A Unified Model for Context-Sensitive Program Analyses

The Blind Men and The Elephant

SWATI JAISWAL, Visvesvaraya National Institute of Technology, India
UDAY P. KHEDKER, Indian Institute of Technology Bombay, India
ALAN MYCROFT, University of Cambridge, UK

Context-sensitive methods of program analysis increase the precision of interprocedural analysis by achieving the effect of call inlining. These methods have been defined using different formalisms and hence appear as algorithms that are very different from each other. Some methods traverse a call graph top-down whereas some others traverse it bottom-up first and then top-down. Some define contexts explicitly whereas some do not. Some of them directly compute data flow values while some first compute summary functions and then use them to compute data flow values. Further, different methods place different kinds of restrictions on the data flow frameworks supported by them. As a consequence, it is difficult to compare the ideas behind these methods in spite of the fact that they solve essentially the same problem. We argue that these incomparable views are similar to those of blind men describing an elephant called context sensitivity, and make it difficult for a non-expert reader to form a coherent picture of context-sensitive data flow analysis.

We bring out this whole-elephant view of context sensitivity in program analysis by proposing a unified model of context sensitivity which provides a clean separation between computation of contexts and computation of data flow values. Our model captures the essence of context sensitivity and defines simple soundness and precision criteria for context-sensitive methods. It facilitates declarative specifications of context-sensitive methods, insightful comparisons between them, and reasoning about their soundness and precision. We demonstrate this by instantiating our model to many known context-sensitive methods.

CCS Concepts: • **Theory of computation** → **Program analysis**;

Additional Key Words and Phrases: Interprocedural data flow analysis, interprocedurally valid paths, context sensitivity, flow sensitivity

ACM Reference Format:
Swati Jaiswal, Uday P. Khedker, and Alan Mycroft. 2021. A Unified Model for Context-Sensitive Program Analyses: The Blind Men and The Elephant. *ACM Comput. Surv.* 1, 1, Article 1 (January 2021), 37 pages. https://doi.org/10.1145/3456563

1 INTRODUCTION

The precision and efficiency of program analysis are influenced significantly by the abstraction of control flow both at the intraprocedural, and the interprocedural level.

Authors' addresses: Swati Jaiswal, swatijaiswal@cse.vnit.ac.in, Visvesvaraya National Institute of Technology, India; Uday P. Khedker, uday@cse.iitb.ac.in, Indian Institute of Technology Bombay, India; Alan Mycroft, Alan.Mycroft@cl.cam.ac.uk, University of Cambridge, UK.

0360-0300/2021/1-ART1 $15.00
https://doi.org/10.1145/3456563

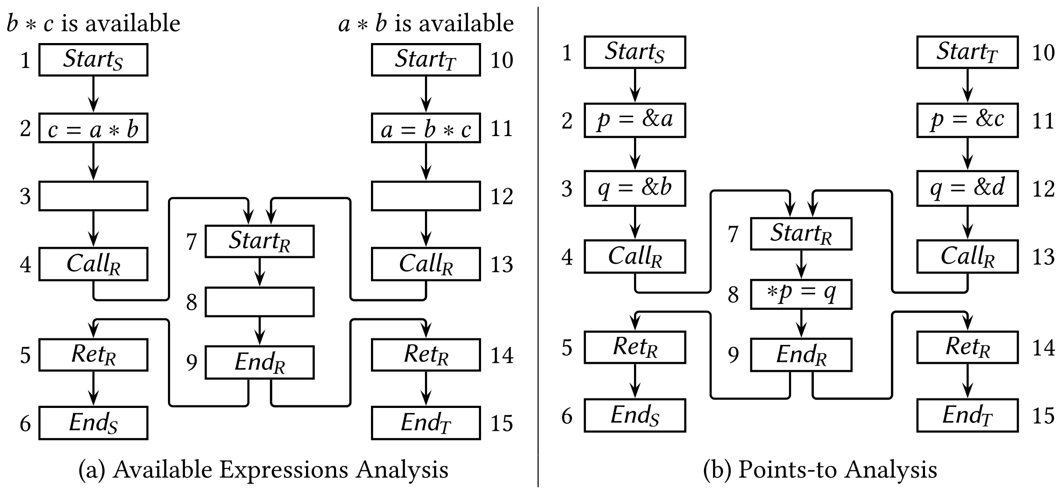


Fig. 1. Example illustrating the imprecision introduced if interprocedurally invalid paths are considered

At the intraprocedural level, if an analysis considers the control flow of a procedure and computes distinct information for each program point, then the analysis is flow-sensitive. If it disregards the control flow and computes a single piece of information that is valid at each program point, then it is flow-insensitive. The former computes more precise information than the latter.

At the interprocedural level, a context-sensitive analysis tries to achieve the effect of inlining of callee procedures by ensuring that the result of interprocedural analysis matches the result obtained after inlining callee procedures. Actual inlining of procedures is undesirable in most cases (because it could increase the size of the code exponentially) and is infeasible in the case of recursion. The effect of inlining can be obtained by (a) inlining a summary of the procedure instead of inlining the procedure, or (b) traversing the call graph by performing proper call and return matchings. A context-insensitive analysis is less precise than a context-sensitive analysis.

Context sensitivity makes an analysis more precise but affects the efficiency of the analysis. Many methods have been proposed to achieve context sensitivity with the primary motive of increasing efficiency. These methods use a variety of formalisms that are very different from each other and hence these methods appear very dissimilar in spite of the fact that they solve the same problem. Besides, they are defined algorithmically and it is very difficult for a non-expert reader to compare them. In this paper we survey many of the known context-sensitive methods in order to bring out the similarities and differences between them.

1.1 The Need for Context Sensitivity

A program is represented by a control flow graph (CFG) in which nodes contain (are labelled with) statements and edges represent control transfers (see Section 2.1 for a formal definition). A CFG can contain multiple procedures; some authors call this a ‘supergraph’ or an ‘interprocedural CFG’.

We write $n : s$ to indicate that node n is labelled with statement $s \in S$. We assume that each call to procedure Q has been split into a *call node* $m : Call_Q$ (with no intraprocedural control flow successors) and a paired *return node* $n : Ret_Q$ (with no intraprocedural control flow predecessors). Each procedure Q has a unique $n : Start_Q$ and a unique $m : End_Q$. When the statement is not required, we simply write the nodes: m, n, etc.

Consider the CFG of a program shown in Figure 1 in which procedure R is invoked from procedures S and T from call nodes $4 : Call_R$ and $13 : Call_R$, respectively. For a precise interprocedural analysis, the effect of inlining can be obtained by traversing the CFG. However, such traversals could introduce interprocedurally invalid paths when the calls and returns do not match properly. For example, path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4\!:\!\mathit{Call}_R \rightarrow 7\!:\!\mathit{Start}_R \rightarrow 8 \rightarrow 9\!:\!\mathit{End}_R \rightarrow 14\!:\!\mathit{Ret}_R \rightarrow 15$ is invalid because the return node $14\!:\!\mathit{Ret}_R$ does not correspond to the call node $4\!:\!\mathit{Call}_R$ indicating that the call does not return to the actual call point. We define the notion of interprocedurally valid path formally in Section 2.1. To see the imprecision caused by interprocedurally invalid paths, consider an interprocedural available expressions analysis for the procedures in Figure 1(a) for determining the expressions that are available in nodes 6 and 15. As illustrated in the table below, if we include interprocedurally invalid paths, the effect of the assignment in node 2 reaches node 15 and the effect of node 11 reaches node 6 making both expressions unavailable at nodes 6 and 15.

