aligned}$$

5.2.4 Contains-prefix-of Relation Between a CFP and a MIPS

We defined the notion of contains-prefix-of for a MIPS in Chapter 2 (Definition 2.1), we now extend this to CFPs as follows.

Definition 5.4. For a CFP σ that reaches an edge e, $\mathit{cpo}_e(\sigma)$ contains all MIPS μ such that $\mathit{prefix}(\mu, e)$ is contained in σ i.e.,

$$\mathit{cpo}_e(\sigma) = \{\mu \mid \mu \in \mathcal{U},\ \sigma \text{ contains } \mathit{prefix}(\mu, e)\} \tag{5.1}$$

Definition 5.5. Let $\sigma.e'$ be a path obtained by extending a path σ that reaches an edge e to its successor edge e'. Then $\mathit{cpo}_{e'}(\sigma.e')$ precisely contains two types of MIPS: 1) MIPS present in $\mathit{cpo}_e(\sigma)$ which contain edge e', and 2) MIPS that start at e' i.e.,

$$\text{Given} \quad \sigma \in \mathit{mpaths}^c(e) \wedge\ e \text{ is predecessor edge of } e' \text{ then}$$

$$\mathit{cpo}_{e'}(\sigma.e') = \big\{\mu \mid (\mu \in \mathit{cpo}_e(\sigma), e' \in \mathit{edges}(\mu)) \vee e' \in \mathit{start}(\mu)\big\} \tag{5.2}$$

$$= \mathit{ext}(\mathit{cpo}_e(\sigma),\ e') \quad \ldots \text{from } 2.2 \tag{5.3}$$

We now formally state the MOP and MOFP solutions (Section 5.2.5). Next, we prove that the FPMFP solution is an sound-approximation of MOFP solution (Section 5.3).

5.2.5 MOP and MOFP solution

Given a path $\sigma \in \mathit{paths}(n_k)$ consisting of nodes $(n_1, n_2, ..., n_{k-1}, n_k)$, let f_σ denote the composition of functions corresponding to the nodes in σ excluding the last node n_k i.e., $f_\sigma = f_{n_{k-1}} \circ \ldots \circ f_{n_1}$. Note that f_σ represents the effect of a path that reaches at IN of node n_k, hence it excludes f_{n_k}. However, f_σ includes the effect of all edges in σ including the edge from n_{k-1} to n_k. For simplicity, we assume the edge flow functions of the input MFP analysis are identity functions for all edges. Observe that if σ contains a single node then f_σ is identity function.

The MOP and MOFP solutions are represented by $\mathit{In}_n/\mathit{Out}_n$ for all nodes n. In particular, the MOP solution is considers the effect of all paths reaching node n and is obtained as follows

$$\mathit{In}_n = \bigsqcap_{\sigma \in \mathit{paths}(n)} f_\sigma(BI) \tag{5.4}$$

$$\mathit{Out}_n = \bigsqcap_{\sigma \in \mathit{paths}(n)} f_n \circ f_\sigma(BI) \tag{5.5}$$

On the other hand, the MOFP solution is obtained by taking the effect of only feasible paths reaching n (i.e., $\mathit{fpaths}(n)$) as follows.

$$\mathit{In}_n = \bigsqcap_{\sigma \in \mathit{fpaths}(n)} f_\sigma(BI) \tag{5.6}$$

$$\mathit{Out}_n = \bigsqcap_{\sigma \in \mathit{fpaths}(n)} f_n \circ f_\sigma(BI) \tag{5.7}$$

5.3 Proof of Soundness of FPMFP Solutions

Theorem 5.3.1. The FPMFP solution is a sound-approximation of the MOFP solution i.e., if $\mathcal{N}$ is the set of all nodes in the CFG of a procedure then

$$\text{Claim 1.} \qquad \forall n \in \mathcal{N}.\ \overline{\overline{\mathit{In}}}_n \sqsubseteq \bigsqcap_{\sigma \in \mathit{fpaths}(n)} f_\sigma(BI) \tag{5.8}$$

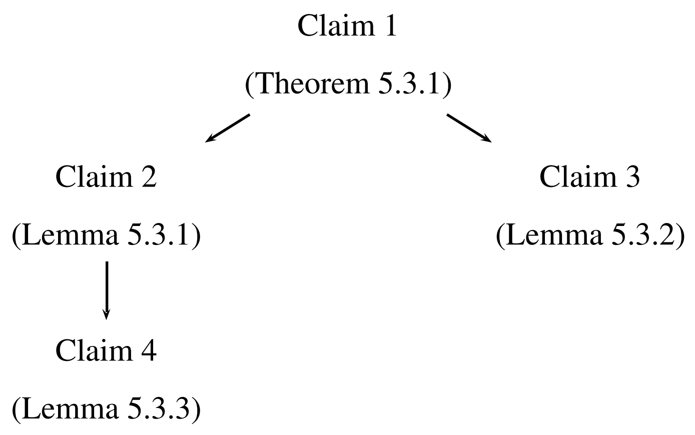


Figure 5.2: Proof Outline. The claim at the root of the tree (claim 1) is the main proof obligation for soundness of FPMFP. A claim at each node is derived using the claims at its child nodes.

Proof. The proof outline is illustrated in Figure 5.2. Below, we state Claim 2 and Claim 3 which together prove Claim 1 i.e.,

$$\text{Claim 2} \wedge \text{Claim 3} \implies \text{Claim 1} \tag{5.9}$$

where,

- Claim 2: at any node n, $\overline{\overline{\mathsf{In}}}_n$ is a sound-approximation of the meet of data flow values reaching along paths in *mpaths*$^c(n)$ i.e.,

$$\text{Claim 2.} \qquad \forall n \in \mathcal{N}.\ \overline{\overline{\mathsf{In}}}_n \sqsubseteq \bigsqcap_{\sigma \in \mathit{mpaths}^c(n)} f_\sigma(BI) \tag{5.10}$$

- Claim 3: at any node n, the meet of data flow values reaching along paths in *mpaths*$^c(n)$ is a sound-approximation of the data flow values reaching along paths in *fpaths*(n) i.e.,

$$\text{Claim 3.} \qquad \forall n \in \mathcal{N}.\ \Big(\bigsqcap_{\sigma \in \mathit{mpaths}^c(n)} f_\sigma(BI)\Big) \sqsubseteq \Big(\bigsqcap_{\sigma \in \mathit{fpaths}(n)} f_\sigma(BI)\Big) \tag{5.11}$$

Claim 2 and Claim 3 together imply Claim 1 by transitivity of $\sqsubseteq$. We prove the Claim 2 and the Claim 3 in Lemma 5.3.1 and Lemma 5.3.2 respectively.

□

We first state Claim 4 that will be used to prove Claim 2.

In the FPMFP computation, the data flow value in each pair $\langle \mathcal{M}, d \rangle$ at an edge $e : m \to n$ is a sound-approximation of the meet of the data flow values reaching along each path σ where

$\sigma \in \mathit{mpaths}^c(e)$ and $\mathit{cpo}_e(\sigma) = \mathcal{M}$ i.e.,

$$\text{Claim 4.}\quad \langle \mathcal{M}, d\rangle \in \overline{g}_e(\overline{\mathit{Out}}_m) \implies d \sqsubseteq \Big(\bigsqcap_{\substack{\sigma \in \mathit{mpaths}^c(e),\\ \mathit{cpo}_e(\sigma) = \mathcal{M}}} f_\sigma(BI)\Big) \tag{5.12}$$

We prove Claim 4 in Lemma 5.3.3. We now show that Claim 2 follows from Claim 4 because for a node n that is not the *Start* node of the CFG, $\mathit{mpaths}^c(n)$ is obtained by union of $\mathit{mpaths}^c(e)$ corresponding to each edge e that is incident on n.

Lemma 5.3.1. At any node n in the CFG of a procedure the following holds

$$\text{Claim 2.}\quad \forall n \in \mathcal{N}.\ \overline{\overline{\mathit{In}}}_n \sqsubseteq \Big(\bigsqcap_{\sigma \in \mathit{mpaths}^c(n)} f_\sigma(BI)\Big) \tag{5.13}$$

Proof. We split the proof of Claim 2 in the following two cases which are mutually exclusive and exhaust all possibilities.

- Case 1: n is the *Start* node of the CFG. In this case, $\mathit{mpaths}^c(n)$ contains a single path containing the *Start* node only. Hence the following holds.

$$\Big(\bigsqcap_{\sigma \in \mathit{mpaths}^c(n)} f_\sigma(BI)\Big) = BI \tag{5.14}$$

$$\overline{\overline{\mathit{In}}}_n = BI \qquad \text{from } 2.11 \tag{5.15}$$

$$\overline{\overline{\mathit{In}}}_n \sqsubseteq \Big(\bigsqcap_{\sigma \in \mathit{mpaths}^c(n)} f_\sigma(BI)\Big) \qquad \text{from } 5.15,\ 5.14 \tag{5.16}$$

- Case 2: n is not the *Start* node of the CFG. In this case, the following holds.

$$\overline{\mathit{In}}_n = \overline{\bigsqcap_{m \in \mathit{pred}(n)}} \overline{g}_{m\to n}(\overline{\mathit{Out}}_m) \quad \ldots\text{from } 2.11 \tag{5.17}$$

$$\langle \mathcal{M}_1, d_1\rangle \in \overline{\mathit{In}}_n \implies d_1 \sqsubseteq \Big(\bigsqcap_{\substack{\langle \mathcal{M}_1, d_1'\rangle \in \overline{g}_{m\to n}(\overline{\mathit{Out}}_m),\\ m \in \mathit{pred}(n)}} d_1'\Big) \quad \ldots\text{from } 2.15 \tag{5.18}$$

$$\langle \mathcal{M}_1, d_1\rangle \in \overline{\mathit{In}}_n \implies d_1 \sqsubseteq \Big(\bigsqcap_{\substack{\sigma \in \mathit{mpaths}^c(n),\\ \mathit{cpo}_e(\sigma) = \mathcal{M}_1}} f_\sigma(BI)\Big) \quad \ldots\text{from } 5.3.3 \tag{5.19}$$

$$\Big(\bigsqcap_{\langle \mathcal{M}_1, d_1\rangle \in \overline{\mathit{In}}_n} d_1\Big) \sqsubseteq \Big(\bigsqcap_{\sigma \in \mathit{mpaths}^c(n)} f_\sigma(BI)\Big) \tag{5.20}$$

$$\overline{\overline{\mathit{In}}}_n \sqsubseteq \Big(\bigsqcap_{\sigma \in \mathit{mpaths}^c(n)} f_\sigma(BI)\Big) \tag{5.21}$$

Equations 5.16 and 5.21 prove the lemma.

□

Lemma 5.3.2. At any node n in the CFG of a procedure the following holds

$$\text{Claim 3.} \quad \forall n \in \mathcal{N}.\ \Big(\bigsqcap_{\sigma \in \mathit{mpaths}^c(n)} f_\sigma(BI)\Big) \sqsubseteq \Big(\bigsqcap_{\sigma \in \mathit{fpaths}(n)} f_\sigma(BI)\Big) \tag{5.22}$$

Proof. The following holds from the definition of $\mathit{mpaths}^c(n)$ and $\mathit{fpaths}(n)$

$$\forall n \in \mathcal{N}.\ \mathit{fpaths}(n) \subseteq \mathit{mpaths}^c(n) \qquad \ldots \text{from } 5.2 \tag{5.23}$$

Claim 3 trivially follows from 5.23.

□

The following observation states some properties of a path of length 1 starting at *Start* of a CFG. These properties are used for proving claim 4 subsequently.

Observation 5.1. Let $\sigma_1 : n_0 \xrightarrow{e} n_1$ be a path of length one that begins at *Start* of a procedure, and $\mathcal{U}$ be the set of all MIPS in the procedure, then σ_1 has the following properties.

1. e does not have a predecessor edge. Hence, e cannot be end edge of any MIPS i.e.,

$$\forall \mathcal{M} \subseteq \mathcal{U}.\ \neg \mathit{endof}(\mathcal{M}, e) \tag{5.24}$$

2. $\mathit{cpo}_e(\sigma_1)$ only contains MIPS that start at e i.e.,

$$\mathit{cpo}_e(\sigma_1) = \{\mu \mid \mu \in \mathcal{U},\ e \in \mathit{start}(\mu)\} \qquad \ldots \text{from } 5.1 \tag{5.25}$$

$$\forall \mathcal{M} \subseteq \mathcal{U}.\ \mathit{ext}(\mathcal{M}, e) = \{\mu \mid \mu \in \mathcal{U},\ e \in \mathit{start}(\mu)\} \qquad \ldots \text{from } 5.1 \tag{5.26}$$

$$\forall \mathcal{M} \subseteq \mathcal{U}.\ \mathit{ext}(\mathcal{M}, e) = \mathit{cpo}_e(\sigma_1) \tag{5.27}$$

$$(\mathcal{M}' \subseteq \mathcal{U}) \wedge (\mathcal{M}' \neq \mathit{cpo}_e(\sigma_1)) \implies (\forall \mathcal{M} \subseteq \mathcal{U}.\ \mathit{ext}(\mathcal{M}, e) \neq \mathcal{M}') \quad \ldots \text{from } 5.25, 5.27 \tag{5.28}$$

3. There exists only one path of length 1 that begins at *Start*

$$mpaths_1^c(e) = \{\sigma_1\} \tag{5.29}$$

$$\Big(\bigsqcap_{\sigma \in mpaths_1^c(e)} f_\sigma(BI)\Big) = f_{\sigma_1}(BI) \quad \ldots \text{from } 5.29 \tag{5.30}$$

$$= f_{n0}(BI) \quad \ldots f_{\sigma_1} = f_{n0} \tag{5.31}$$

4. Since n_0 is the Start node of the procedure, $\overline{\mathit{In}}_{n0}$ is the boundary value.

$$\overline{\mathit{In}}_{n0} = \overline{\mathit{BI}} = \big\{\langle\{\}, \mathit{BI}\rangle\big\} \cup \big\{\langle\mathcal{M}, \top\rangle \mid \mathcal{M} \neq \{\}, \mathcal{M} \subseteq \mathcal{U}\big\} \quad \ldots \text{from } 2.11 \tag{5.32}$$

$$\overline{f}_{n0}(\overline{\mathit{In}}_{n0}) = \big\{\langle\{\}, f_{n0}(\mathit{BI})\rangle\big\} \cup \big\{\langle\mathcal{M}, f_{n0}(\top)\rangle \mid \mathcal{M} \neq \{\}, \mathcal{M} \subseteq \mathcal{U}\big\} \quad \ldots \text{from } 2.12 \tag{5.33}$$

$$\overline{\mathit{Out}}_{n0} = \big\{\langle\{\}, f_{n0}(\mathit{BI})\rangle\big\} \cup \big\{\langle\mathcal{M}, f_{n0}(\top)\rangle \mid \mathcal{M} \neq \{\}, \mathcal{M} \subseteq \mathcal{U}\big\} \quad \ldots \text{from } 2.12 \tag{5.34}$$

Lemma 5.3.3. In the FPMFP computation, the data flow value in each pair $\langle\mathcal{M}, d\rangle$ at an edge $e : m \rightarrow n$ is a sound-approximation of the meet of data flow values reaching along each path σ where $\sigma \in$ *mpaths*$^c(e)$ and $cpo_e(\sigma) = \mathcal{M}$ i.e.,

$$\text{Claim 4.} \quad \langle\mathcal{M}, d\rangle \in \overline{g}_e(\overline{\mathit{Out}}_m) \implies d \sqsubseteq \Big(\bigsqcap_{\substack{\sigma \in mpaths^c(e), \\ cpo_e(\sigma) = \mathcal{M}}} f_\sigma(BI)\Big) \tag{5.35}$$

Proof.

We prove this by induction on the length of paths reaching from *Start* to the edge e. Let *mpaths*$_l^c(e)$ denote the paths of length l from *mpaths*$^c(e)$ (edges may be repeated in the path).

We re-write the proof obligation in the following equivalent form.

$$\forall l \geq 1.\ \langle\mathcal{M}, d\rangle \in \overline{g}_e(\overline{\mathit{Out}}_m) \implies d \sqsubseteq \Big(\bigsqcap_{\substack{\sigma \in mpaths_l^c(e), \\ cpo_e(\sigma) = \mathcal{M}}} f_\sigma(BI)\Big) \tag{5.36}$$

Basis. $l = 1$ There exists only one path of length 1 that begins from *Start* node. Let that path be $\sigma_1 : n_0 \xrightarrow{e} n_1$, where $n_0 =$ *Start*. Observe that e cannot be an inner or end edge of any

MIPS because e does not have a predecessor edge. Consequently, $\mathit{cpo}_e(\sigma_1)$ only contains MIPS that start at e. We have stated these properties in Observation 5.1.

Proof obligation for the base case (obtained by substituting $l = 1$ and $m = n_0$ in 5.36):

$$\langle \mathcal{M}', d' \rangle \in \overline{g}_e(\overline{\mathit{Out}}_{n0}) \implies d' \sqsubseteq \bigsqcap_{\substack{\sigma \in \mathit{mpaths}_1^c(e), \\ \mathit{cpo}_e(\sigma) = \mathcal{M}'}} f_\sigma(BI) \tag{5.37}$$

We split the proof of the base case in the following two cases which are mutually exclusive and exhaustive for each pair $\langle \mathcal{M}', d' \rangle$ in $\overline{g}_e(\overline{\mathit{Out}}_{n0})$.

- Case 1: There exists a path $\sigma \in \mathit{mpaths}_1^c(e)$ such that $\mathcal{M}' = \mathit{cpo}_e(\sigma)$. Since $\mathit{mpaths}_1^c(e)$ contains a single path σ_1, Case 1 becomes $\mathcal{M}' = \mathit{cpo}_e(\sigma_1)$.

$$\big(\langle \mathcal{M}', d' \rangle \in \overline{g}_e(\overline{\mathit{Out}}_{n0})\big) \wedge \big(\mathcal{M}' = \mathit{cpo}_e(\sigma_1)\big)$$

$$\implies d' = \bigsqcap_{\substack{\langle \mathcal{M}, d \rangle \in \overline{\mathit{Out}}_{n0}, \\ \mathit{ext}(\mathcal{M}, e) = \mathcal{M}'}} d \qquad \text{from } 2.17,\ 5.24 \tag{5.38}$$

$$\implies d' = \bigsqcap_{\substack{\langle \mathcal{M}, d \rangle \in \overline{\mathit{Out}}_{n0}, \\ \mathit{ext}(\mathcal{M}, e) = \mathit{cpo}_e(\sigma_1)}} d \qquad \mathcal{M}' = \mathit{cpo}_e(\sigma_1) \tag{5.39}$$

$$\implies d' = \bigsqcap_{\langle \mathcal{M}, d \rangle \in \overline{\mathit{Out}}_{n0}} d \qquad \text{from } 5.27 \tag{5.40}$$

$$\implies d' = f_{n0}(BI) \sqcap f_{n0}(\top) \qquad \text{from } 5.34 \tag{5.41}$$

$$\implies d' = f_{n0}(BI) \qquad f_{n0}(BI) \sqsubseteq f_{n0}(\top) \tag{5.42}$$

$$\implies d' = \bigsqcap_{\sigma \in \mathit{mpaths}_1^c(e)} f_\sigma(BI) \qquad \text{from 5.30 and 5.31} \tag{5.43}$$

$$\implies d' = \bigsqcap_{\substack{\sigma \in \mathit{mpaths}_1^c(e), \\ \mathit{cpo}_e(\sigma) = \mathit{cpo}_e(\sigma_1)}} f_\sigma(BI) \qquad \text{from } 5.29 \tag{5.44}$$

$$\implies d' = \bigsqcap_{\substack{\sigma \in \mathit{mpaths}_1^c(e),\\ \mathit{cpo}_e(\sigma) = \mathcal{M}'}} f_\sigma(BI) \qquad \mathcal{M}' = \mathit{cpo}_e(\sigma_1) \tag{5.45}$$

- Case 2: There does not exists a path $\sigma \in \mathit{mpaths}_1^c(e)$ such that $\mathcal{M}' = \mathit{cpo}_e(\sigma)$. Since $\mathit{mpaths}_1^c(e)$ contains a single path σ_1, Case 2 becomes $\mathcal{M}' \neq \mathit{cpo}_e(\sigma_1)$.

$$\big(\langle \mathcal{M}', d'\rangle \in \overline{g}_e(\overline{\mathit{Out}}_{n0})\big) \wedge \big(\mathcal{M}' \neq \mathit{cpo}_e(\sigma_1)\big)$$

$$\implies d' = \bigsqcap_{\substack{\langle \mathcal{M}, d\rangle \in \overline{\mathit{Out}}_{n0},\\ \mathit{ext}(\mathcal{M}, e) = \mathcal{M}'}} d \qquad \text{from } 2.17,\ 5.24 \tag{5.46}$$

$$\implies d' = \top \qquad \text{from } 5.28 \tag{5.47}$$

$$\implies d' = \bigsqcap_{\sigma \in \{\}} f_\sigma(BI) \qquad \bigsqcap_{\{\}} = \top \tag{5.48}$$

$$\text{Since } \{\sigma \mid \sigma \in \mathit{mpaths}_1^c(e), \mathit{cpo}_e(\sigma) \neq \mathit{cpo}_e(\sigma_1)\} = \Phi \qquad \text{from } 5.29 \tag{5.49}$$

$$\implies d' = \bigsqcap_{\substack{\sigma \in \mathit{mpaths}_1^c(e),\\ \mathit{cpo}_e(\sigma) = \mathcal{M}'}} f_\sigma(BI) \qquad \text{from } 5.29 \tag{5.50}$$

From Case 1 and 2, we get equation 5.37

Induction Hypothesis. Let n be any arbitrary node in the CFG of a procedure such that there exist a CFP of length $l-1$ reaching n, and n has at least one outgoing edge. We assume that Lemma 5.3.3 holds at each edge e' that is incident on n (of the form $m \xrightarrow{e'} n$) and is the last edge of the path of length $l-1$ reaching n i.e.,

$$\langle \mathcal{M}, d\rangle \in \overline{g}_{e'}(\overline{\mathit{Out}}_m) \implies d \sqsubseteq \bigsqcap_{\substack{\sigma \in \mathit{mpaths}_{l-1}^c(e'),\\ \mathit{cpo}_{e'}(\sigma) = \mathcal{M}}} f_\sigma(BI) \tag{5.51}$$

Inductive Step. We prove that Lemma 5.3.3 holds for any successor edge of e' w.r.t. paths of length l i.e., if $m \xrightarrow{e'} n \xrightarrow{e''} o$ then

$$\langle \mathcal{M}'', d'' \rangle \in \overline{g}_{e''}(\overline{\mathit{Out}}_n) \implies d'' \sqsubseteq \bigsqcap_{\substack{\sigma \in \mathit{mpaths}_l^c(e''),\\ \mathit{cpo}_{e''}(\sigma) = \mathcal{M}}} f_\sigma(BI) \tag{5.52}$$

Proof of the inductive step. We build the proof in two steps.

Let the function $\mathit{inEdges}(n)$ return the set of incoming edges of a node n in the CFG, then in step 1, we prove that the following holds at $\overline{\mathit{Out}}_n$.

$$\langle \mathcal{M}, d' \rangle \in \overline{\mathit{Out}}_n \implies d' \sqsubseteq \bigsqcap_{\substack{\sigma \in \mathit{mpaths}_{l-1}^c(e'),\\ e' \in \mathit{inEdges}(n),\\ \mathit{cpo}_{e'}(\sigma) = \mathcal{M}}} f_n \circ f_\sigma(BI) \tag{5.53}$$

In step 2, we prove that for any successor edge e'' of e', equation 5.52 holds.

Step 1:

$$\overline{\mathit{In}}_n = \overline{\bigsqcap}_{m \in pred(n)} \overline{g}_{m \to n}(\overline{\mathit{Out}}_m) \quad \ldots \text{from 2.11} \tag{5.54}$$

$$\langle \mathcal{M}, d \rangle \in \overline{\mathit{In}}_n \implies d \sqsubseteq \bigsqcap_{\substack{\langle \mathcal{M}, d' \rangle \in \overline{g}_{m \to n}(\overline{\mathit{Out}}_m),\\ (m \to n) \in \mathit{inEdges}(n)}} d' \quad \ldots \text{from } 5.54,\ 2.15 \tag{5.55}$$

$$\langle \mathcal{M}, d \rangle \in \overline{\mathit{In}}_n \implies d \sqsubseteq \bigsqcap_{\substack{\sigma \in \mathit{mpaths}_{l-1}^c(e'),\\ e' \in \mathit{inEdges}(n),\\ \mathit{cpo}_{e'}(\sigma) = \mathcal{M}}} f_\sigma(BI) \quad \ldots \text{from } 5.51 \tag{5.56}$$

since f_n is monotone

$$\langle \mathcal{M}, d \rangle \in \overline{\mathit{In}}_n \implies f_n(d) \sqsubseteq \bigsqcap_{\substack{\sigma \in \mathit{mpaths}_{l-1}^c(e'),\\ e' \in \mathit{inEdges}(n),\\ \mathit{cpo}_{e'}(\sigma) = \mathcal{M}}} f_n \circ f_\sigma(BI) \tag{5.57}$$

$$\langle \mathcal{M}, d' \rangle \in \overline{\mathit{Out}}_n \implies \exists \langle \mathcal{M}, d \rangle \in \overline{\mathit{In}}_n.\ f_n(d) = d' \quad \ldots \text{from 2.12} \tag{5.58}$$

$$\langle \mathcal{M}, d' \rangle \in \overline{\mathsf{Out}}_n \implies d' \sqsubseteq \bigsqcap_{\substack{\sigma \in \mathit{mpaths}^c_{l-1}(e'),\\ e' \in \mathit{inEdges}(n),\\ \mathit{cpo}_{e'}(\sigma) = \mathcal{M}}} f_n \circ f_\sigma(BI) \quad \ldots \text{from } 5.57,\ 5.58 \tag{5.59}$$

Step 2: We now prove that at any successor edge $e'' : n \rightarrow o$ of e' the following holds

$$\langle \mathcal{M}'', d'' \rangle \in \overline{g}_{e''}(\overline{\mathsf{Out}}_n) \implies d'' \sqsubseteq \big(\bigsqcap_{\substack{\sigma \in \mathit{mpaths}^c_l(e''),\\ \mathit{cpo}_{e''}(\sigma) = \mathcal{M}''}} f_\sigma(BI) \big) \tag{5.60}$$

We consider the following possibilities for any arbitrary pair $\langle \mathcal{M}'', d'' \rangle$ in $\overline{g}_{e''}(\overline{\mathsf{Out}}_n)$. These cases are mutually exclusive and exhaustive.

- Case A: e'' is end edge of some MIPS μ contained in $\mathcal{M}''$ i.e., $\mathit{endof}(\mathcal{M}'', e'')$=*true* In this case, we prove that equation 5.60 holds because the following is true.

$$d'' = \top \tag{5.61}$$

$$\big(\bigsqcap_{\substack{\sigma \in \mathit{mpaths}^c_l(e''),\\ \mathit{cpo}_{e''}(\sigma) = \mathcal{M}''}} f_\sigma(BI) \big) = \top \tag{5.62}$$

$d'' = \top$ follows from the definition of the edge flow function (equation 2.17) i.e.,

$$\langle \mathcal{M}'', d'' \rangle \in \overline{g}_{e''}(\overline{\mathsf{Out}}_n) \wedge \mathit{endof}(\mathcal{M}'', e'') \implies d'' = \top \quad \ldots \text{from } 2.17 \tag{5.63}$$

Equation 5.62 follows from the fact that there cannot exist a path σ' that satisfies the constraint $\sigma' \in \mathit{mpaths}^c(e'') \wedge \mathit{endof}(\mathit{cpo}_{e''}(\sigma'), e'')$ from the definition of $\mathit{mpaths}^c(e'')$.