Node	Interprocedural path reaching the node	Valid?	Availability
6	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4\!:\!\mathit{Call}_R \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 5\!:\!\mathit{Ret}_R \rightarrow 6$	Yes	$\{a * b\}$
	$10 \rightarrow 11 \rightarrow 12 \rightarrow 13\!:\!\mathit{Call}_R \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 5\!:\!\mathit{Ret}_R \rightarrow 6$	No	$\{b * c\}$
15	$10 \rightarrow 11 \rightarrow 12 \rightarrow 13\!:\!\mathit{Call}_R \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 14\!:\!\mathit{Ret}_R \rightarrow 15$	Yes	$\{b * c\}$
	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4\!:\!\mathit{Call}_R \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 14\!:\!\mathit{Ret}_R \rightarrow 15$	No	$\{a * b\}$

A similar effect is seen for points-to analysis; with interprocedurally invalid paths, we get spurious points-to pairs: (a, d) at node 6 and (c, b) at node 15. Thus in order to avoid imprecision, we need to exclude interprocedurally invalid paths.

1.2 Procedure Summaries for Context-Sensitive Interprocedural Analysis

The primary requirement of context sensitivity is to achieve the effect of call inlining during the analysis. Every context-sensitive method summarizes a procedure in some form or the other and uses the summary in the callers of the procedure. A summary may be computed in one of two ways: (a) By analyzing a procedure for a particular incoming data flow value by traversing the call graph *top-down* and propagating the information from callers to callees. The resulting summary depends on the context. (b) Summarizing the procedure independently of the information being propagated from callers to callee. The resulting summary is used to replace the calls in the callers to construct the summary of the callers. Thus the call graph is traversed *bottom-up*.

The summary function constructed by a top-down traversal can also be viewed as an *extensional representation* of the function which is enumeration of key-value pairs. Consider a function $\mathit{square} : \text{Int} \rightarrow \text{Int}$ which takes an integer and returns the square of the input. Then the enumeration of key-value pairs for function *square* are $\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9, \ldots\}$. The summary function constructed by a bottom-up traversal can be viewed as an *intensional representation* as a parameterized expression such as $\lambda x.x^2$ or simply x^2 for function *square*. Consider procedure Q containing a statement sequence $x = y + 1; y = 2;$ with contexts σ_1 and σ_2 reaching the procedure with values $y \mapsto \bot$ in σ_1 and $y \mapsto 8$ in σ_2. Then, for constant propagation the extensional representation of the summary for procedure Q, denoted by $\mathbb{S}_Q$, is $\{(\sigma_1, \{x \mapsto \bot, y \mapsto 2\}), (\sigma_2, \{x \mapsto 9, y \mapsto 2\})\}$. The intensional representation of the summary is $\mathbb{S}_Q(X) = X[x \mapsto X(y) + 1, y \mapsto 2]$ where X is a map from variables to their values, $X(y)$ gives the value of y in the map, and $X[\ldots]$ denotes how the values of specific variables in map X are updated.

1.3 Contributions and Organization of the Paper

The known context-sensitive methods use very different formalisms and appear very dissimilar in spite of the fact that they attempt to solve the same problem. We probe the notion of context sensitivity in these methods and propose a unified model of context-sensitive methods of data flow analysis that brings out a whole-elephant view of context sensitivity.[1] Our model provides a clean separation between computation of contexts (captured by an *abstract context structure*) and computation of data flow values (captured by an *abstract value structure*). The model allows us to identify simple formal criteria for soundness and precision of a method instantiated in the model.

We model most of the known context-sensitive methods using our unified model and show how they satisfy the soundness and precision criteria. This modelling also uncovers the hidden notion of contexts in some methods, facilitates insightful comparisons between different methods, and facilitates cross fertilization of ideas and suggest interesting improvements in the known methods.

In this work we only consider forward data flow analyses for simplicity of exposition. Backward flows are duals of forward flows and our model can accommodate them easily.

The rest of the paper is organized as follows: Section 2 provides a brief review of data flow analysis. Section 3 describes various methods of context-sensitive analysis by using a running example to bring out the differences and similarities between them. Section 4 describes the proposed unified model by defining abstract context structure and abstract value structure. It also instantiates the model to all methods surveyed in Section 3. It defines soundness and precision criteria and shows how the surveyed methods satisfy the criteria. Section 5 presents the bigger picture describing ideas that we have not modelled. Section 6 concludes the paper.

2 A BRIEF REVIEW OF DATA FLOW ANALYSIS

This section reviews data flow analysis briefly and defines some terms and notations.

2.1 Program Representations for Data Flow Analysis

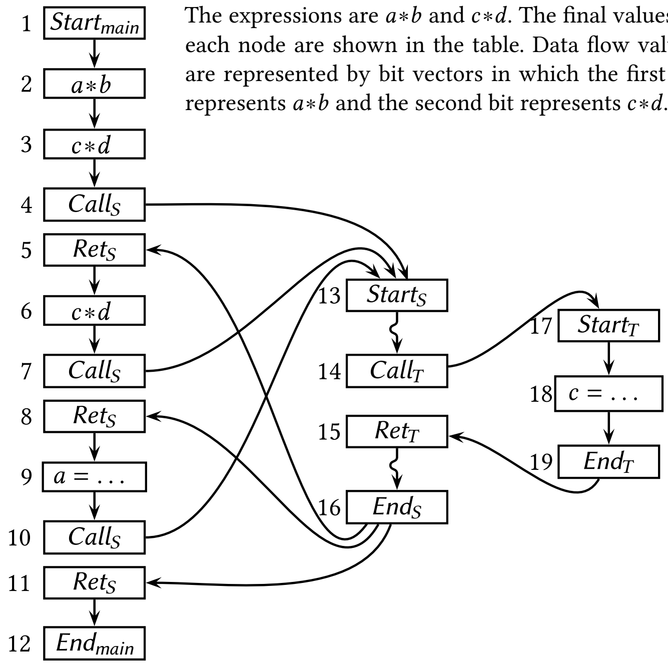
Let $\mathbb{P}\text{roc}$ be the set of procedures in a program. Assuming a set S of statements, a *control flow graph*, or CFG, $(\mathbb{N}, \mathbb{E} \subseteq \mathbb{N} \times \mathbb{N}, \mathcal{L} : \mathbb{N} \to S)$ is a labelled directed graph whose nodes $n \in \mathbb{N}$ are labelled with statements $\mathcal{L}(n) \in S$. We write $n : s$ to indicate that node n is labelled with statement $s \in S$. We write $\mathbb{N}_Q$ for the set of nodes in procedure $Q \in \mathbb{P}\text{roc}$; the sets $\mathbb{N}_Q$ partition $\mathbb{N}$ (i.e. are exclusive and exhaustive). Edges $\mathbb{E}$ are discussed below.

We assume each call to procedure Q has been split into a *call node* $m\,{:}\,Call_Q$ (having no intraprocedural control flow successors) and a paired *return node* $n\,{:}\,Ret_Q$ (having no intraprocedural control flow predecessors). Given $n\,{:}\,Ret_Q$ we write $\widehat{n}$ for its associated $m\,{:}\,Call_Q$. We write $\mathbb{C} = \{n \in \mathbb{N} \mid n\,{:}\,Call_Q \text{ for some Q}\}$; this is the set of *call sites*. Each procedure Q has a unique $n\,{:}\,Start_Q$ and, for the convenience of modelling, a unique $m\,{:}\,End_Q$. Hence, by abuse of notation, we sometimes write $Start_Q$ or End_Q when a node n is formally required, e.g. In_{Start_Q}.

Control flow in the program is represented by the set of edges $\mathbb{E}$. Edges (m, n) are classified by three predicates: $\mathsf{IE}(m, n)$, $\mathsf{CE}(m, n)$, and $\mathsf{RE}(m, n)$ as intraprocedural edge, call edge and return edge respectively. $\mathsf{IE}(m, n)$ represents intraprocedural flow. $\mathsf{CE}(m, n)$ holds for edge (m, n), when node $m\,{:}\,Call_Q$ contains a call to some procedure Q and node $n\,{:}\,Start_Q$ is the start node of the callee procedure Q. $\mathsf{RE}(m, n)$ holds for edge (m, n), when node $m\,{:}\,End_Q$ is the end node of some procedure Q and node $n\,{:}\,Ret_Q$ is the return point of some call to Q (here $\widehat{n}\,{:}\,Call_Q$).

For each node m exactly one of $\mathsf{IE}(m, n_1)$, $\mathsf{CE}(m, n_2)$, $\mathsf{RE}(m, n_3)$ and $m\,{:}\,End_{main}$ holds, and similarly for each node n exactly one of $\mathsf{IE}(m_1, n)$, $\mathsf{CE}(m_2, n)$, $\mathsf{RE}(m_3, n)$ and $n\,{:}\,Start_{main}$ holds. We write $pred(n)$ for $\{m \mid \mathsf{IE}(m, n)\}$, and define predicate $\mathsf{IntraNode}(n)$ to assert that node n has only intraprocedural predecessors: $\mathsf{IntraNode}(n) \Leftrightarrow \exists m.\mathsf{IE}(m, n)$.

[1]Our use of the "blind men" metaphor is only to highlight the difficulty of a non-expert reader in forming a consistent and coherent view of these methods and does not question the wisdom of the designers of the methods in any way. This metaphor refers to the parable in which people see different parts and are unable to visualize the whole [1, 28]. We strongly believe that the story of context-sensitive methods is no different.



The expressions are $a*b$ and $c*d$. The final values at each node are shown in the table. Data flow values are represented by bit vectors in which the first bit represents $a*b$ and the second bit represents $c*d$.

Node	In	Out
1	00	00
2	00	10
3	10	11
4	11	11
5	10	10
6	10	11
7	11	11
8	10	10
9	10	00
10	00	00
11	00	00
12	00	00
13	00	00
14	00	00
15	00	00
16	00	00
17	00	00
18	00	00
19	00	00

Fig. 2. A motivating example of interprocedural available expressions analysis.

For simplicity, we assume that there are no indirect calls (i.e. no calls through function pointers, virtual function calls or higher-order function calls); this gives the *at-most-one-image* property $\mathsf{CE}(m,n) \land \mathsf{CE}(m,n') \Rightarrow n = n'$. This property is equivalent to $\mathsf{RE}(m,n)$ being *injective*: $\mathsf{RE}(m,n) \land \mathsf{RE}(m',n) \Rightarrow m = m'$.

We require the existence of a unique *main* procedure, assumed non-recursive. Also, we assume that all nodes are reachable (via IE, CE and RE edges forming a possibly infeasible path) from $Start_{main}$ (i.e., all procedures are callable) and End_{main} is similarly reachable from every node. Classical intraprocedural analyses are recovered by requiring that the CFG contains only procedure *main* and no call instructions.

Paths. We write $Paths(n)$ for the set of control flow paths from $Start_{main}$ to n defined as follows:

$$\frac{}{n \in Paths(n)} \text{ if } n\!:\!Start_{main} \qquad \frac{\pi \cdot m \in Paths(m)}{\pi \cdot m \cdot n \in Paths(n)} \text{ if } (m,n) \in \mathbb{E} \tag{1}$$

In other words, paths π are sequences of $k+1$ ($k \geq 0$) nodes representing k edges such that every pair of consecutive nodes forms an edge in the CFG. In the presence of loops, the length of a path π is unbounded and $Paths(n)$ may be infinite.

For intraprocedural analysis, it is assumed that there are no procedure calls in any path. However, using *Paths* naively on a CFG containing calls causes, in general, non-executable paths and imprecision in analysis (e.g. in Figure 2 *main* calls procedure S thrice, so *Paths* as defined above includes cyclic paths where a call at the second call site (node 7) returns to the first call site (node 5)). Hence, we now refine the definition.

Call Strings and Interprocedurally Valid Paths. A *call string*, denoted σ, is a sequence of nodes n of the form $n\!:\!Call_Q$. The set of all call strings is denoted by $\mathbb{\Sigma}$, with the empty call string written ϵ; for recursive programs, $\mathbb{\Sigma}$ is infinite. The set of *interprocedurally valid paths* reaching node n

from $Start_{main}$ is denoted $IVPC(n)$ and consists of pairs (π, σ) where π is a sequence of nodes and σ is a sequence of call nodes representing active calls. It greatly simplifies the formulation of interprocedurally valid path to use pairs (π, σ) here; the traditional definition is available as π.

Then, $IVPC(n)$ is defined as follows:

$$\frac{}{(n,\ \epsilon) \in IVPC(n)} \text{ if } n\!:\!Start_{main} \qquad \frac{(\pi\cdot m,\ \sigma) \in IVPC(m)}{(\pi\cdot m\cdot n,\ \sigma\cdot m) \in IVPC(n)} \text{ if } \mathsf{CE}(m,n)$$
$$\frac{(\pi\cdot m,\ \sigma) \in IVPC(m)}{(\pi\cdot m\cdot n,\ \sigma) \in IVPC(n)} \text{ if } \mathsf{IE}(m,n) \qquad \frac{(\pi\cdot m,\ \sigma\cdot \widehat{n}) \in IVPC(m)}{(\pi\cdot m\cdot n,\ \sigma) \in IVPC(n)} \text{ if } \mathsf{RE}(m,n) \tag{2}$$

In the presence of loops or recursive calls, a path π is unbounded and $IVPC(n)$ is infinite.

For convenience, we define predicate $ReachC_n(\sigma)$ to assert that call string σ reaches node n.

$$ReachC_n(\sigma) \Leftrightarrow \exists \pi \text{ such that } (\pi\cdot n, \sigma) \in IVPC(n)$$

For the motivating example in Figure 2, $IVPC(5) = \{(\pi, \sigma)\}$ where π is $1\cdot 2\cdot 3\cdot 4\cdot 13\cdot 14\cdot 17\cdot 18\cdot 19\cdot 15\cdot 16\cdot 5$ and σ is ϵ. There is only one interprocedurally valid path π reaching node 5 and the call string σ reaching node 5 is ϵ. Similarly, $IVPC(17) = \{(\pi_1, \sigma_1), (\pi_2, \sigma_2), (\pi_3, \sigma_3)\}$ where

$\pi_1 = 1\cdot 2\cdot 3\cdot 4\cdot 13\cdot 14\cdot 17$ $\qquad \sigma_1 = 4\cdot 14$
$\pi_2 = 1\cdot 2\cdot 3\cdot 4\cdot 13\cdot 14\cdot 17\cdot 18\cdot 19\cdot 15\cdot 16\cdot 5\cdot 6\cdot 7\cdot 13\cdot 14\cdot 17$ $\qquad \sigma_2 = 7\cdot 14$
$\pi_3 = 1\cdot 2\cdot 3\cdot 4\cdot 13\cdot 14\cdot 17\cdot 18\cdot 19\cdot 15\cdot 16\cdot 5\cdot 6\cdot 7\cdot 13\cdot 14\cdot 17$ $\qquad \sigma_3 = 10\cdot 14$
$\quad 18\cdot 19\cdot 15\cdot 16\cdot 8\cdot 9\cdot 10\cdot 13\cdot 14\cdot 17$

There are three interprocedurally valid paths reaching node 17; the call strings along these three paths are $4\cdot 14$, $7\cdot 14$, and $10\cdot 14$.

2.2 Mathematical Background

A *complete lattice* L is a set L along with a partial order $\sqsubseteq$ which has least upper bounds (denoted $\sqcup$) and greatest lower bounds (denoted $\sqcap$) of all subsets of L. It therefore has a greatest element $\top$ and a least element $\bot$. Our formulation for data-flow analysis uses L as a ($\sqcap$-)semilattice. A *chain* in L is a totally ordered subset of L. A lattice L is of *finite height* if all its chain are finite. We only consider complete lattices, and so tend to omit the word 'complete'.