Hence the following holds,

$$\big(\bigsqcap_{\substack{\sigma \in \mathit{mpaths}^c_l(e''),\\ \mathit{cpo}_{e''}(\sigma) = \mathcal{M}'',\\ \mathit{endof}(\mathcal{M}'', e'')}} f_\sigma(BI) \big) = \top \tag{5.64}$$

From 5.63 and 5.64 we get

$$\langle \mathcal{M}'', d'' \rangle \in \overline{g}_{e''}(\overline{\mathit{Out}}_n) \wedge \mathit{endof}(\mathcal{M}'', e'') \implies d'' \sqsubseteq \Big(\bigsqcap_{\substack{\sigma \in \mathit{mpaths}^c_l(e''),\\ \mathit{cpo}_{e''}(\sigma) = \mathcal{M}''}} f_\sigma(BI) \Big) \tag{5.65}$$

- Case B: e'' is not end edge of any MIPS in $\mathcal{M}''$ i.e., $\mathit{endof}(\mathcal{M}'', e'')$=*false*.

$$\begin{aligned}
&\langle \mathcal{M}'', d'' \rangle \in \overline{g}_{e''}(\overline{\mathit{Out}}_n) \wedge \neg \mathit{endof}(\mathcal{M}'', e'') \\
&\implies d'' = \bigsqcap_{\substack{\langle \mathcal{M}, d \rangle \in \overline{\mathit{Out}}_n,\\ \mathit{ext}(\mathcal{M}, e'') = \mathcal{M}''}} d \quad \ldots \text{from } 2.17 \\
&\implies d'' \sqsubseteq \Big(\bigsqcap_{\substack{\sigma \in \mathit{mpaths}^c_{l-1}(e'),\\ e' \in \mathit{inEdges}(n),\\ \mathit{cpo}_{e'}(\sigma) = \mathcal{M},\\ \mathit{ext}(\mathcal{M}, e'') = \mathcal{M}''}} f_n \circ f_\sigma(BI) \Big) \quad \ldots \text{from } 5.59 \\
&\implies d'' \sqsubseteq \Big(\bigsqcap_{\substack{\sigma \in \mathit{mpaths}^c_{l-1}(e'),\\ e' \in \mathit{inEdges}(n),\\ \mathit{cpo}_{e''}(\sigma.e'') = \mathcal{M}''}} f_{\sigma.e''}(BI) \Big) \quad \ldots \text{from definition } 5.5 \\
&\implies d'' \sqsubseteq \Big(\bigsqcap_{\substack{\sigma \in \mathit{mpaths}^c_l(e''),\\ \mathit{cpo}_{e''}(\sigma) = \mathcal{M}''}} f_\sigma(BI) \Big)
\end{aligned} \tag{5.66}$$

From case A and B, we get equation 5.52.

□

5.4 Proof of Soundness of the Optimizations of Scalability

In this section, we prove that the optimizations proposed in Section 4.1 retain the soundness and precision of the computed FPMFP solution. Optimization 1 and Optimization 2 are proved in Sections 5.4.1 and 5.4.2 respectively.

5.4.1 Proof for Optimization 1

Optimization 1 is applied iff the input MIPS satisfy the property $\mathcal{P}$ (Section 4.1.1) which is as follows.

> $\mathcal{P}$: for a MIPS $\mu : e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n$, if the condition on the end edge e_n is *cond*(e_n) then *cond*(e_n) evaluates to *false* at the start edge e_1 of μ and variables in *cond*(e_n) are not modified along μ.

Formally, if $[\![p]\!]_e$ represents the evaluation of a condition expression p at an edge e, and *modified*(p, μ) is the set of variables in p that are modified at intermediate nodes of a MIPS μ then

$$\text{“}\mu \text{ is a MIPS”} \iff \big(([\![\mathit{cond}(e_n)]\!]_{e_1} = \mathit{false}) \wedge (\mathit{modified}(\mathit{cond}(e_n), \mu) = \phi)\big) \tag{5.67}$$

We now prove that if two MIPS satisfy $\mathcal{P}$, and have the same end edge then they can be considered equivalent. In particular, we prove that if two MIPS have the same end edge e, then any CFP that extends from start edge of either MIPS till e by traversing edges in either MIPS is an infeasible path. Hence, the data flow values associated with such MIPS can be merged.

Theorem 5.4.1. If μ_1, μ_2 are two MIPS that have the same end edge e_n, and they intersect at some node x_k, then μ_3, μ_4—constructed as shown below using split and join of μ_1, μ_2 at the intersecting node—are also MIPS.

$$\begin{aligned}
\mu_1 :& \underline{x_1 \xrightarrow{ex1} x_2 \rightarrow \ldots \rightarrow x_k \rightarrow x_{k+1} \rightarrow \ldots \xrightarrow{e_n} n} \\
\mu_2 :& y_1 \xrightarrow{ey1} y_2 \rightarrow \ldots \rightarrow x_k \rightarrow y_j \rightarrow \ldots \xrightarrow{e_n} n \\
\\
\mu_3 :& \underline{x_1 \xrightarrow{ex1} x_2 \rightarrow \ldots \rightarrow x_k} \rightarrow y_j \rightarrow \ldots \xrightarrow{e_n} n \\
\mu_4 :& y_1 \xrightarrow{ey1} y_2 \rightarrow \ldots \rightarrow \underline{x_k \rightarrow x_{k+1} \rightarrow \ldots \xrightarrow{e_n} n}
\end{aligned} \tag{5.68}$$

Claim 5. “μ_1,μ_2 are MIPS” $\implies$ “μ_3,μ_4 are MIPS”

Proof.

We use the following fact to prove the claim 5: μ_3 and μ_4 are constructed using the edges and the nodes from μ_1 and μ_2 so the following holds

$$\begin{aligned}&(\textit{modified}(\textit{cond}(e_n), \mu_1) = \phi) \ \wedge \ (\textit{modified}(\textit{cond}(e_n), \mu_2) = \phi)\\ \implies &(\textit{modified}(\textit{cond}(e_n), \mu_3) = \phi) \ \wedge \ (\textit{modified}(\textit{cond}(e_n), \mu_4) = \phi)\end{aligned} \tag{5.69}$$

$$\begin{aligned} LHS : \text{“}\mu_1,\mu_2 \text{ are MIPS”} \implies & ([\![cond(e_n)]\!]_{ex1} = [\![cond(e_n)]\!]_{ey1} = \textit{false}) \wedge \\ & (\textit{modified}(\textit{cond}(e_n), \mu_1) = \phi) \wedge \\ & (\textit{modified}(\textit{cond}(e_n), \mu_2) = \phi) && \text{from 5.67} \\ \implies & ([\![cond(e_n)]\!]_{ex1} = [\![cond(e_n)]\!]_{ey1} = \textit{false}) \wedge \\ & (\textit{modified}(\textit{cond}(e_n), \mu_3) = \phi) \wedge \\ & (\textit{modified}(\textit{cond}(e_n), \mu_4) = \phi) && \text{from 5.69} \\ \implies & \text{“}\mu_3,\mu_4 \text{ are MIPS”} && \square \end{aligned}$$

5.4.2 Proof of Optimization 2

At an edge, optimization 2 (Chapter 4.1.2) merges pairs that contain the same data flow value. In this section, we prove that this optimization retains the precision and the soundness of the FPMFP solution. A crucial difference between FPMFP and MFP is that in the FPMFP computation certain data flow values are blocked at end edges of MIPS. We prove that the same set of values are blocked at the same set of end edges with or without optimization 2.

Theorem 5.4.2. If $\overline{g}_e(\overline{\mathit{In}}_e)$ contains two pairs $\langle \mathcal{M}_1, d\rangle$ and $\langle \mathcal{M}_2, d\rangle$ with the same data flow value d then shifting d to the pair corresponding to a set of MIPS $\mathcal{M}_3$ defined below does not affect the soundness or precision of the resulting FPMFP solution. Specifically, it does not affect the blocking of d at the end edges of MIPS in $\mathcal{M}_1$ and $\mathcal{M}_2$.

$$\begin{aligned}\mathcal{M}_3 =& \{\mu_1 \mid \mu_1 \in \mathcal{M}_1,\ \exists \mu_2 \in \mathcal{M}_2.\, \mu_2 \in \textit{cso}_e(\mu_1)\}\\ & \cup \\ & \{\mu_2 \mid \mu_2 \in \mathcal{M}_2,\ \exists \mu_1 \in \mathcal{M}_1.\, \mu_1 \in \textit{cso}_e(\mu_2)\}\end{aligned} \tag{5.70}$$

Proof. We consider the following two operations each indicating optimization 2 is performed or not. Next, we prove that the FPMFP solution is identical after both the operations:

1. Operation 1: pairs $\langle \mathcal{M}_1, d\rangle$ and $\langle \mathcal{M}_2, d\rangle$ are kept as it is in $\overline{g}_e(\overline{\mathit{In}}_e)$ (meaning optimization 2 is not applied).

2. Operation 2: pairs $\langle \mathcal{M}_1, d\rangle$ and $\langle \mathcal{M}_2, d\rangle$ are replaced with $\langle \mathcal{M}_3, d\rangle$ in $\overline{g}_e(\overline{\mathit{In}}_e)$ (meaning optimization 2 is applied). For simplicity of exposition, we assume there is no value associated with $\mathcal{M}_3$ initially, otherwise we add $\langle \mathcal{M}_3, d' \sqcap d\rangle$ where d' is the value associated with $\mathcal{M}_3$ in $\overline{g}_e(\overline{\mathit{In}}_e)$.

The possible cases for MIPS in $\mathcal{M}_1$ and $\mathcal{M}_2$ w.r.t. operation 1 and 2 are shown in Table 5.1. Let the data flow value d is not killed along a MIPS μ_1 in $\mathcal{M}_1$ (if it is killed then d does not reach $\mathit{end}(\mu_1)$ after either operation is performed). We prove that both the above operations achieve similar effect in the following two cases which are mutually exclusive and exhaustive for all MIPS in $\mathcal{M}_1$ (similar argument holds for MIPS in $\mathcal{M}_2$):

1. Case 1: if $\mu_1 \in \mathcal{M}_1$ and $\mu_1 \in \mathcal{M}_3$, then d is blocked at $\mathit{end}(\mu_1)$ after both the operations.

2. Case 2: if $\mu_1 \in \mathcal{M}_1$ and $\mu_1 \notin \mathcal{M}_3$, then d reaches $\mathit{end}(\mu_1)$ after both the operations.

Proof of Case 1.

If $\mu_1 \in \mathcal{M}_1$ and $\mu_1 \in \mathcal{M}_3$, then d is blocked at $\mathit{end}(\mu_1)$ after both the operations, as explained below.

1. Operation 1 ($\langle \mathcal{M}_1, d\rangle$,$\langle \mathcal{M}_2, d\rangle$ are kept as it is in $\overline{g}_e(\overline{\mathit{In}}_e)$).

   In this case, d is blocked within $\langle \mathcal{M}_1, d\rangle$ at $\mathit{end}(\mu_1)$ because $\mu_1 \in \mathcal{M}_1$. Also, d is blocked within $\langle \mathcal{M}_2, d\rangle$ because there exists a MIPS μ_2 in $\mathcal{M}_2$ whose suffix is contained in μ_1 i.e.,

$$\mu_1 \in \mathcal{M}_1 \wedge \mu_1 \in \mathcal{M}_3 \implies \exists \mu_2 \in \mathcal{M}_2 . \mu_2 \in \mathit{cso}_e(\mu_1) \tag{5.71}$$

$$\mu_2 \in \mathcal{M}_2 \wedge \mu_2 \in \mathit{cso}_e(\mu_1) \implies \mathit{end}(\mu_2) \in \mathit{edges}(\mu_1) \text{ and} \tag{5.72}$$

$$d \text{ is blocked within } \langle \mathcal{M}_2, d\rangle \text{ at } \mathit{end}(\mu_2) \tag{5.73}$$

   Hence, in this scenario, d is blocked within both pairs before reaching $\mathit{end}(\mu_1)$ or at $\mathit{end}(\mu_1)$.

		is d blocked at the edge in *end*(μ) after	
		operation 1	operation 2
Cases	1. $\mu \in \mathcal{M}_1, \mu \in \mathcal{M}_3$	Yes	Yes
	2. $\mu \in \mathcal{M}_1, \mu \notin \mathcal{M}_3$	No	No
	3. $\mu \in \mathcal{M}_2, \mu \in \mathcal{M}_3$	Yes	Yes
	4. $\mu \in \mathcal{M}_2, \mu \notin \mathcal{M}_3$	No	No

Table 5.1: Equivalence of operation 1 and operation 2 w.r.t. cases in optimization 2. Optimization 2 involves shifting a data flow value d from pairs $\langle \mathcal{M}_1, d\rangle$ and $\langle \mathcal{M}_2, d\rangle$ to pair $\langle \mathcal{M}_3, d\rangle$. For simplicity, we assume d is not killed along μ.

2. Operation 2 ($\langle \mathcal{M}_1, \top\rangle$,$\langle \mathcal{M}_2, \top\rangle$,$\langle \mathcal{M}_3, d\rangle$ belong to $\overline{g}_e(\overline{\mathit{In}}_e)$ and $\mu_1 \in \mathcal{M}_3$).

   In this case, d is blocked within $\langle \mathcal{M}_3, d\rangle$ at *end*(μ_1) because $\mu_1 \in \mathcal{M}_3$.

Proof of Case 2.

If $\mu_1 \in \mathcal{M}_1$ and $\mu_1 \notin \mathcal{M}_3$, then d reaches *end*(μ_1) after both the operations.

1. Operation 1 ($\langle \mathcal{M}_1, d\rangle$,$\langle \mathcal{M}_2, d\rangle$ are kept in $\overline{g}_e(\overline{\mathit{In}}_e)$ as it is).

   In this case, d is blocked within $\langle \mathcal{M}_1, d\rangle$ at *end*(μ_1). However, d in $\langle \mathcal{M}_2, d\rangle$ is not blocked at or before *end*(μ_1) because μ_1 does not contain *suffix*(μ_2, e) for any MIPS μ_2 in $\mathcal{M}_2$ (meaning μ_2 either 1) does not follow the same CFP as μ_1 after e, or 2) follows the same CFP as μ_1 but does not end before μ_1 on the CFP). Thus, d in $\langle \mathcal{M}_2, d\rangle$ reaches *end*(μ_1) along a CFP that goes through *suffix*(μ_1, e).

2. Operation 2 ($\langle \mathcal{M}_1, \top\rangle$,$\langle \mathcal{M}_2, \top\rangle$,$\langle \mathcal{M}_3, d\rangle$ belong to $\overline{g}_e(\overline{\mathit{In}}_e)$).