We write $\mathbb{B} = \{\bot, \top\}$. Given a set S, its powerset 2^S ordered by $\subseteq$ or $\supseteq$ is a complete lattice. Given a set S and a lattice L then the set of function $S \longrightarrow L$ is also a lattice with ordering $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in S$.

Given lattices L and M a function $f : L \longrightarrow M$ is

- *monotonic* if for every pair $x \sqsubseteq x' \in L$ we have $f(x) \sqsubseteq f(x')$;
- $\sqcap$-*continuous* if for every non-empty chain $X \subseteq L$ we have $f(\bigsqcap X) = \bigsqcap_{x\in X} f(x)$;
- $\sqcap$-*distributive* if for every non-empty $X \subseteq L$ we have $f(\bigsqcap X) = \bigsqcap_{x\in X} f(x)$.

Distributivity implies continuity implies monotonicity, but in general their converses do not hold. However, continuity is equivalent to monotonicity for finite-height lattices.

Note that there are many bijections between 2^S and $S \longrightarrow \{0, 1\}$ as *sets*, and it often makes sense to treat them as two representations of the same thing. If we treat 2^S as a *lattice* ordered by $\subseteq$ (respectively $\supseteq$) then it is isomorphic to $S \longrightarrow \mathbb{B}$ with $\{\} \leftrightarrow \bot$, (respectively $\{\} \leftrightarrow \top$). Similarly, $\{0, 1\}$ ordered with $0 < 1$ is isomorphic to $\mathbb{B}$ ($0 \leftrightarrow \bot, 1 \leftrightarrow \top$), but if we order it with $1 < 0$ then ($1 \leftrightarrow \bot, 0 \leftrightarrow \top$),

Let S and T be sets and $\mathbf{0}$ represent a distinguished element not present in S. Then every distributive function $f : 2^S \longrightarrow 2^T$ is characterised by a function $g : (S \cup \mathbf{0}) \longrightarrow 2^T$. To see this: Take $g(\mathbf{0}) = f\{\}$ and $g(x) = f\{x\}$ otherwise. Then the function f' defined by $f'(\{\}) = g(\mathbf{0})$ and, for non-empty $X \in 2^S$, by $f'(X) = \bigcup_{x\in X} g(x)$ is equal to f.

A distributive function $f : 2^S \longrightarrow 2^S$ can also be characterised [40] by its *representation relation* $R_f \subseteq (S \cup \mathbf{0}) \times (S \cup \mathbf{0})$ given by

$$R_f \;=\; \{\langle s_1, s_2\rangle \mid s_2 \in f(\{s_1\}) \wedge s_2 \notin f(\emptyset)\} \;\cup\; \{\langle \mathbf{0}, s_2\rangle \mid s_2 \in f(\emptyset)\} \;\cup\; \{\langle \mathbf{0}, \mathbf{0}\rangle\}$$

This is useful because a flow-function meet can be represented by $\cup$, flow-function composition by relation composition, and the initial $\top$ flow-function by the empty relation.

We now define, given set S, the property of a function $f : 2^S \longrightarrow 2^S$ being *separable*, that is when it can be represented as a family of *basis functions* $f|_{s \in S} : \{0,1\} \longrightarrow \{0,1\}$ (these are $\lambda b.0$, $\lambda b.1$ and $\lambda b.b$; we omit $\lambda b.1{-}b$ as this is not monotonic when we add an order to $\{0,1\}$). We use $f|_s$ to construct $f' : 2^S \longrightarrow 2^S$ (which attempts to recover f):

$$f|_s(b) = \begin{cases} 0 & s \notin f(S) \\ 1 & s \in f(\{\}) \\ b & \text{otherwise} \end{cases}$$

$$f'(X) = \{x \in X \mid f|_x(0) = 1\}$$

We say that f is *separable* when $f = f'$. The following is immediate: Given set S, every separable function $f : 2^S \longrightarrow 2^S$ is characterised by the functions $f|_{s \in S} : \{0,1\} \longrightarrow \{0,1\}$.

Note that separability implies distributivity, but the converse does not in general hold. Separability captures the familiar set-based gen-kill properties for simple liveness (but note that the flow functions for strong liveness are distributive but not separable).

2.3 Intraprocedural Solutions of Data Flow Analysis

We use $\mathbb{L}$ for the lattice of data flow values. Each node has an associated *flow function* $f_n : \mathbb{L} \longrightarrow \mathbb{L}$.

2.3.1 Intraprocedural Meet Over Paths Solution. The classical intraprocedural definition of the (forward) meet-over-paths solution (*MoP*) at node n (of procedure *main* containing no calls) is:

$$MoP(n) \;=\; \bigsqcap_{\pi \,\in\, Paths(n)} f_\pi(BI) \tag{3}$$

where *BI* (short for Boundary Information) denotes the external data flow information reaching *main*. We define f_π as the composition of the flow functions of the nodes appearing in $Paths(n)$ up to, but not including, n. This coheres with the use of In_n for data flow variables in a forwards analysis not including the effects of f_n.

2.3.2 Intraprocedural Maximum Fixed Point Solution. Since *MoP* solution is uncomputable in general, data flow analysis is performed by computing the *maximum fixed point* solution defined in terms of data flow equations. These equations relate data-flow values $\mathsf{In}_n \in \mathbb{L}$ at node $n \in \mathbb{N}$ with those at adjacent nodes; the desired *MFP* solution is the maximum fixed point of the equations:

$$\mathsf{In}_n = \begin{cases} BI & n\!:\!Start_{main} \\ \displaystyle\bigsqcap_{p \in pred(n)} f_p(\mathsf{In}_p) & \text{otherwise} \end{cases} \tag{4}$$

In_n is a sound approximation of $MoP(n)$ in that $\forall n \in \mathbb{N}, \mathsf{In}_n \sqsubseteq MoP(n)$.

2.4 Interprocedural Solutions of Data Flow Analyses

We assume that the flow function associated with *Start*, *End*, *Call* and *Ret* are the identity (i.e., for simplicity in this version we do not handle formal parameters or return values).

2.4.1 Meet Over Interprocedurally Valid Paths Solution. Consider $(\pi, \sigma) \in IVPC(n)$ where $\pi = m_1 \cdot \ldots m_k \cdot n$. As above, f_π denotes the composition $f_{m_k} \circ \ldots \circ f_{m_1}$ of the flow functions of nodes appearing in path π up to, but not including, n.

With these provisions, we extend Definition (3) to *meet-over-interprocedurally-valid-paths* (denoted *MoIVPC*) as follows:

$$MoIVPC(n) \quad = \quad \bigsqcap_{(\pi,\sigma)\, \in\, IVPC(n)} f_\pi(BI) \tag{5}$$

2.4.2 Interprocedural Maximum Fixed Point Solution. We incorporate context-sensitivity and extend Equation (4) to define the *Interprocedural Maximum Fixed Point solution using call strings* (*MFPC*). The data flow variables InC and OutC in *MFPC* are associated with pairs (n, σ) where $\sigma \in \Sigma$ is a call string. Rather than writing $\mathsf{InC}_{(n,\sigma)}$ it is convenient to use curried notation: $\mathsf{InC}_n : \Sigma \to \mathbb{L}$. Thus $\mathsf{InC}_n(\sigma) \in \mathbb{L}$ denotes the data flow value for a context $\sigma \in \Sigma$ at node $n \in \mathbb{N}$.

$$\mathsf{InC}_n(\sigma) = \begin{cases} BI & n\!:\!Start_{main} \wedge ReachC_n(\sigma) \\ \mathsf{InC}_m(\sigma') & n\!:\!Start_Q \wedge Q \neq main \wedge ReachC_n(\sigma) \\ & \text{where } m, \sigma' \text{ (uniquely) satisfy } \sigma = \sigma' \cdot m. \\ \mathsf{InC}_{End_Q}(\sigma \cdot \widehat{n}) & n\!:\!Ret_Q \wedge ReachC_n(\sigma) \\ \bigsqcap_{p \in pred(n)} f_p(\mathsf{InC}_p(\sigma)) & \mathsf{IntraNode}(n) \wedge ReachC_n(\sigma) \\ \top & \neg ReachC_n(\sigma) \end{cases} \tag{6}$$

In the presence of recursion, the *MFPC* solution may be uncomputable because Σ is infinite and the length of σ is unbounded. Practical methods compute *MFPC* by abstracting σ suitably or use properties such as distributivity of flow functions to define a computable version of *MFPC*.

Note that the data flow variable InC in Equation (6) is parameterised by both node and context. We sometimes need a variant $\overline{\mathsf{InC}}$ of InC which is only parameterised by node, for example to match *MFP* (Equation 4) or *MoIVPC* (Equation 5). We do this by defining:

$$\forall n \in \mathbb{N}.\ \overline{\mathsf{InC}}_n = \bigsqcap_{\sigma \in \Sigma} \mathsf{InC}_n(\sigma) \tag{7}$$

2.4.3 Soundness of Context-Sensitive Interprocedural Data Flow Analysis Relative to MoIVPC. In this section we show that the data flow values in the context-sensitive *MFPC* solution (Equation 7) over-approximate the data flow values in *MoIVPC* solution (Equation 5). For this purpose, we need to show that,

$$\forall n \in \mathbb{N}.\ \overline{\mathsf{InC}}_n \quad \sqsubseteq \quad MoIVPC(n) \tag{8}$$

which, from (7) and (5), is equivalent to showing the following

$$\forall n \in \mathbb{N}.\ \bigsqcap_{\sigma \in \Sigma} \mathsf{InC}_n(\sigma) \quad \sqsubseteq \quad \bigsqcap_{(\pi,\sigma)\, \in\, IVPC(n)} f_\pi(BI) \tag{9}$$

which, in turn, can be established by showing that

$$\forall n \in \mathbb{N},\ \forall (\pi, \sigma) \in IVPC(n). \quad \mathsf{InC}_n(\sigma) \ \sqsubseteq\ f_\pi(BI) \tag{10}$$

Claim (10) can be proved by induction on the number of nodes k in π. The base case is when π consists of a single node $n\!:\!Start_{main}$ for which the claim trivially holds. For the inductive hypothesis, assume that it holds for every node m and for every $(\pi', \sigma') \in Paths(m)$ such that path π'

contains fewer than k nodes i.e.

$$\mathsf{InC}_m(\sigma') \quad \sqsubseteq \quad f_{\pi'}(BI)$$

Then, the inductive step requires extending π' by a single edge (m, n) to obtain π. Since we have three kinds of edges, the following three remaining mutually exclusive cases cover all possibilities:

(1) $\mathsf{CE}(m, n)$. Let $\sigma'' = \sigma' \cdot m$.

$$\begin{aligned}\mathsf{InC}_m(\sigma') \sqsubseteq f_{\pi'}(BI) &\Rightarrow f_m(\mathsf{InC}_m(\sigma')) \sqsubseteq f_m(f_{\pi'}(BI)) \\ &\Rightarrow \mathsf{InC}_n(\sigma'') \sqsubseteq f_{\pi'\cdot m}(BI) && \ldots(\text{from } (6),\ \mathsf{InC}_n(\sigma'') \sqsubseteq f_m(\mathsf{InC}_m(\sigma'))) \\ &\Rightarrow \mathsf{InC}_n(\sigma'') \sqsubseteq f_{\pi}(BI) && \ldots(\pi = \pi' \cdot m)\end{aligned}$$

Since every σ'' reaching $n\!:\!Start_Q$ is computed from σ' by appending m, it follows that

$$\forall(\pi, \sigma) \in IVPC(n). \quad \mathsf{InC}_n(\sigma) \quad \sqsubseteq \quad f_\pi(BI)$$

(2) $\mathsf{RE}(m, n)$. Let $\sigma' = \sigma'' \cdot \widehat{n}$.

$$\begin{aligned}\mathsf{InC}_m(\sigma') \sqsubseteq f_{\pi'}(BI) &\Rightarrow f_m(\mathsf{InC}_m(\sigma')) \sqsubseteq f_m(f_{\pi'}(BI)) \\ &\Rightarrow \mathsf{InC}_n(\sigma'') \sqsubseteq f_{\pi'\cdot m}(BI) && \ldots(\text{from } (6),\ \mathsf{InC}_n(\sigma'') \sqsubseteq f_m(\mathsf{InC}_m(\sigma'))) \\ &\Rightarrow \mathsf{InC}_n(\sigma'') \sqsubseteq f_{\pi}(BI) && \ldots(\pi = \pi' \cdot m)\end{aligned}$$

Since every σ'' reaching $n\!:\!Ret_Q$ is computed from σ' by removing $\widehat{n}$, it follows that

$$\forall(\pi, \sigma) \in IVPC(n). \quad \mathsf{InC}_n(\sigma) \quad \sqsubseteq \quad f_\pi(BI)$$

(3) $\mathsf{IE}(n, m)$. In this case, the context does not change.

$$\begin{aligned}\mathsf{InC}_m(\sigma') \sqsubseteq f_{\pi'}(BI) &\Rightarrow f_m(\mathsf{InC}_m(\sigma')) \sqsubseteq f_m(f_{\pi'}(BI)) \\ &\Rightarrow \mathsf{InC}_n(\sigma') \sqsubseteq f_{\pi'\cdot m}(BI) && \ldots(\text{from } (6),\ \mathsf{InC}_n(\sigma') \sqsubseteq f_m(\mathsf{InC}_m(\sigma'))) \\ &\Rightarrow \mathsf{InC}_n(\sigma') \sqsubseteq f_{\pi}(BI) && \ldots(\pi = \pi' \cdot m)\end{aligned}$$

Since every σ' reaching m also reaches n, it follows that

$$\forall(\pi, \sigma) \in IVPC(n). \quad \mathsf{InC}_n(\sigma) \quad \sqsubseteq \quad f_\pi(BI)$$

3 A SURVEY OF EXISTING CONTEXT-SENSITIVE METHODS

In this section we briefly review the known context-sensitive methods with the help of a motivating example. We use the following notation: In most cases, the lattice of data flow values, $\mathbb{L}$ is the set of maps $\mathbb{D} \to \mathbb{V}$ where $\mathbb{D}$ is a set of symbols representing program entities appearing in the program text such as variables or expressions, and $\mathbb{V}$ is the lattice of data flow values corresponding to the symbols. In many cases $\mathbb{V} = \{0, 1\}$; in such cases, $\mathbb{L}$ is a power set lattice $2^{\mathbb{D}}$, but note that some analyses order $\{0, 1\}$ with $0 < 1$ (i.e. order the powerset by $\subseteq$), while other analysis order $\{0, 1\}$ with $1 < 0$ (i.e. order the powerset by $\supseteq$). Some other examples of $\mathbb{V}$ are: For may-points-to analysis, $\mathbb{D}$ is the set of pointers and $\mathbb{V}$ is 2^{Loc} where Loc is the set of all locations. For constant propagation, $\mathbb{D}$ is the set of variables and $\mathbb{V}$ is the lattice $\mathbb{Z}_\perp^\top$, i.e. integers augmented with $\top$ and $\perp$ values. In such cases, $\mathbb{L}$ is not a powerset lattice. In some cases such as typestate analysis [18] $\mathbb{D}$ is a set of typestates, which are non-program symbols.