   In this case, μ_1 does not contain *suffix*(μ_3, e) for any MIPS μ_3 in $\mathcal{M}_3$ i.e.,

   $\mu_1 \in \mathcal{M}_1 \land \mu_1 \notin \mathcal{M}_3 \implies \forall \mu_3 \in \mathcal{M}_3 . \mu_3 \notin \mathit{cso}_e(\mu_1)$

   This can be proved by proving that following is a contradiction.

$$\begin{aligned}
&\mu_1 \in \mathcal{M}_1 \land \mu_1 \notin \mathcal{M}_3 \land \exists \mu_3 \in \mathcal{M}_3 . \mu_3 \in \mathit{cso}_e(\mu_1) && \\
&\implies \mu_3 \notin \mathcal{M}_2 && \text{from 5.70} \\
&\implies \mu_3 \in \mathcal{M}_1 \land \exists \mu_2 \in \mathcal{M}_2 . \mu_2 \in \mathit{cso}_e(\mu_3) && \text{from 5.70} \\
&\implies \exists \mu_2 \in \mathcal{M}_2 . \mu_2 \in \mathit{cso}_e(\mu_1) && \mathit{cso}_e(\mu_3) \subseteq \mathit{cso}_e(\mu_1) \\
&\implies \mu_1 \in \mathcal{M}_3 && \text{from 5.70}
\end{aligned}$$

$\implies$ *false*

Hence d in $\langle \mathcal{M}_3, d \rangle$ is not blocked at or before *end*(μ_1). Thus, d in $\langle \mathcal{M}_3, d \rangle$ reaches *end*(μ_1) along a CFP that goes through *suffix*(μ_1, e).

Similar cases can be proved for MIPS in $\mathcal{M}_2$. Thus, the FPMFP solutions with or without optimization 2 is equivalent. □

5.5 Chapter Summary

In this chapter, we defined the MOFP solutions of data flow analyses and proved that FPMFP solutions are a sound approximation of MOFP solutions. Further, we proved the optimizations introduced for improving scalability of FPMFP solutions do not affect the soundness or precision of the solutions.

Chapter 6

Experiments and Results

In this chapter, we empirically compare FPMFP solution with MFP solution with respect to the precision of the solutions and the efficiency of their computation. Indeed, we did not find any other existing technique that is as generic as the MFP computation, and eliminates the effect of infeasible paths from exhaustive data flow analysis (existing techniques are explained in Chapter 7).

The chapter is organized as follows. First, we explain the experimental setup and benchmark characteristics (Section 6.1). Next, we describe two client analyses on which we compared precision of FPMFP and MFP solutions (Section 6.2), followed by comparing the performance of FPMFP and MFP solutions (Section 6.3). Lastly, we discuss infeasible path patterns observed in our benchmarks (Section 6.4)

6.1 Experimental Setup

We performed our experiments on a 64 bit machine with 16GB RAM running Windows 7 with Intel core i7-5600U processor having a clock speed of 2.6 GHz. We used 30 C programs of up to 150KLoC size which contained 7 SPEC CPU-2006 benchmarks [1], 5 industry codesets, and 18 open source codesets [33, 40, 56]. These programs contain at least one infeasible path and are chosen to cover a variety of application areas with different program characteristics. No

[1] The spec benchmarks like specrand which did not have any infeasible paths are not included. For other benchmarks like gcc, gromacs, perlbench we were unable to get them compiled with our framework because of framework issues.

additional criteria is used for choosing the benchmarks. The number of distinct MIPS in these benchmarks range from 17 for the *acpid* benchmark to 2.1k for the *h264ref* benchmark. In these benchmarks, up to 61% (geomertic mean 29%) functions had at least one MIPS in their CFG. On average 77% MIPS were overlapping with at least one other MIPS.

We have used the algorithms proposed by Bodik et al. [8] in the pre-processing stage of FPMFP computation to generate a set of MIPS. We present these algorithms in Appendix. The corresponding performance overhead for detecting MIPS is described in Section 6.3.

We used the iterative fix-point based data flow analysis algorithm [2, 31] implemented in Java in a commercial static analysis tool *TCS Embedded Code Analyzer* [1] for our data flow analysis. TCS ECA addresses 25 different type of C program issues using static analysis like potentially uninitialized variables, change impact analysis, array index out of bounds, zero division, deadlock detection etc.

Initially, the tool transforms the source code to an intermediate representation, and constructs CFGs for procedures in the program. Later, it runs some basic inbuilt analyses like pointer analysis, alias analysis in flow sensitive and context insensitive manner, followed by constructing the program call graph. Since FPMFP just lifts the abstract operations given in the input MFP specification, it remains oblivious to and respects the analysis specific choices like use of aliasing, pointers, and field sensitivity. In our evaluation, we implemented the functional approach of inter-procedural analysis for computing MFP and FPMFP solutions for potentially uninitialized variable analysis and reaching definitions analysis. The details of the same are discussed in Section 6.2.

6.2 Comparing the Precision of FPMFP and MFP Solutions

6.2.1 Reduction in the Number of Potentially Uninitialized Variable Alarms

A potentially uninitialized variable analysis reports the variables that are used at a program point but are not initialized along at least one path that reaches the program point. We computed the potentially uninitialized variables as follows: first, we implemented the *must defined variables* analysis that computes the set of non-array variables that are defined along all CFPs reaching a program point. Next, we take the complement of the *must defined* data flow results to report the

Table 6.1: Benchmark properties: three types of benchmarks that we used include Open source [33, 40, 56], Industry, and SPEC CPU 2006 benchmarks.

Benchmark Properties						
Type	Name	KLOC	#MIPS	#Overlaping MIPS	#Balanced MIPS	%Funcs Impacted
Open Source	1.acpid	0.3	6	6	6	7
	2.polymorph 0.96	0.4	24	24	17	11
	3.nlkain	0.5	33	8	3	58
	4.spell	0.6	14	11	12	23
	5.ncompress	1	30	27	16	30
	6.gzip	1.3	49	46	40	25
	7.stripcc	2.1	107	97	74	30
	8.barcode-nc	2.3	60	49	22	25
	9.barcode	2.8	76	66	28	24
	10.archimedes	5	118	86	53	38
	11.combine	5.7	234	210	124	29
	12.httpd	5.7	226	211	88	16
	13.sphinxbase	6.6	213	199	109	16
	14.chess	7.5	262	186	112	39
	15.antiword	19.3	669	570	306	32
	16.sendmail	20.3	726	681	349	32
	17.sudo	45.8	250	223	148	15
	18.ffmpeg	80	1363	1201	891	9
SPEC	19.mcf	5.4	19	17	4	20
	20.bzip2	16	501	485	184	40
	21.hmmer	16.5	595	522	270	20
	22.sjeng	27	438	379	198	33
	23.milc	30	497	375	141	38
	24.h264ref	35.9	2189	1696	526	31
	445.gobmk	150	1834	1435	756	13
Industry	26.NZM	2.6	52	43	34	43
	27.IC	3.7	50	29	13	15
	28.AIS	6	83	56	21	14
	29.FRSC	6	42	21	1	15
	30.FX	7.5	140	78	39	34

Table 6.2: Reduction in the number of potentially uninitialized variable Alarms

Benchmarks		Uninitialized Variable Alarms		
		MFP	FPMFP	reduction(%)
Open Source	1.acpid	1	1	0(0)
	2.polymorph	4	4	0(0)
	3.nlkain	4	0	4(100)
	4.spell	3	3	0(0)
	5.ncompress	7	7	0(0)
	6.gzip	0	0	-
	7.stripcc	27	1	26(96.30)
	8.barcode-nc	0	0	-
	9.barcode	2	0	2(100)
	10.archimedes	61	56	5(8.20)
	11.combine	63	63	0(0)
	12.httpd	117	117	0(0)
	13.sphinxbase	46	43	3(6.52)
	14.chess	16	16	0(0)
	15.antiword	18	18	0(0)
	16.sendmail	103	102	1(0.97)
	17.sudo	62	58	4(6.45)
	18.ffmpeg	124	112	12(9.68)
SPEC	19.mcf	5	5	0(0)
	20.bzip2	72	35	37(51.39)
	21.hmmer	636	629	7(1.10)
	22.sjeng	19	19	0(0)
	23.milc	188	168	20(10.64)
	24.h264ref	52	48	4(7.69)
	25.gobmk	1216	1199	17(1.39)
Industry	26.NZ	3	0	3(100)
	27.IC	5	5	0(0)
	28.AIS	0	0	-
	29.FRSC	3	3	0(0)
	30.FX	84	84	0(0)

Table 6.3: Reduction in the number of def-use pairs: 1) MFP (resp. FPMFP) column shows the sum of number of def-use pairs at all program nodes n in the CFG of a program for global and local variables.

Benchmarks		Reaching Definition Analysis		
Type	Name	#def-use pairs		reduction(%)
		MFP	FPMFP	
Open Source	1.acpid	156	156	0(0.00)
	2.polymorph	228	228	0(0.00)
	3.nlkain	1042	965	77(7.39)
	4.spell	516	515	1(0.19)
	5.ncompress	1201	1175	26(2.16)
	6.gzip	3423	3401	22(0.64)
	7.stripcc	2703	2645	58(2.15)
	8.barcode-nc	3051	3007	44(1.44)
	9.barcode	3709	3653	56(1.51)
	10.archimedes	44337	44216	121(0.27)
	11.combine	16618	15859	759(4.57)
	12.httpd	10475	10072	403(3.85)
	13.sphinxbase	9641	8482	1159(12.02)
	14.chess	31386	31303	83(0.26)
	15.antiword	68889	60144	8745(12.69)
	16.sendmail	102812	101470	1342(1.31)
	17.sudo	14391	14211	180(1.25)
	18.ffmpeg	89148	86607	2541(2.85)
SPEC	19.mcf	3570	3565	5(0.14)
	20.bzip2	82548	77967	4581(5.55)
	21.hmmer	70597	70021	576(0.82)
	22.sjeng	77602	76959	643(0.83)
	23.milc	18133	17828	305(1.68)
	24.h264ref	426346	409325	17021(3.99)
	25.gobmk	213243	184142	29101(13.65)
Industry	26.NZ	2652	2651	1(0.04)
	27.IC	3111	3063	48(1.54)
	28.AIS	2475	2414	61(2.46)
	29.FRSC	1297	1295	2(0.15)
	30.FX	4971	4925	46(0.93)

Table 6.4: Comparing performance of MFP and FPMFP computation. *Prep.* represents the MIPS detection phase. Note the analysis times are not comparable with gcc or clang, because our implementation is in Java.

Benchmarks		Analysis Time (Sec)				Memory Consumption (MB)			
		Prep	MFP	FPMFP	Increase(x)	Prep	MFP	FPMFP	Increase(x)
Open Source	1.acpid	1.1	0.8	0.7	-0.1	0	0	0	0
	2.polymorph 0.96	1.1	0.8	0.7	-0.1	1	0	0	0
	3.nlkain	1.2	0.9	1.2	0.3	1	1	1	0
	4.spell	1.1	0.8	1	0.2	1	1	0	0
	5.ncompress	1	0	1	1(1)	2	1	4	2(2)
	6.gzip	1	1	5	4(4)	5	10	24	14(1.4)
	7.stripcc	3	1	6	5(5)	8	5	21	15(3)
	8.barcode-nc	2	1	2	1(1)	6	8	6	-1(-0.1)
	9.barcode	2	2	2	0	7	10	8	-1(-0.1)
	10.archimedes	6	3	32	29(10)	24	44	156	112(2.5)
	11.combine	5	3	5	2(0.6)	16	23	19	-3(-0.1)
	12.httpd	40	14	19	5(0.3)	28	39	21	-18(-0.46)
	13.sphinxbase	7	3	3	0	13	16	7	-9(-0.56)
	14.chess	13	7	30	23(3)	31	93	191	98(1)
	15.antiword	13	16	82	66(4)	47	101	218	117(1)
	16.sendmail	110	142	2060	1918(13)	81	340	1548	1208(3.5)
	17.sudo	15	9	17	8(1)	35	71	19	-52(-0.7)
	18.ffmpeg	234	51	80	29 (0.5)	158	266	90	-175(-0.6)
SPEC 2006	19.mcf	1	1	1	0	0	3	3	0
	20.bzip2	8	7	69	62 (9)	28	44	90	46(1)
	21.hmmer	14	9	23	14(1.5)	50	67	58	-8(-0.1)
	22.sjeng	10	16	62	46(2.8)	27	122	215	92(0.75)
	23.milc	6	4	7	3(0.7)	25	25	14	-11(-0.4)
	24.h264ref	60	115	451	336(2.9)	131	1003	1644	640(0.6)
	25.gobmk	4218	1696	9818	8122(4.7)	390	9365	17392	8032(0.85)
Industry	26.NZM	1	1	2	1(1)	3	8	7	-1(-0.1)
	27.IC	5	4	3	-1(-0.25)	11	20	9	-11(-0.5)
	28.FRSC	3	2	1	-1(-0.5)	-15	10	4	-6(-0.6)
	29.AIS	4	3	4	1(0.3)	9	20	16	-4(-0.2)
	30.FX	7	3	7	4(1.3)	20	35	47	11(0.3)

Table 6.5: Impact of Optimization 1 on the FPMFP Computation (in optimization 1, we consider two MIPS with the same end edge as equivalent, and subsequently merge the pairs that contain equivalent MIPS). M,B,T represent million, billion, and trillion respectively.

Benchmarks		Number of Distinct MIPS Before and After Optimization 1		
		Before	After	Reduction(%)
Open Source	1.acpid	83	6	77(92.77)
	2.polymorph 0.96	65	24	41(63.08)
	3.nlkain	35	33	2(5.71)
	4.spell	99	14	85(85.86)
	5.ncompress	14163	30	14133(99.79)
	6.gzip	23.5M	49	23.5M(99.99)
	7.stripcc	0.7M	107	0.7M(99.99)
	8.barcode-nc	208	60	148(71.15)
	9.barcode	3851	76	3775(98.03)
	10.archimedes	1M	118	1M(99.99)
	11.combine	27061	234	26827(99.14)
	12.httpd	76.5B	226	76.5B(99.99)
	13.sphinxbase	6463	213	6250(96.70)
	14.chess	19.7B	262	19.7B(99.99)
	15.antiword	1446.7T	669	1446.7T(99.99)
	16.sendmail	94T	726	94T(99.99)
	17.sudo	549565	250	549315(99.95)
	18.ffmpeg	5T	1363	5T(99.99)
SPEC 2006	19.mcf	44	19	25(56.82)
	20.bzip2	115M	501	115M(99.99)
	21.hmmer	8.6M	595	8.6M(99.99)
	22.sjeng	1508T	438	1508T(99.99)
	23.milc	2322	497	1825(78.60)
	24.h264ref	51B	2189	51B(99.99)
	25.gobmk	$> 2^{64}$	1834	$> 2^{64}$(99.99)
Industry	26.NZM	1064	52	1012(95.11)
	27.IC	158	50	108(68.35)
	28.AIS	176	83	93(52.84)
	29.FRSC	89	42	47(52.81)
	30.FX	704	140	564(80.11)

Table 6.6: Impact of Optimization 2 on the FPMFP Computation. Timeout indicates the analysis did not finish within 10K seconds. For smaller codesets (1 to 4), applying Optimization 2 has not produced any reduction in analysis time, instead the total analysis time has increased because of cost of applying Optimization 2.

Benchmarks		Analysis Time (in Seconds) Before and After Optimization 2		
		Before	After	Reduction(%)
Open Source	1.acpid	0.6	0.7	-0.1(-16)
	2.polymorph 0.96	0.5	0.7	-0.2(-40)
	3.nlkain	0.6	1.2	-0.6(-100)
	4.spell	0.7	1	-0.3(-42)
	5.ncompress	3	1	2(66)
	6.gzip	22	5	17(77)
	7.stripcc	55	6	49(89)
	8.barcode-nc	2	2	0
	9.barcode	2	2	0
	10.archimedes	53	32	21(39)
	11.combine	19	5	14(73)
	12.httpd	290	19	271 (93)
	13.sphinxbase	4	3	1(25)
	14.chess	76	30	46(60)
	15.antiword	705	82	623(88)
	16.sendmail	Timeout	2060	-
	17.sudo	64	17	47(73)
	18.ffmpeg	Timeout	80	-
SPEC 2006	19.mcf	1	1	0
	20.bzip2	931	69	862 (92)
	21.hmmer	143	23	120(83)
	22.sjeng	501	62	439(87)
	23.milc	11	7	4(36)
	24.h264ref	2977	451	2526(84)
	25.gobmk	Timeout	9818	-
Industry	26.NZM	4	2	2(50)
	27.IC	6	3	3(50))
	28.AIS	7	4	3(42)
	29.FRSC	3	1	2(66)
	30.FX	9	7	2(22)

variables that are used at a program point p but are not initialized along at least one CFP that reaches p. We call such CFPs as *witness CFPs*.

As shown in Table 6.2, we computed two sets of potentially uninitialized variable alarms corresponding to the MFP and FPMFP solutions of must defined variable analysis. In particular, the potentially uninitialized variable analysis that used the FPMFP solution of *must defined* analysis reported up to 100% (average 18.5%, geo. mean 3%) fewer alarms compared to when it used the MFP solution. In the reduced alarm cases, all the witness CFPs were found infeasible.

More specifically, in 14 out of 27 benchmarks the FPMFP solution helped achieve positive reduction in the uninitialized variable alarms, while in 13 benchmarks there was no reduction in the alarms. In the latter case the following two scenarios were observed: in the first scenario, the addresses of uninitialized variables are passed as parameter to library functions whose source code is not available and hence its behavior is conservatively approximated (i.e, the analysis assumed that the variables are not initialized inside the functions), and in the second scenario a context sensitive analysis is required to eliminate the false positive alarms (our implementation is context insensitive).

6.2.2 Reduction in the Number of Def-Use Pairs

We implemented the classic reaching definitions analysis [2, 31] (non SSA based) in which for each program point, we compute the set of definitions that reach the point. We measured the effect of the reduction in the number of reaching definitions on Def-Use pair computation, which is a client analysis of the reaching definitions analysis. In this, we pair the definition of a variable with its reachable use point.

In the FPMFP computation, the def and use points that are connected with each other by only infeasible paths are eliminated. However, we found that most def-use points are connected by at least one feasible CFP, and hence a 100% reduction is not achievable by any technique.

As shown in Table 6.3, we found that up to 13.6% (average 2.87%, geometric mean 1.75%) fewer def-use pairs were reported when using FPMFP solution in place of the MFP solution. In particular, in 28 out of 30 benchmarks the FPMFP solution lead to fewer def-use pairs than the MFP solution. In general, we observed that the precision increased with increase in the number of MIPS. For example, the *gobmk* benchmark contains a large number of MIPS (1.9k), consequently we observed significant (13%) reduction in the number of def-use pairs. On the other hand, the *acpid* benchmark had only 17 MIPS, consequently we did not see any reduction in the number of def-use pairs. As an outlier case, on the *sphinxbase* benchmark that had only 216 MIPS we observed 12% reduction in def-use pairs. Here, we observed that the end edges of many MIPS are reachable only along infeasible CFPs, hence in FPMFP computation the data flow along these edges is completely blocked, thus all the corresponding def-use pairs (at the program points pre-dominated by the end edges) are eliminated.

The def-use pairs have multifold applications; in particular, they are useful for program slicing, program dependency graph creation, program debugging, and test case generation. For example in program testing, some coverage criteria mandate that each def-use pair should be covered by at least one test case [52, 54]. Here, up to 13.6% fewer def-use pairs means fewer manual test cases need to be written to cover these pairs, which is a significant reduction in the manual time and efforts.

6.3 Comparing the Performance of FPMFP and MFP Computations

Computation of the FPMFP solution was less efficient than the corresponding MFP solutions. The details of the relative memory and time consumption of both the analyses for reaching definitions computation are presented in Table 6.4. The *Prep.* column represents the time and memory consumption for infeasible path detection phase. This is performed once for each program, and hence is reported separately.

The FPMFP analysis took 2.9 times more time than MFP analysis on average (geo. mean 1.6). In general, the analysis time increased with increase in the number of MIPS in the program. Apart from this, the FPMFP analysis consumed 0.5 times more memory than the MFP analysis on average. However, on 50% of the bechmarks the FPMFP analysis consumed less memory than the MFP analysis because the definitions reaching along infeasible CFPs are ignored during the analysis.

The effect of optimizations for scalability on analysis time

Table 6.5 shows the reduction in the number of distinct MIPS after applying Optimization

1 (Section 4.1.1). Here, two MIPS are considered equivalent if they have the same end edge, in this case we can merge two pairs if they contain equivalent MIPS. We found that if we count MIPS with same end edge single time, then the total number of MIPS reduces by up to 99%(average 85%, geometric mean 78%). This leads to reduction in number of pairs at various program points.

Table 6.6 shows the effectiveness of optimization 2 (Section 4.1.2) on top of optimization 1. In particular, it shows the analysis time before and after applying the optimization 2. For smaller benchmarks (1 to 4), applying Optimization 2 has not resulted in any reduction in analysis time, instead the total analysis time has increased because of overhead of applying Optimization 2. On larger benchmarks, however, we saw reduction in analysis time after applying Optimization 2. In particular, we found that 3 out of 30 applications timed out (i.e., did not finish within 10K seconds) when we disabled optimization 2. Similarly, for remaining applications the analysis time increased by up to 99%(average 43%, geometric mean 19.8%) after disabling optimization 2.

In Chapter 7, we distinguish FPMFP from other techniques of improving data flow precision. In particular, we did not find any other generic technique that eliminates effect of infeasible paths from data flow analysis without using CFG restructuring. The techniques that use CFG restructuring [50, 3, 4, 7, 5, 36] can blow up the size of CFG exponentially. Moreover they have additional overhead of replicating CFG nodes which is not present in our approach. Lastly, these techniques do not take advantage of analysis specific information (like Optimization 2) to discard the information separation where it is not useful.

6.4 Types of MIPS Observed

In general case, knowing whether a path is infeasible or not is undecidable. Nevertheless, a large number of infeasible paths in a program can be identified using statically detectable correlations between two branch conditions or between a branch condition and an assignment statement. Indeed, in our work, we detected a large number of MIPS using Bodik's approach (their approach is described in Appendix A).

We observe that a lot of MIPS in our benchmarks belong to one of the following two types resulting from correlation between an assignment statement and a branch statement or between

```
If(...){
 A: x = 5;
}
...
If(x < 5){
 B: print("%d", &x)
}
```

(a)

```
Const int N = 100
A: x = 0;
While(x < N){
...
x++;
}
B:
```

(b)

```
If(x < 5){
A:
}
...
If(x < 5){
...
}else{
 B:
}
```

(c)

Figure 6.1: Illustrating the types of MIPS observed. In each of the example above there is a MIPS going from point A to point B. The variable x is not modified in the code portion represented by three dots (. . .) .

two branch statements.

1. The first type involves assignment of constant value to a variable along some node in a path followed by the variable being used in some conditional expression that performs either *equality* or *less than* or *greater than* check on the variable later in the path. Two typical examples of this type are shown Figure 6.1a and 6.1b. In the first example a variable is assigned a constant value inside the if branch (point A) and subsequently used in other branch condition that appears later in the path (point B). The second example contains a loop that executes at least once hence the path that goes from point A to point B without executing the loop is infeasible.

2. The second type involves two syntactically same conditions used in two branches at different places in a path and the variables in the conditions are not modified along the path that connects the branches. For example in Figure 6.1c the path that goes from point A to point B is infeasible.

6.5 Summary of Empirical Observations

Previous empirical evidence [8] on linux kernel code shows 9-40% of conditional statements contribute to at least one infeasible CFP. In our benchmarks up to 61% (geometric mean 29%)

functions had at least one MIPS in their CFG. Precise elimination of data flow values reaching from infeasible CFPs has allowed us to reduce the number of def-use pairs by up to 13.6% with an average of 2.87% and a geometric mean of 1.75% over MFP solution. Similarly, the reduction in the potentially uninitialized variable alarms was up to 100% (average 18.5%, geo. mean 3%).

Chapter 7

Related Work

In this chapter, we compare FPMFP with existing approaches of improving data flow precision. In particular, we begin by classifying the existing approaches that improve the precision of data flow analysis into two categories. The approaches in the first category avoid the effect of infeasible paths from data flow analysis (Section 7.1), while the approaches in second category avoid the join points (of two or more CFPs) that lead to imprecision in the analysis results (Section 7.2). Lastly, we explain how FPMFP differs from approaches in both the categories.

7.1 Approaches that Avoid Infeasible Paths

In this section, we discuss approaches that improve precision of data flow analysis by avoiding infeasible paths.

In general, presence of infeasible paths in programs is well known [58, 22, 35, 32, 28, 17, 46, 55]. Hedley et al. [22] presented a detailed analysis of the causes and effects of infeasible paths in programs. Malevris et al. [35] observed that the greater the number of conditional statements contained in a path, the greater the probability of the path being infeasible. Bodik et al. [8] found that around 9-40% of the conditional statements in programs show statically detectable correlation with infeasible control flow paths.

Many approaches have been proposed to eliminate the effect of infeasible paths from data flow analysis. We classify these approaches in two categories as shown in Figure 7.1. The first category includes approaches that are analysis dependent (Section 7.1.1), while the second category includes the approaches that are analysis independent (Section 7.1.2). We describe and

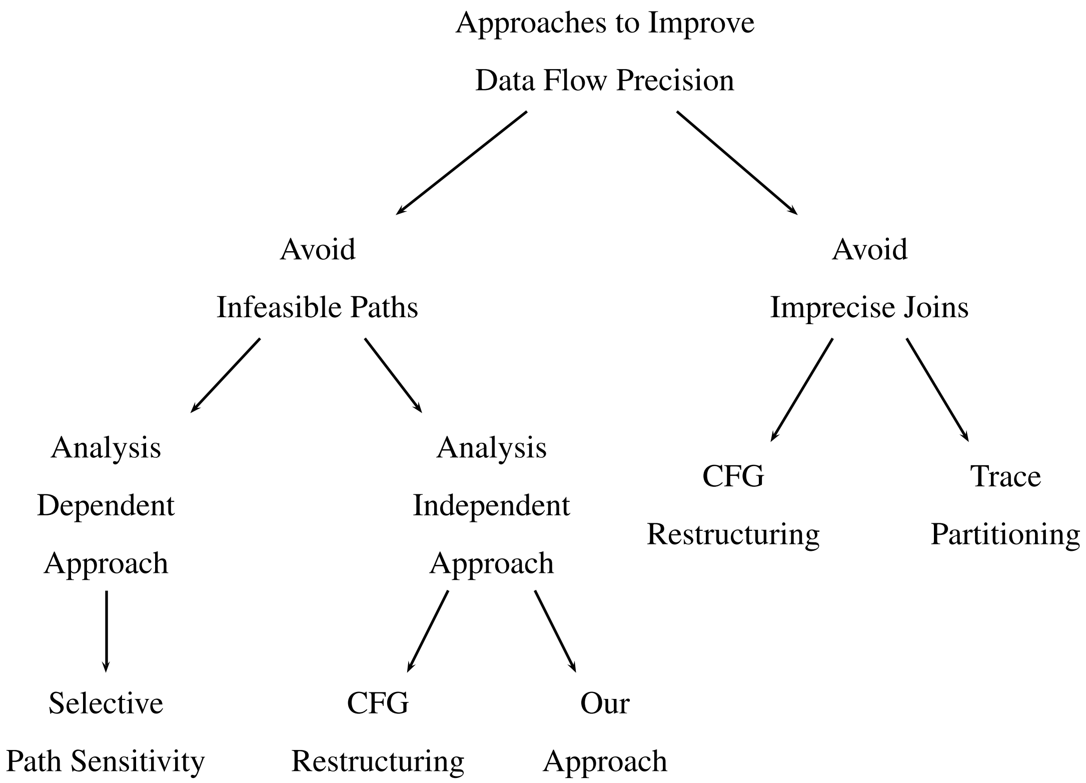


Figure 7.1: Categorization of approaches that improve data flow precision by either avoiding infeasible paths or imprecise joins.

relate approaches in each of these categories with FPMFP below.