Consider the four-step control flow path from $Start_{main}$ to $Start_s$ in Figure 2. We use the following variants of notation to describe the path: when the types of nodes related to interprocedural control flow are relevant, we describe the path as $1\!:\!Start_{main} \to 2 \to 3 \to 4\!:\!Call_S \to 13\!:\!Start_S$. When we need the type of only the call nodes, we describe it as $1 \to 2 \to 3 \to 4\!:\!Call_S \to 13$. Sometimes we describe the same path without any node type as $1 \to 2 \to 3 \to 4 \to 13$. Finally, when some

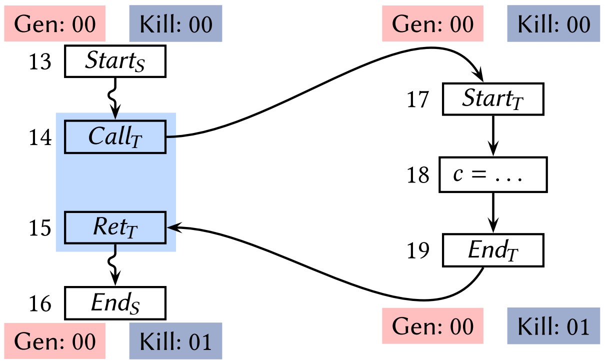


Fig. 3. Computing functional summary for the motivating example in Figure 2

intermediate nodes are not required, we shorten the path by discarding the nodes not required as $1\!:\!Start_{main} \rightsquigarrow 4\!:\!Call_S \longrightarrow 13\!:\!Start_S$.

3.1 A Motivating Example

We use the CFG shown in Figure 2 to illustrate different methods of context-sensitive interprocedural data flow analysis for available expression analysis. Procedure S is invoked from procedure *main* from call sites $4\!:\!Call_S$, $7\!:\!Call_S$, and $10\!:\!Call_S$ with data flow values 11, 11, and 00, respectively (11 indicates that both $a{*}b$ and $c{*}d$ are available; 00 indicates that neither $a{*}b$ nor $c{*}d$ are available). Procedure T is invoked from procedure S at call site $14\!:\!Call_T$. Expression $c{*}d$ is killed in node 18 in procedure T. The figure also shows the final result of the context-sensitive available expressions analysis. For simplicity, the subsequent illustrations for various methods do not show the *main* procedure.

For our example, $\mathbb{D} = \{a{*}b, c{*}d\}$ and $\mathbb{L}$ is a powerset lattice $\mathbb{L} = 2^{\mathbb{D}}$, or alternatively, $\mathbb{L} = \mathbb{D} \longrightarrow \{0, 1\}$. We represent the data flow values by bit vectors in which the first bit represents $a{*}b$ and the second bit represents $c{*}d$. In examples, we also refer to the two expressions as e_1 and e_2, respectively. The flow functions are of the form $f(X) = \text{Gen} \cup (X - \text{Kill})$ where X, Gen, and Kill are subsets of $\mathbb{D}$ represented using bit vectors; Gen and Kill are constant for a given statement.

3.2 Functional Approach

The functional approach forms and inlines the summary of a procedure using a bottom-up traversal over the call graph and there is no need to define contexts. Thus the analysis of a procedure is context-independent and computes the summaries using intensional representations. It is efficient because it analyses every procedure only once even if the procedure has multiple calls. Here we review the functional method proposed by Sharir and Pneuli [29], which requires flow functions to be distributive. Section 5.4 describes a functional method for points-to analysis which has non-distributive flow functions.

Figure 3 shows the procedure summaries computed using a functional method for our example. They are represented using Gen and Kill sets and their construction requires computing these two sets. The construction begins by assuming that the flow function associated with the *Start* nodes is identity; for $f(X) = X$, both Gen and Kill must be $\emptyset$ (or 00 in bit vector notation).

Since procedure T is the leaf node of the call graph, we begin with the identity function represented by Gen = Kill = 00 at $17\!:\!Start_T$. Procedure T does not generate any expression and kills expression $c{*}d$ in node 18. Thus the summary for procedure T is Gen = 00 and Kill = 01, or

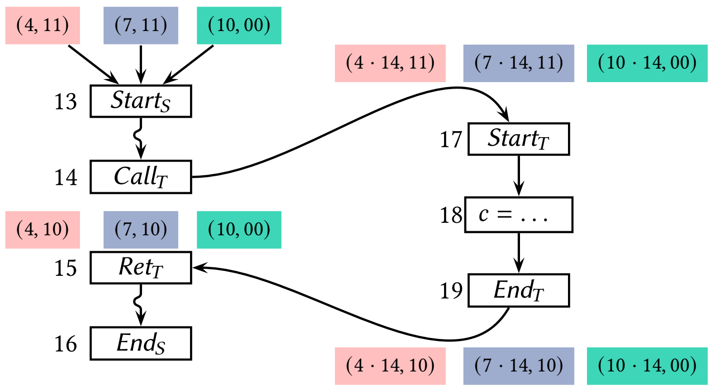


Fig. 4. Explaining classical call-strings method on the motivating example in Figure 2. The three call strings reaching procedure S are 4, 7 and 10 and the corresponding call strings reaching procedure T are $4 \cdot 14$, $7 \cdot 14$ and $10 \cdot 14$.

$\mathbb{S}_T(X) = X - 01$. Then we process procedure S starting with Gen = Kill = 00 at $13\!:\!Start_S$. The summary of procedure T is inlined at to replace the pair of nodes $14\!:\!Call_T$ and $15\!:\!Ret_T$. The summary for procedure S is computed as Gen = 00 and Kill = 01, or $\mathbb{S}_S(X) = X - 01$. It is then used at the three call sites $4\!:\!Call_S$, $7\!:\!Call_S$, and $10\!:\!Call_S$ in procedure *main*.

After computing the summaries in phase 1, the data flow values are computed in phase 2 as follows: For every procedure $Q \neq main$, the data flow values reaching $n\!:\!Start_Q$ are computed from every call to Q. Their meet defines BI_Q which is then used by applying summary for node n to BI_Q to compute the data flow value for node n. Since the data flow values from all callers are merged to define BI_Q, the precision of the method requires the data flow frameworks to be distributive. The tabulation version (see below) of the method additionally requires the lattice $\mathbb{L}$ to be finite.

In our example, the data flow value 11 reaches $13\!:\!Start_S$ from call site $4\!:\!Call_S$. The data flow value reaching $13\!:\!Start_S$ from $7\!:\!Call_S$ is 11 but that from $10\!:\!Call_S$ is 00. This allows us to compute the data flow values within procedure S. Also, the data flow value reaching $5\!:\!Ret_S$ is computed by $\mathbb{S}_S(11) = 10$ (implying that procedure S kills expression $c*d$).

The functional method, as has been defined, does not describe any specific representation for the intensional form of summaries. Thus, the main challenge in a functional approach is to find a concise closed-form representation to model the summaries. This is easy for bit vector frameworks, because the summaries can be represented by constant Gen and Kill sets as illustrated by our example. For other data flow frameworks, the feasibility of this method depends on the feasibility of finding compact representations for the $\sqcap$ and $\circ$ (i.e., meets and compositions) of flow functions. The method however, can also store the summaries in their extensional form (as pairs of input-output values); this version of the method is called the tabulation version.

The other challenge is to construct summaries in the presence of recursion. This can be handled by repeatedly constructing summaries for the procedures involved in recursion until a fixed point (of procedure summaries) is reached. The convergence of this process critically depends on the representation chosen for the analysis and needs to be established explicitly.

3.3 Full Call-Strings-Based Approach

The classical full call-strings-based approach [29] performs a top-down traversal over the call graph and records call strings to perform call-return matching. The analysis of a procedure is context-dependent and records the summaries using an intensional representation. The method defines contexts explicitly in terms of call strings. We refer to full call-strings method as call-strings method, unless otherwise qualified e.g. by k-limited.

The analysis begins with the empty call string ϵ for procedure *main*. When a context σ reaches the call site $n\!:\!Call_Q$, the context reaching the callee Q from call site $n\!:\!Call_Q$ is $\sigma \cdot n$ which is obtained by suffixing n to σ. For our motivating example (Figure 2) call site $4\!:\!Call_S$ invokes procedure S. The context for procedure S is obtained by suffixing 4 to the initial call string ϵ for procedure *main*. This is represented by the call string 4. Thus the three contexts reaching procedure S are 4, 7, and 10 with data flow value 11, 11, and 00, respectively (Figure 4).

- Context 4 reaches procedure S along the path $1\!:\!Start_{main} \rightsquigarrow 4\!:\!Call_S \rightarrow 13\!:\!Start_S$.
- Context 7 reaches S for its second call along the path $1\!:\!Start_{main} \rightsquigarrow 4\!:\!Call_S \rightarrow 13\!:\!Start_S \rightsquigarrow 14\!:\!Call_T \rightarrow 17\!:\!Start_T \rightsquigarrow 19\!:\!End_T \rightarrow 15\!:\!Ret_T \rightsquigarrow 16\!:\!End_S \rightarrow 5\!:\!Ret_S \rightsquigarrow 7\!:\!Call_S \rightarrow 13\!:\!Start_S$.
- Context 10 reaches S for its third call along the path $1\!:\!Start_{main} \rightsquigarrow 4\!:\!Call_S \rightarrow 13\!:\!Start_S \rightsquigarrow 14\!:\!Call_T \rightarrow 17\!:\!Start_T \rightsquigarrow 19\!:\!End_T \rightarrow 15\!:\!Ret_T \rightsquigarrow 16\!:\!End_S \rightarrow 5\!:\!Ret_S \rightsquigarrow 7\!:\!Call_S \rightarrow 13\!:\!Start_S \rightsquigarrow 10\!:\!Call_S \rightarrow 13\!:\!Start_S$.

Procedure T is invoked from procedure S at call site 14 which is suffixed to the contexts reaching procedure S. Thus the contexts reaching procedure T are $4 \cdot 14$, $7 \cdot 14$, and $10 \cdot 14$ with data flow values 11, 11, and 00, respectively. Contexts $4 \cdot 14$ and $7 \cdot 14$ have the same data flow value reaching the start of the procedure. Yet, it is analyzed separately for each context.

At $n\!:\!End_Q$, the last node (say m) in a call string identifies the call site of the call to procedure Q. Hence, when the data flow value associated with a call string $\sigma \cdot m$ reaches a return node $l\!:\!Ret_Q$, if $m = \widehat{l}$, then the data flow value is propagated across $l\!:\!Ret_Q$ with the context σ (obtained by removing m to indicate that the call to Q is over). In our example, for the context $4 \cdot 14$ reaching $19\!:\!End_T$, the last call site is 14 and thus the return site is 15. The context reaching the return point removes 14 from the call sequence to get the context after node $15\!:\!Ret_T$ as 4. Thus, a call string facilitates call-return matching and eliminates interprocedurally invalid paths. For our example, $\mathbb{S}_S$ is $\{4 \mapsto 10, 7 \mapsto 10, 10 \mapsto 00\}$ and $\mathbb{S}_T$ is $\{4 \cdot 14 \mapsto 10, 7 \cdot 14 \mapsto 10, 10 \cdot 14 \mapsto 00\}$. Note that these are extensional representations.

In the absence of recursion Σ is finite and the call string length is bounded by the maximum number of distinct call nodes in any call chain (say K). For recursive programs Σ is infinite. However, for finite $\mathbb{L}$, there exists a known fixed length M such that no new data flow values can be computed for call strings longer than M [29]. Thus it is sufficient to construct call strings only up to length M and the longer call strings can be safely discarded without compromising soundness or precision. For general data flow frameworks, $M = K \cdot (|\mathbb{L}| + 1)^2$. For separable frameworks, $M = K \cdot (|\widehat{\mathbb{L}}| + 1)^2$ where $\widehat{\mathbb{L}}$ is the component lattice (i.e., the lattice associated with the symbols in $\mathbb{D}$). For bit vector frameworks, the value of M further reduces to $3K$.

Note that unlike the round-robin iterative algorithm the call-strings method does not have a means of discovering convergence.[2], Hence, the method as proposed, constructs all call strings up to the length M. Since the length of the call strings is quadratic in the number of data flow values, the number of call strings is rather large, rendering the method impractical.

3.4 Value-Based Termination of Call-Strings Method

The Value-Based Termination of Call-Strings method [10] (VBTCS) proposes to terminate the construction of call strings when the data flow values associated with newer call strings reaching $n\!:\!Start_Q$ are same as the data flow values associated with existing call strings at $n\!:\!Start_Q$.

[2]For a round robin iterative method, $d + 1$ iterations are sufficient to achieve convergence for unidirectional bit vector frameworks, where d is the maximum number of back edges in any acyclic path. Practically, convergence is achieved when there is no change of information in two consecutive iterations and $d + 1$ iterations may not be needed.

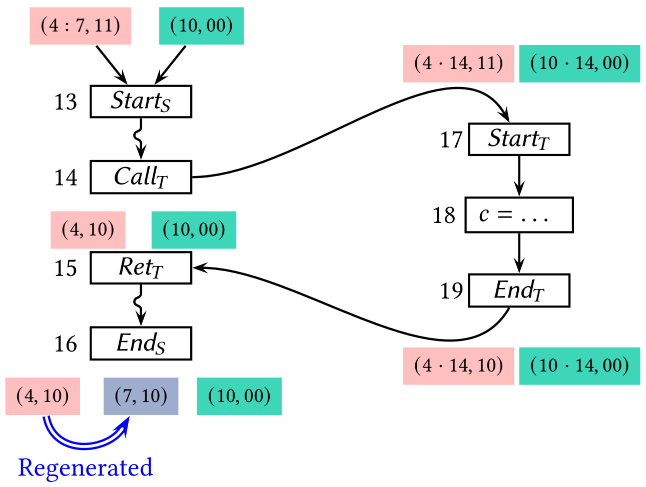


Fig. 5. Explaining value-based termination of the call-strings method on the motivating example in Figure 2

Like the call-strings method, it defines contexts in terms of call strings. However, it partitions the call strings reaching $n : Start_Q$ into equivalence classes on the basis of data flow values and Q need not be reanalyzed for a new call string with the same data flow value reaching Q. It is sufficient to analyze procedure Q for a single call string for each equivalence class because doing so covers all data flow values reaching procedure Q. This reduces the number of call strings significantly. If $\mathbb{L}$ is finite, we get a natural bound on the number of the call strings even in the presence of recursion.

The method uses a *representation function R* which maintains equivalence classes of call strings associated with data flow values. The representation function is implemented as a mutable list of call strings, one for each data flow value $v \in \mathbb{L}$. Given data flow value v, its first associated call string to be encountered appears as the head of the list corresponding to v, and is used as the representative; subsequent call strings associated with v are stored later in the list.

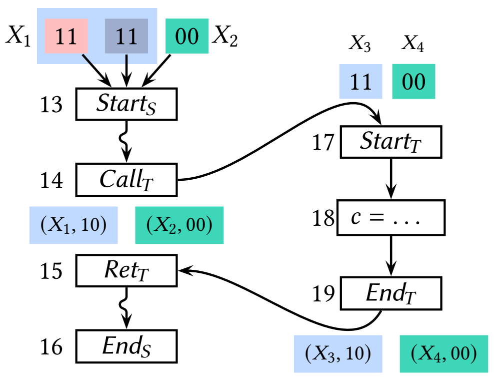
In the classical call-strings approach shown in Figure 4, call strings 4 and 7 reaching procedure S have the same data flow value 11. VBTCS groups these call strings together. It analyses procedure S with context 4 for data flow value 11. Since context 7 has the same data flow value as context 4, it belongs to the same equivalence class as context 4. Thus procedure S is not analyzed for context 7. Instead, the effect of context 4 is regenerated at the end of procedure S for context 7. Available expressions analysis using VBTCS on our motivating example is shown in Figure 5. Since procedure S is not analyzed for context 7, context $7 \cdot 14$ does not reach procedure T.