7.1.1 Analysis Dependent Approaches

In this section, we describe approaches that eliminate effect of infeasible paths from data flow analysis in analysis dependent way. In particular, these approaches either depend on analysis specific heuristics, or they detect infeasible paths on the fly during a data flow analysis (and hence they detect infeasible paths separately for each different data flow analysis over the same program thereby increasing the analysis cost). Specifically, these approaches use the idea of path sensitive analysis to distinguish between the data flow values reaching along different classes of CFPs [11, 12, 18, 53, 13, 14].

In a completely path sensitive analysis each path information is separately tracked along with the corresponding path constraints. If the path constraint is found unsatisfiable, the path information is not propagated further. This approach does not scale to most practical programs due to presence of loops and recursion leading to an unbounded number of program paths combined with data flow lattices containing possibly infinite values.

The above limitations of a completely path sensitive analysis are avoided by adapting a selective approach to path sensitive analysis [14, 13, 53, 18, 12, 11, 50, 10]. Here, they use selective path constraints governed by various heuristics for deciding which path information should be kept separately and which path information can be approximated at join points. These heuristics are often aligned with the end goal of the analysis. Our work differs from these approaches in that our work does not depend on analysis specific heuristics for eliminating the effect of infeasible paths, and therefore can be generically applied to all MFP based data flow analyses. Below, we describe the approaches that use selective path sensitive analyses.

Dor et al [14] proposed a precise path sensitive value flow analysis approach using value bit vectors. Here, they symbolically evaluate a program by generating symbolic states that include both the execution state and the value alias set. At merge points in the control flow graph, if two symbolic states have the same value alias set, they produce a single symbolic state by merging their execution states. Otherwise, they process the symbolic states separately. Like our work, they avoid the cost of full path sensitive analysis, yet capture relevant correlations. Their work differs from our work in that their work is applicable only to separable data flow problems , while our work is applicable to any data flow problem that can be solved using MFP computation.

Chen et al. [45] proposed an algorithm for detecting infeasible paths by incorporating the branch correlations analysis into the data flow analysis technique. Next, the detected infeasible paths are then used to improve the precision of structural testing by ignoring infeasible paths.

Hampapuram et al. [18] performed a symbolic path simulation approach in which they executed each path symbolically and propagated the information along these paths. If a path is found infeasible in simulation its information is not propagated. A lightweight decision procedure is used to check the feasibility of a path during path simulation.

Dillig et al. [13] proposed a selective path sensitive analysis for modular programs. In this, they ignored the unobservable variables that are the variables representing the results of unknown functions (functions unavailable for analysis); hidden system states (e.g., the garbage collectors free list, operating systems process queue, etc.), or imprecision in the analysis (e.g., selecting an unknown element of an array). In particular, they observed that unobservable variables are useful within their natural scope, for example, tests on unobservable variables can possibly be proven mutually exclusive (e.g., testing whether the result of a malloc call is null, and then testing whether it is non-null). However, outside of their natural scope unobservable variables provide no additional information and can be eliminated. Ignoring unobservable variables allowed them to reduce the number of distinctions that needs to be maintained in the path sensitive analysis, for example, they did not distinguish between paths that differ only in the values of unobservable variables. They have applied their work for proving a fixed set of program properties. In that, they eliminate the effect of infeasible paths while proving the properties because the analysis is path sensitive. However, they detect infeasible paths separately for each different data flow analysis on the same underlying program, which does not happen in our approach.

Xie et al. [53] focused on memory access errors (like *array index out of bounds* and *null pointer dereferences*) because they may lead to non-deterministic and elusive crashes. They proposed a demand-driven path sensitive inter-procedural analysis to bound the values of both variables and memory sizes. For achieving efficiency, they restrict the computation to only those variables and memory sizes that directly or indirectly affect the memory accesses of interest. They use the following observation about $\mathcal{C}$ programs: most of the times memory sizes are passed as parameters to functions, along with the pointers to corresponding memory addresses; further, these sizes are used in conditions that guard the memory accesses. Hence, by tracking the actual values of the memory sizes the corresponding memory accesses can be evaluated as safe or unsafe. Xie et al. were able to detect many memory access errors in real code sets using this method. Moreover, they could eliminate the effect of some of the infeasible paths that reach the memory accesses of interest because they used a path sensitive analysis. Their work differs from our work on two fronts: first, they do a demand driven analysis while our work is exhaustive; second in their work the infeasible paths are detected on the fly during a data flow analysis, so the same path may be detected multiple times for the same underlying program, while in our work infeasible paths are detected only once for a program.

Das et al. [11] proposed a property specific path sensitive analysis. In this at first, a set of program properties of interest are identified. Next, a partial path sensitive analysis is performed where two paths are separated iff they contain different values for the variables related to properties of interest.

On somewhat similar lines, Dhurjati et al. [11] proposed an iterative refinement based approach to path sensitive data flow analysis. Here, at first, a path insensitive analysis is performed. Next, if the analysis is not able to prove the properties of interest then the analysis results are used to compute a set of predicates that are related to the property of interest . In the next iteration of the analysis, these predicates are used to differentiate paths along which predicate evaluation is different. If this partially path sensitive analysis proves the properties of interest then the analysis is terminated, else the analysis is repeated with a different set of predicates discovered in the recent iteration of the analysis. Thus, the analysis continues in multiple iterations.

Infeasible control flow paths is a property of programs instead of any particular data flow analysis. We use this observation to first detect infeasible paths in programs and propose an approach to improve the precision of any data flow analysis over the programs, unlike above approaches that remove the impact of infeasible paths from a particular data flow analysis.

7.1.2 Analysis Independent Approaches

We now describe approaches that eliminate effect of infeasible paths from data flow analysis in analysis independent way. In particular, these approaches do not depend on analysis specific heuristics, and they identify infeasible paths only once for a program. These approaches are classified into two categories depending on whether they use CFG restructuring or not, as explained below.

Approaches that Use CFG Restructuring

In this section, we discuss approaches that use CFG restructuring for improving data flow precision by eliminating infeasible paths from CFG.

Bodik et al. [7] proposed an inter-procedural version of infeasible path detection and its application to improve the data flow precision. However, they use control flow graph restructuring which is not done in our approach. CFG restructuring may blow up the size of the CFG. Moreover, it does not take the advantage of analysis specific information, unlike FPMFP that uses analysis specific information to dynamically discard the distinctions between MIPS, where they do not lead to precision improvement as described in Optimization 2 (Section 4.1.2).

Balakrishna et al. [5] presented a technique for detecting infeasible paths in programs using abstract interpretation. The technique uses a sequence of path-insensitive forward and backward runs of an abstract interpreter to infer paths in the control flow graph that cannot be exercised in concrete executions of the program. Next, they refined program CFG by removing detected infeasible paths in successive iterations of a property verification.

Marashdih et al. [36] proposed a CFG restructuring approach to eliminate infeasible paths for PHP programs. In particular, they proposed a methodology for the detection of XSS (Cross Site Scripting) from the PHP web application using static analysis. The methodology eliminates the infeasible paths from the CFG thereby reducing the false positive rate in the outcomes. On similar lines, we have found reduction in false positives in possibly uninitialized variable alarms but without using CFG restructuring.

Our Approach

We used the work done by Bodik et al. [8] for detecting infeasible paths from the CFG of a program. Next, we automatically lift any data flow analysis to an analysis that separates the data flow values reaching along known infeasible CFPs from the values that do not reach along the infeasible CFPs. This allows us to block the data flow values that reach along infeasible CFPs at a program point, thereby eliminating the effect of detected infeasible paths from data flow analyses without using CFG restructuring and in a generic manner.

7.2 Approaches that Avoid Imprecise Joins

In this section, we discuss approaches that improve data flow precision by avoiding imprecise joins i.e., the merging of information at the nodes that are shared by two or more paths, and that may lead to loss of precision.

7.2.1 Approaches that Use Trace Partitioning

In abstract interpretation, the trace partitioning approaches [19, 24, 37] partition program traces where values of some variable or conditional expression differ in two selected traces. The variables or expressions are selected based on the alarms to be analyzed or some heuristic.

Mauborgne et al. [37] used trace partitioning to delay the merging of information at join points in CFG where the join points lead to decrease in precision. They separate sets of traces before the join points and merge traces after the join points. The separation and merge points for traces are specified as input to the program in the form of pre-processing directives. Handjieva et al. [19] proposed a restricted version of trace partitioning where merging of parts in partitions was not allowed (Unlike Mauborgne et al.'s work).

The trace partitioning is similar to our approach because it keeps data flow values separately for each part (representing a set of traces) in partition. However, our approach does not rely on alarms, does not need the designer of an analysis to decide a suitable heuristic, and is not restricted to a particular analysis. We lift partitioning from within an analysis to the infeasibility of control flow paths which is a fundamental property of control flow paths independent of an analysis. This allows us to devise an automatic approach to implement a practical trace partitioning. An interesting aspect of our approach is that although it is oblivious to any analysis, it can be seen as dynamic partitioning that uses infeasibility of control flow paths as criteria for partitioning.

7.2.2 Approaches that Use CFG Restructuring

CFG restructuring has also been used to avoid imprecise joins. In particular, Aditya et al. [50] proposed an approach in which at first they identify join points that lead to loss of precision in data flow analysis. Next, they improve the data flow precision by eliminating these join points from control flow graph through control flow graph restructuring.

Ammon et al. [3, 4] proposed a CFG restructuring for avoiding information merging along hot paths i.e., the paths that are more frequently executed in program compared to other paths. They showed that this improves precision of various data flow analysis and increases the opportunities of program optimizations. Their work differs from our work in that they did not address the imprecision added by infeasible control flow paths.

7.2.3 Other Approaches

The work by Yulei [49] improves precision of pointer analysis by eliminating spurious pointer relations generated due to join of pointer information along different paths at joint points. However, they do not address the imprecision added by infeasible paths.

The work on hot path ssa form [44, 25, 34] tries to improve program performance by separating data along hot paths from rest of the data. However their analysis differs from our analysis in that they do a speculative analysis targeted at improving performance through optimizations, and they do not eliminate the effect of infeasible paths.

7.3 Chapter Summary

In this chapter, we discussed two categories of approaches for improving data flow precision: the first category approaches avoid infeasible paths, while the second category approaches avoid imprecise joins. In the former category, FPMFP enriches the state of the art by avoiding the effect of infeasible paths in analysis independent way without using CFG restructuring, which is not possible by existing approaches. The latter category approaches differ from FPMFP in that they do not address the imprecision added by infeasible paths.

Chapter 8

Conclusions and Future Work

We conclude the thesis by reflecting on our ideas in the context of key challenges in achieving scalable and precise data flow analysis (Section 8.1). Next, we discuss the possibilities for future work in this direction (Section 8.2).

8.1 Reflections

An exhaustive data flow analysis answers all queries of a particular type over all possible executions of a program. However, in practice it is hard to see an exhaustive analysis that is efficient and computes a precise result, because of the following issues that often have conflicting solutions.

- Issue 1: to achieve efficiency, the analysis may need to merge the information reaching at a program point along different paths, because the number of paths as well as the corresponding data flow information may be very large or even unbounded.
- Issue 2: to achieve precision, the analysis may need to distinguish between the information reaching from different paths, in order to 1) discard the information reaching along infeasible paths, and 2) avoid potential loss of information because of merging that happens at join points of two or more paths.

These issues have been handled in past as follows: the first issue is handled by merging the information reaching along different paths at their shared path segments in a CFG (instead of keeping the information separate). The approach achieves efficiency because it reduces the amount of information computed. However, the computed information is usually weaker than the actual possible information because of the information merging, and inclusion of information reaching along infeasible paths.

To handle the second issue, two types of approaches have been proposed.

1. The first type of approaches eliminate the effect of infeasible paths from analysis in analysis dependent or analysis independent way. In particular, analysis dependent approaches use selective path sensitive analysis that uses heuristics to selectively distinguish information reaching along some paths, and subsequently discard the information along infeasible paths thereby eliminating effect of infeasible paths from the specific analysis. On the other hand, analysis independent approaches use CFG restructuring to eliminate infeasible paths from CFG itself thereby eliminating the effect of infeasible paths from all analyses performed on restructured CFG.

2. The second type of approaches eliminate the effect of imprecise joins by using either CFG restructuring or trace partitioning. More specifically, in CFG restructuring they eliminate known imprecise joins points from CFG. On the other hand, in trace partitioning they create equivalence classes of program traces using user specified criteria. Next, at imprecise join points, only the information corresponding to traces in same class are merged, and information from traces belonging to different classes is kept separate. Thus, they reduce the effect of imprecise joins to some extent. Both the approaches of removing imprecise joins rely on analysis specific heuristics because imprecise joins are analysis specific.

In this thesis, we proposed a generic approach to remove the effect of infeasible paths from exhaustive data flow analyses. In particular, we introduced the notion of feasible path MFP solutions that separate and discard the data flow values reaching along infeasible paths. A key insight that enabled FPMFP is the following.

> Infeasible paths is a property of programs and not of any particular data flow analysis over the programs. Hence, we can separate the identification of infeasible paths in the CFG of a program from discarding the corresponding data flow values during a data flow analysis.

The above insight allows us to devise a generic technique that intuitively lifts a data flow analysis to multiple parallel and interacting data flow analyses each of which eliminates the effect of a class of known infeasible paths. In particular, this is realized in FPMFP through a two phase approach: in the first phase, we detect minimal infeasible path segments from the input program. In the second phase, we lift the input data flow analysis so as to separate the values that flow through a MIPS from the values that do not flow through the MIPS. Further, the analysis blocks the values that flow through the MIPS at the end of the MIPS thereby eliminating the effect of infeasible paths from data flow analysis.

In our experiments on 30 Benchmarks (selected from Open Source, Industry, and SPEC CPU 2006), we compared precision and performance of FPMFP solutions with that of MFP solutions. Here, we observed up to 13.6% (average 2.87%, geo. mean 1.75%) reduction in the number of def-use pairs in the reaching definitions analysis, and up to 100% (average 18.5%, geo. mean 3%) reduction in the number of alarms in the possibly-uninitialized variables analysis, when using FPMFP solution in place of MFP solution. Further, the FPMFP computation took $2.9\times$ more time compared to the MFP computation. In our experience, this cost-precision trade-off is acceptable considering the corresponding reduction in the manual efforts. Specifically, in program testing testcases need to be written to ensure each def-use pair is covered by at least one testcase. Similarly, for validating the possibly- uninitialized variable alarms requires manual efforts and is error prone.

8.2 Possible Directions for Future Work

We envision three directions for future work as explained below.

8.2.1 Enhancing Precision and Scalability of FPMFP Solutions

We see that the FPMFP solutions can be further improved both in terms of scalability and precision. First, the scalability can be improved by identifying more optimizations that can discard the pairs that do not lead to precision improvement. Second, the precision can be improved by adding handling for wider class of inter-procedural MIPS. However, this should be complemented by corresponding optimizations to keep the distinctions to bare minimum to retain the efficiency of the approach.

8.2.2 Anticipating Precision Gain in FPMFP through Pre-analysis

We see a possibility that the precision gain and scalability of FPMFP on a program can be anticipated using a lightweight pre-analysis of the program. This can help decide whether to apply FPMFP to the program or not. Similar approaches have been proved helpful in earlier attempts of precise and scalable context sensitive program analyses [41, 42, 51, 21]. We believe this will help for FPMFP also.

8.2.3 Separating Program Properties from Analysis Properties

The key idea of FPMFP is recognizing that infeasible paths is a property of programs and not of any data flow analysis. This allowed FPMFP to be generically applicable to all MFP based data flow analyses, without repeatedly identifying infeasible paths for different data flow analyses over same underlying program. On similar lines, more program properties e.g., loops, recursion, infeasible call chains (i.e., function call sequences that appear in callgraph but are never realized in any execution of the program) can be identified to improve the precision of data flow analysis in a generic way.

Appendix A

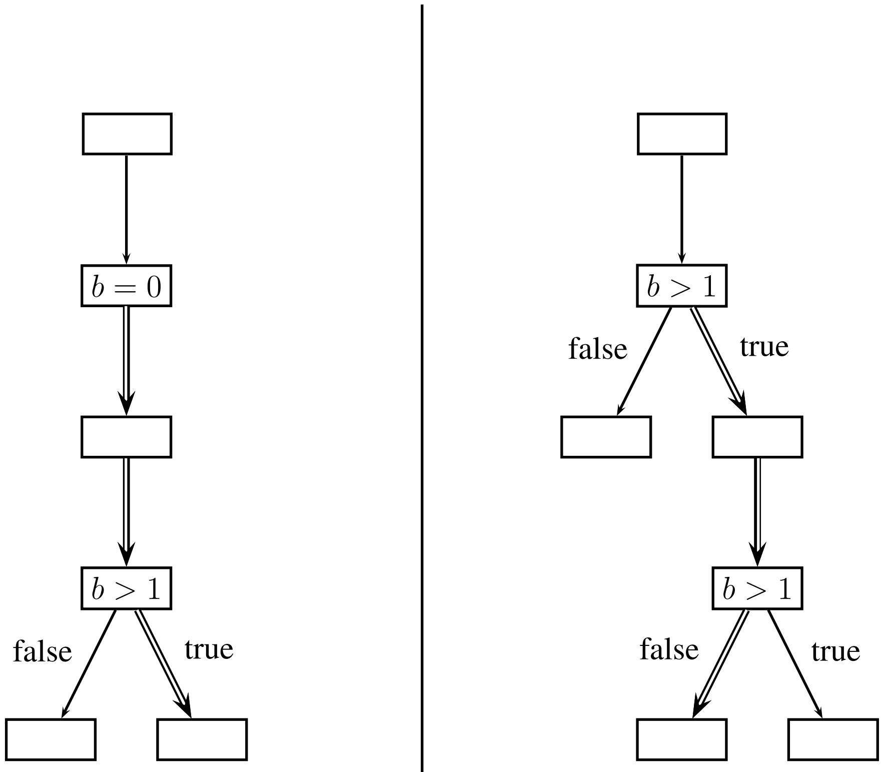
Bodik's Approach for Detecting Minimal Infeasible Path Segments

In this section, we present the algorithms proposed by Bodik et al. [8] for detection of MIPS. We have used these algorithms in the pre-processing stage of FPMFP computation to generate a set of MIPS. The section is organized as follows. First, we give an overview of Bodik's approach of infeasible path detection (Section A.1). Next, we describe the algorithms for detection of infeasible paths in detail (Section A.2).

A.1 Overview of Bodik's Approach

Bodik et al. observed that many infeasible paths are caused by statically detectable correlations between two branch conditions, or between a branch condition and an assignment statement appearing on the paths. Hence, to find infeasible paths arising from branch correlations, they use the following criteria: if the constraint of a conditional edge evaluates to FALSE along any CFP reaching the edge, then the CFP is infeasible.

Example A.1. In Figure A.1a, the branch constraint $(b > 1 = true)$ is in conflict with the assignment statement $b = 0$, hence the path marked with double line arrows is infeasible. Similarly, in Figure A.1b the constraint $(b > 1 = false)$ of the branch at bottom is in conflict with constraint $(b > 1 = true)$ of the branch at top. Hence the path marked with double line arrows is infeasible.



(a) Correlation between assignment statement and branch statement.

(b) Correlation between two branch statements.

Figure A.1: Statically detectable correlations between program statements leading to infeasible paths.

Intuitively, the idea is to enumerate CFPs that reach a conditional edge and identify CFPs along which the edge constraint evaluates to FALSE, because such CFPs are infeasible. In particular they proceed as follows. They start from a branch node, and backward propagate the corresponding constraint (i.e., branch condition) along the incoming paths that reach the node. Here, they evaluate the branch constraint using predefined rules at nodes encountered in backward propagation along a path. A path is labeled as infeasible if the assertions at a node and the constraint have a conflict that is detectable using predefined rules.

Example A.2. For example in Figure A.2, the branching node $n_7 : \ a > 1$ is correlated with node n_1 in that the constraint $a > 1$ evaluates to FALSE at n_1. Therefore, the following path σ that connects n_1 to the TRUE branch of n_7 is infeasible, $\sigma : n_1 \xrightarrow{e_1} n_2 \xrightarrow{e_3} n_5 \xrightarrow{e_6} n_6 \xrightarrow{e_7} n_7 \xrightarrow{e_8} n_8$. Bodik's approach detects this path by backward propagating the constraint $(a > 1)$ from node n_7 to n_1 along σ.

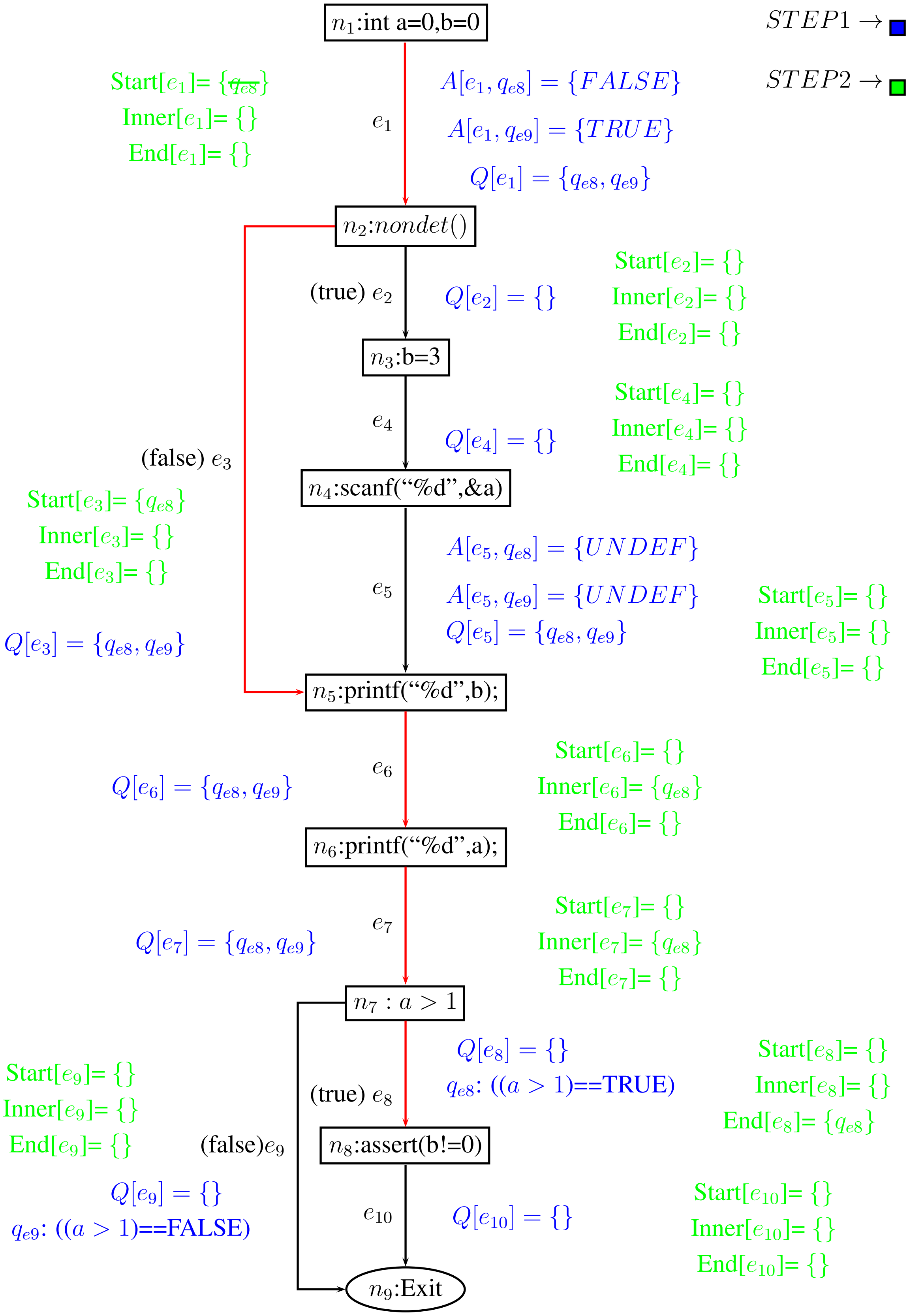


Figure A.2: Example illustrating detection of MIPS using Bodik's Approach

Algorithm 1 Step 1 to detect MIPS that end at a conditional edge e. Let the condition on e be $(v \leq c) = x$, where x is either {TRUE or FALSE}, and $v \leq c$ is the expression at the source node of e. The comments (prefixed with //) are added before some instructions.

```
Initialize Q[e] to {} at each edge $e$; set *worklist* to {}.
*//raise the initial query $q_e : ((v \leq c) = x)$, at each immediate predecessor edge of $e$*
*//these are edges incident on source node of e, and are represented by* predEdge$(e)$
For each $e_m \in$ *predEdge*$(e)$
    *raiseQuery*$(e_m, q_e)$
End For
While *worklist* not empty
    remove pair $(e, q)$ from *worklist*
      *//assume unknown outcome of q at the entry edge (i.e., the outgoing edge of program entry).*
    If $e$ is entry edge
      A[e,q]:=UNDEF
    Else      *//attempt to answer q using assertions generated at source node of e.*
      $answer := resolve(e, source(e), q)$
      If $answer \in \{TRUE, FALSE, UNDEF\}$
        $A[e, q] := answer$
      Else
        For each $e_m \in$ *predEdge*$(e)$
          *raiseQuery*$(e_m, q)$
        End For
      End If
    End If
End While
Procedure *raiseQuery*(Edge $e$, Query $q$)
    *//raise q at e unless previously raised there (check to terminate analysis of loops)*
    If $q \notin Q[e]$
      add $q$ to $Q[e]$
      add pair $(e, q)$ to *worklist*
    End If
```

Algorithm 2 Step 2: mark the edges with MIPS information

```
//begin analysis from the edges where any query was resolved to FALSE in lines
//5 to 19 of Step1
worklist:=$\{e \mid$ a query was resolved to FALSE at edge $e\}$
//raise the initial query at the analyzed edge, to mark end of MIPS.
While worklist not empty
  remove an edge $e$ from the worklist
  //if query resolved to false at this edge then mark
  //the edge as start for corresponding MIPS
  For each query $q_{ex}$ such that $A[e, q_{ex}] = FALSE$
    add $q_{ex}$ to $Start[e]$
    //mark e as start and ex as end edge for MIPS resulting from query $q_{ex}$
    //resolving to FALSE at edge $e$.
    add $q_{ex}$ to $End[ex]$
  End For
  //determine answers for each query that was propagated backward
  For each query $q$ from $Q[e]$ s.t. $q$ was not resolved at $e$
    //move the MIPS started along all predecessor edges of $e$
    If (for all $e_m \in predEdge(e), q \in Start[e_m]$)
      add $q$ to $Start[e]$
      For each $e_m \in predEdge(e)$
        remove $q$ from $Start[e_m]$
      End For
    Else      //make $e$ as the inner edge of started MIPS
      For each $e_m \in predEdge(e)$
        copy $Start[e_m]$ to $Inner[e]$
        copy $Inner[e_m]$ to $Inner[e]$
      End For
    End If
  End For
End While
```

A.2 Detailed Explanation of Bodik's Approach

The Bodik's approach involves two steps as described below.

1. In the first step, the constraints from branch nodes are propagated backwards (towards CFG entry) to identify nodes (if any) at which these constraints evaluate to FALSE.

2. In the second step, a forward traversal of CFG is done to mark the infeasible paths using data from Step 1.

We now explain each of these steps in detail below.

A.2.1 Step 1

The results of Step 1 for the example in Figure A.2 are marked in blue. The details of Step 1 are given in Algorithm 1 and are described below.

At the beginning of Step 1, we raise a query q_e at each conditional edge e such that the query q_e represents the constraint on e. Consequently e is visited in an execution only if q_e is TRUE.

Example A.3. For example, the query for the edge e_8 is q_{e8} : $((a > 1)$=TRUE), and for the edge e_9 is q_{e9} : $((a > 1)$=FALSE). This means the edge e_8 (resp. e_9) will be executed only when q_{e8} (resp. q_{e9}) evaluates to TRUE. Similarly, we raise queries q_{e2} and q_{e3} at conditional edges e_2 and e_3 respectively.

Next, the query raised at an edge e is propagated along each predecessor edge of e say p_e. Here, we try to resolve the query at p_e using the assertions generated at source of p_e if the assertions restrict the value of variables in the query, for example, because of assignment of values to the variables, or branching out of a conditional expression that tests value of the variables. In the former case, if there is an assignment to a variable present in the query then the query resolves to either TRUE, FALSE, or UNDEF (meaning details are not sufficient to resolve the query to TRUE or FALSE) [1].

In the latter case, if a variable present in the query is tested in a conditional at the source of p_e then the query may resolve to TRUE or FALSE or remain unresolved. If the query is

[1] In the UNDEF case, Bodik also proposes the idea of *query substitution* which is not described here.

unresolved at p_e then the query is back propagated to predecessors of p_e and so on [2].

Example A.4. For example, in Figure A.2, we propagate queries q_{e8} and q_{e9} along edge e_7, since source node of e_7 is a *printf* statement, none of the queries are resolved, and hence they are back propagated to edge e_6 and so on. Further, at the edge e_5 both the queries q_{e8} and q_{e9} are resolved to UNDEF because of the statement *scanf*("%d",&a) at the source of e_5. Similarly, at the edge e_1 the query q_{e8} evaluates to FALSE, and q_{e9} evaluates to TRUE because of assignments at the source of e_1.

At the end of Step 1 we have two arrays *Q* and *A*, where *Q* stores the queries raised or backward propagated at an edge, and *A* stores queries resolution at an edge.

Recall that we defined the query q_{ex} such that edge ex executes only if q_{ex} is TRUE. Consequently, if in Step 1, a query q_{ex} is resolved to FALSE at the edge e, then this implies there is an infeasible path segment from edge e to the edge ex. For example in Figure A.2 at edge e_1 the query q_{e8} : $((a > 1)$=TRUE) evaluates to FALSE, hence there is an infeasible path (marked by red edges) that goes from e_1 to e_8. The infeasible path marking happens in Step 2 (Section A.2.2).

A.2.2 Step 2

Step 2 marks the edges in the CFG with the corresponding infeasible paths that pass through the edges. In particular, this is achieved by maintaining three sets namely Start, Inner, and End at each edge e, indicating the MIPS that contain e as start, inner, or end edge respectively. For our running example, the results of Step 2 obtained by using Algorithm 2 are marked in the green color in Figure A.2.

The Step 2 proceeds as follows. For each edge e where any query q_{ex} was resolved to FALSE (in Step 1), we mark e as the start edge of an *infeasible path segment* (IPS) that ends at ex . We also mark ex (i.e., the edge at which query q_{ex} was raised in Step 1) as the end edge of the IPS. For example, in the Figure A.2, edge e_1 is marked as start edge, and the edge e_8 is marked as end edge for IPS that ends at e_8.

[2] Note that only intra-procedural predecessor edges are considered. In case, the source of p_e is a call node and the variables present in the query are modified inside the callee function, then the query is not propagated to predecessor edges of p_e.

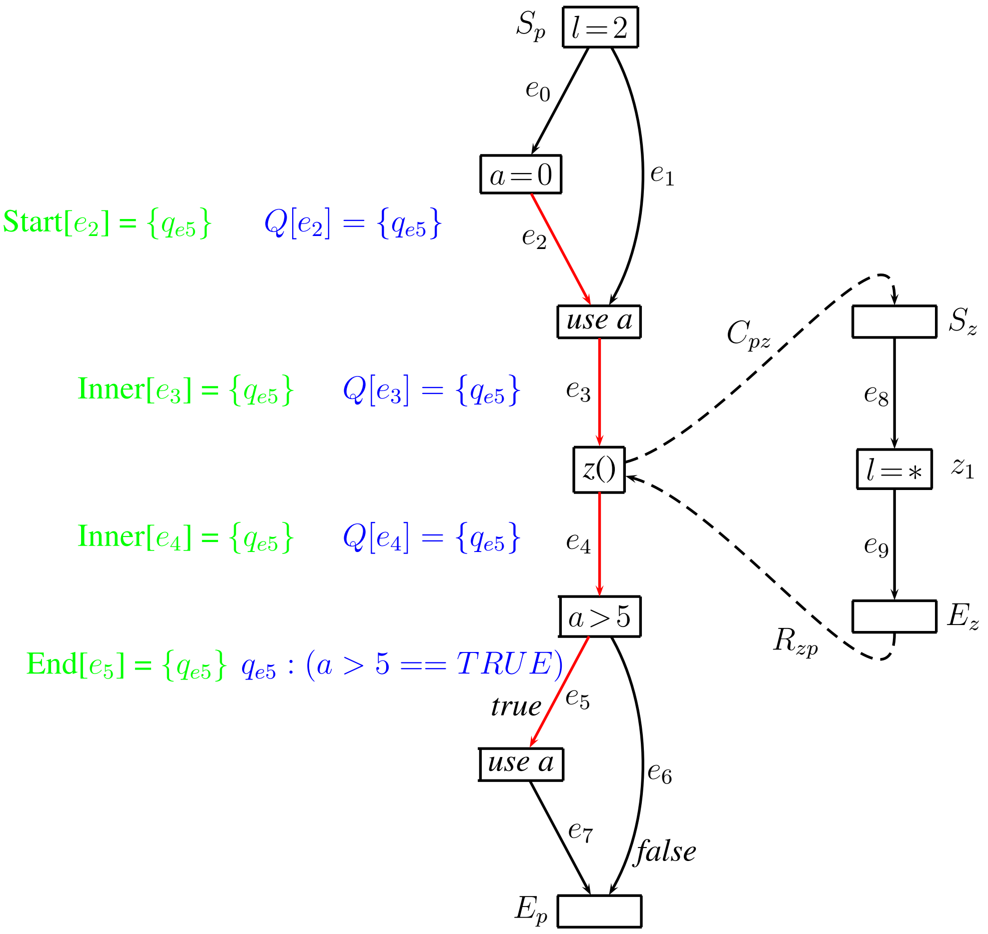


Figure A.3: Detecting a balanced interprocedural MIPS. The variable a is not modified inside the procedure z, hence the query $(q_{e5} : a > 5 = \text{TRUE})$ is back propagated from e_4 to e_3. S, E represent the start and end nodes of a procedure respectively. C_{pz} represents the transfer of control from procedure p to procedure z at a call node, and R_{zp} represents the return of control from z to p. For brevity, cases where Q, $Start$, $Inner$, or End is empty are not shown.

Next, if at some edge e query q_{ex} was propagated backwards (in Step 1) and all its predecessor edges are marked as start edge for some IPS that ends at ex (in Step 2) then e can also be marked as start edge for the IPS. This follows from the fact that the path segment from each predecessor edge p_e to ex is infeasible path segment but path segment from e to ex is also IPS and is shorter than the IPS from p_e to ex. Using this justification, we mark edge e_3 as the start edge for IPS that ends at e_8 in our running example; observe that the IPS that goes from e_3 to e_8 is shorter than the one that goes from e_1 to e_8 (hence we un-mark e_1 as start edge).

On the other hand, if at some edge e query q_{ex} was propagated backwards (in Step 1) and only some of its immediate predecessor edges are marked as start edge or at least one of the predecessor edge is marked as the inner edge for a IPS that ends at ex then we mark e as inner edge for the IPS. For instance, we mark edges $e_6, \ \ e_7$ as inner edges for IPS that ends at e_8 in our running example.

Finally, at the end of Step 2, each path segment $\sigma : e_1 \rightarrow e_2 \rightarrow ... \rightarrow e_n$ in CFG is a MIPS if there exists a query q present in $Start[e_1]$, $End[e_n]$, and $Inner[e_i], 1 < i < n$.

Example A.5. Figure A.3 shows how balanced inter-procedural MIPS (definition 3.1 from Chapter 3) are detected using Bodik's approach. In particular, the variables in the query q_{e5} are not modified inside the procedure z, hence q_{e5} is propagated from e_4 to e_3 to e_2. Subsequently, in Step 2, $e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow e_5$ is marked as a MIPS.

References

[1] Tcs embedded code analyzer, 2017.

[2] Alfred V Aho. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[3] Glenn Ammons and James R Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 72–84, 1998.

[4] Glenn Ammons and James R Larus. Improving data-flow analysis with path profiles. *ACM SIGPLAN Notices*, 39(4):568–582, 2004.

[5] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, Ou Wei, and Aarti Gupta. Slr: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *International Static Analysis Symposium*, pages 238–254. Springer, 2008.

[6] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[7] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *ACM SIGPLAN Notices*, volume 32, pages 146–158. ACM, 1997.

[8] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In *Software EngineeringESEC/FSE'97*, pages 361–377. Springer, 1997.

[9] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):341–395, 1990.

[10] Ting Chen, Tulika Mitra, Abhik Roychoudhury, and Vivy Suhendra. Exploiting branch constraints without exhaustive path enumeration. In *OASIcs-OpenAccess Series in Informatics*, volume 1. Schloss Dagstuhl-Leibniz-Zentrum fr Informatik, 2007.

[11] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *ACM Sigplan Notices*, volume 37, pages 57–68. ACM, 2002.

[12] Dinakar Dhurjati, Manuvir Das, and Yue Yang. Path-sensitive dataflow analysis with iterative refinement. In *International Static Analysis Symposium*, pages 425–442. Springer, 2006.

[13] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Notices*, volume 43, pages 270–280. ACM, 2008.

[14] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. Software validation via scalable path-sensitive value flow analysis. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 12–22. ACM, 2004.

[15] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 37–48, 1995.

[16] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):992–1030, 1997.

[17] Huiquan Gong, Yuwei Zhang, Ying Xing, and Wei Jia. Detecting interprocedural infeasible paths via symbolic propagation and dataflow analysis. In *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, pages 282–285. IEEE, 2019.

[18] Hari Hampapuram, Yue Yang, and Manuvir Das. Symbolic path simulation in path-sensitive dataflow analysis. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 52–58. ACM, 2005.

[19] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS*, volume 98, pages 200–214. Springer, 1998.

[20] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering*, (3):243–250, 1977.

[21] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 13–18, 2017.

[22] David Hedley and Michael A Hennell. The causes and effects of infeasible paths in computer programs. In *Proceedings of the 8th international conference on Software engineering*, pages 259–266. IEEE Computer Society Press, 1985.

[23] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. *ACM SIGPLAN Notices*, 36(5):24–34, 2001.

[24] L Howard Holley and Barry K Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, (1):60–78, 1981.

[25] Smriti Jaiswal, Praveen Hegde, and Subhajit Roy. Constructing hpssa over ssa. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, pages 31–40, 2017.

[26] Swati Jaiswal, Uday P Khedker, and Supratik Chakraborty. Demand-driven alias analysis: Formalizing bidirectional analyses for soundness and precision. *arXiv preprint arXiv:1802.00932*, 2018.

[27] Swati Jaiswal, Uday P Khedker, and Supratik Chakraborty. Bidirectionality in flow-sensitive demand-driven analysis. *Science of Computer Programming*, 190:102391, 2020.

[28] Shujuan Jiang, Hongyang Wang, Yanmei Zhang, Meng Xue, Junyan Qian, and Miao Zhang. An approach for detecting infeasible paths based on a smt solver. *IEEE Access*, 7:69058–69069, 2019.

[29] John B Kam and Jeffrey D Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

[30] John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta informatica*, 7(3):305–317, 1977.

[31] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.

[32] Elgin Wei Sheng Lee. Infeasible path detection and code pattern mining. 2019.

[33] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 299–314. Springer, 2012.

[34] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative optimizations. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1(3):247–271, 2004.

[35] N Malevris, DF Yates, and A Veevers. Predictive metric for likely feasibility of program paths. *Information and Software Technology*, 32(2):115–118, 1990.

[36] Abdalla Wasef Marashdih, Zarul Fitri Zaaba, and Herman Khalid Omer. Web security: detection of cross site scripting in php web application using genetic algorithm. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 8(5), 2017.

[37] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*, pages 5–20. Springer, 2005.

[38] Steven S Muchnick and Neil D Jones. *Program flow analysis: Theory and applications*, volume 196. Prentice-Hall Englewood Cliffs, New Jersey, 1981.

[39] Frank Mueller and David B Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 56–66, 1995.

[40] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Repositioning of static analysis alarms. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 187–197, 2018.

[41] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. *ACM SIGPLAN Notices*, 49(6):475–484, 2014.

[42] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective x-sensitive analysis guided by impact pre-analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(2):1–45, 2015.

[43] Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

[44] Subhajit Roy and YN Srikant. The hot path ssa form: Extending the static single assignment form for speculative optimizations. In *International Conference on Compiler Construction*, pages 304–323. Springer, 2010.

[45] Chen Rui. *Infeasible path identification and its application in structural test*. PhD thesis, Beijing: Institute of Computing Technology of Chinese Academy of Sciences, 2006.

[46] Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for wcet analysis. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11. IEEE, 2016.

[47] Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences , 1978.

[48] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. *ACM SIGPLAN Notices*, 40(10):59–76, 2005.

[49] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. Spas: Scalable path-sensitive pointer analysis on full-sparse ssa. In *Asian Symposium on Programming Languages and Systems*, pages 155–171. Springer, 2011.

[50] Aditya Thakur and R Govindarajan. Comprehensive path-sensitive data-flow analysis. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 55–63. ACM, 2008.

[51] Shiyi Wei and Barbara G Ryder. Adaptive context-sensitive analysis for javascript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[52] Elaine J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, 1988.

[53] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *ACM SIGSOFT Software Engineering Notes*, 28(5):327–336, 2003.

[54] Cheer-Sun D Yang, Amie L Souter, and Lori L Pollock. All-du-path coverage for parallel programs. *ACM SIGSOFT Software Engineering Notes*, 23(2):153–162, 1998.

[55] Fuping Zeng, Wenjing Liu, and Xiaodong Gou. Type analysis and automatic static detection of infeasible paths. In *International Conference on Geo-Spatial Knowledge and Intelligence*, pages 294–304. Springer, 2017.

[56] Dalin Zhang, Dahai Jin, Yunzhan Gong, and Hailong Zhang. Diagnosis-oriented alarm correlations. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 172–179. IEEE, 2013.

[57] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 197–208, 2008.

[58] Honglei Zhu, Dahai Jin, Yunzhan Gong, Ying Xing, and Mingnan Zhou. Detecting interprocedural infeasible paths based on unsatisfiable path constraint patterns. *IEEE Access*, 7:15040–15055, 2019.