Similar to the classical call-strings method, the last call site in a call string identifies the call of the procedure and thus identifies the return site as well as the context at the return site. For our example, $\mathbb{S}_S$ is $\{4 \mapsto 10, 7 \mapsto 10, 10 \mapsto 00\}$ and $\mathbb{S}_S$ is $\{4 \cdot 14 \mapsto 10, 10 \cdot 14 \mapsto 00\}$. As before, these are extensional representations.

3.5 VASCO

VASCO [22] is similar to VBTCS in spirit. However, unlike VBTCS, it does not name a partition by a representative call string. Instead, it explicates the use of data flow values and defines contexts in terms of data flow values instead of call strings. We call such contexts as value-contexts.

Consider the motivating example in Figure 2. Let the context in procedure *main* be X_0 where the context represents the boundary information for *main*. Procedure *main* invokes procedure S from three call sites 4, 7, and 10 with data flow values 11, 11, and 00, respectively. VASCO analyses



Context Table

Context	Proc	Entry	Exit
X_0	*main*	00	00
X_1	S	11	10
X_2	S	00	00
X_3	T	11	10
X_4	T	00	00

Context transition diagram

$$\begin{tikzcd}
& X_1 \arrow[r, "14"] & X_3 \\
X_0 \arrow[ur, bend left, "4"] \arrow[ur, "7"'] \arrow[dr, "10"'] & & \\
& X_2 \arrow[r, "14"] & X_4
\end{tikzcd}$$

Fig. 6. Explaining VASCO on the motivating example in Figure 2

a procedure only once for a particular incoming data flow value. Every distinct data flow value reaching a procedure defines a context. Since two distinct data flow values 11 and 00 reach procedure S, the procedure will be analyzed only twice. Thus context X_1 and X_2 are created for data flow values 11, and 00, respectively as shown in Figure 6.

VASCO uses the tabulation-based approach to record the data flow values reaching a procedure in a context table. It also maintains a context transition diagram to record call sequences in order to perform call-return matching; this diagram represents the context-sensitive call graph and its traversal gives the call strings. For our motivating example, the context transition diagram is shown in Figure 6. It records transitions from context X_0 to context X_1 on call sites 4 and 7, and from context X_0 to context X_2 on call site 10. The two data flow values reaching procedure T are 11 and 00. Thus two contexts X_3 and X_4 are created for procedure T and transitions from X_1 to X_3 on 14 and from X_2 to X_4 on 14 are created.

At the end of procedure T, the context transition diagram helps perform call-return matching. It identifies the return site in the caller as well as the corresponding context in the caller. For context X_3, there is an edge from X_1 to X_3 on 14. Thus, the return point is node 15 and the corresponding context at node 15 is X_1 in procedure S. Similarly, for context X_4, the return point is 15 and the corresponding context at node 15 is X_2. Since there are two edges reaching context X_1, the data flow value computed at the end of procedure S for context X_1 is propagated to both the return sites 5 and 8 with the corresponding context at both the sites as X_0.

The summary $\mathbb{S}_S$ can be viewed as an extensional representation $\{X_1 \mapsto 10, X_2 \mapsto 00\}$ and $\mathbb{S}_T$ can be viewed as $\{X_3 \mapsto 10, X_4 \mapsto 00\}$. They are explicated by the context transition table as shown in Figure 6. The call-strings approach analyses both the procedures S and T thrice for distinct call-sequences reaching the procedure, whereas VASCO analyses both the procedures only twice.

Since data flow values are used to define contexts, a bound on the data flow values acts as a natural bound on the number of contexts to be created in case of data flow framework with finite lattices. Thus, there is no need to compute call strings up to a prescribed length, even in the presence of recursion. However, some form of approximation may be required for instances of data flow frameworks with infinite lattices.

3.6 Restricted Contexts

A classical variant of the call-strings method (which was published almost simultaneously) [17] is called the *restricted-contexts* method in [9]. It performs a top-down traversal over the call graph. A curious aspect of this method is that it only stores call strings suffixes of length one and yet does not lose precision. This is because it is restricted to only bit vector frameworks in which

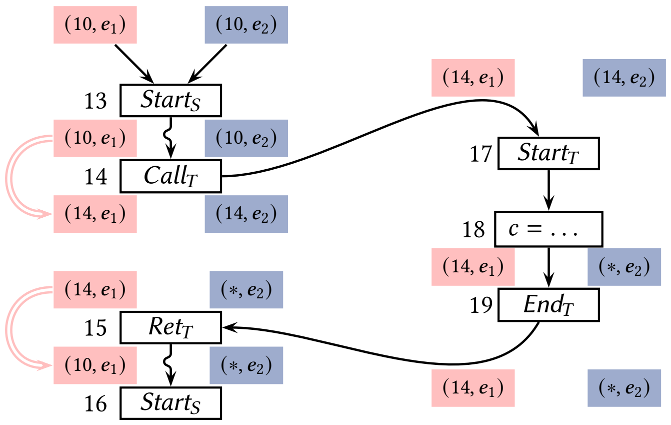


Fig. 7. Explaining the restricted-contexts method on the motivating example in Figure 2. We depict the complement of the available expression analysis problem

- the flow functions are separable (Section 2.2), i.e. the data flow values of the symbols in $\mathbb{D}$ are independent of each other, and
- only two data flow values $\top$ and $\bot$ are possible for each symbol.

This allows storing at every node, *only* the symbols that have $\bot$ value at the node by associating the last call node in the call chain with each symbol $x \in \mathbb{D}$ separately. Thus, the data flow values are relations $\mathbb{C} \cup \{*\} \times \mathbb{D}$ where $\mathbb{C}$ is the set of all call sites and $*$ is a wild card character for call sites. In other words, each pair (m, d) reaching node n records the last call site m from which the $\bot$ value of symbol d reaches n along some path. When the $\bot$ value of d reaches n without crossing any call site (i.e. is generated in a callee procedure or in BI), it is represented by $(*, d)$ at n.

When a pair (m, d) reaches $n\colon Ret_Q$, if $m = \widehat{n}$, then the pair (l, d) that reached $m\colon Call_Q$ is propagated further; this has the effect of retrieving the previous call site in a call chain on reaching a return node. If $m = *$, then the pair $(*, d)$ is propagated unchecked.

Remembering only the last call site has the surprising effect of ensuring full precision as if the full call chain is remembered because of the following reason: Since the flow functions are separable and pairs (m, d) are maintained for each symbol $x \in \mathbb{D}$, there are 'parallel' call strings for each symbol d and each call node in a call string has exactly the *same* ($\bot$) value of the *same* symbol associated with it. As a consequence, there is no need to construct the entire call string. It is sufficient to remember just the last call node because the previous call node in the call strings can be retrieved at $l\colon Ret_Q$ by examining the mapping made at $\widehat{l}\colon Call_Q$.

An application of the restricted-contexts method on our motivating example is shown in Figure 7. Since the method propagates only $\bot$ values, we model available expressions analysis as its complement problem in which only the "unavailable expressions" are remembered. Expressions e_1 and e_2 are both available at $13\colon Start_S$ from the call sites $4\colon Call_S$ and $7\colon Call_S$ in function *main*. However, neither e_1 nor e_2 are available from the call site $10\colon Call_S$ in function *main*. Since we model unavailability of expressions, the data flow value reaching $Start_S$ is $\{(10, e_1), (10, e_2)\}$. At call site 14 in procedure S, data flow value $(14, e_1)$ is created from $(10, e_1)$ and $(14, e_2)$ is created from $(10, e_2)$. Thus, the data flow value reaching $17\colon Start_T$ from call node $14\colon Call_T$ is $\{(14, e_1), (14, e_2)\}$.

If a data flow value is generated at any node in the procedure, a special context $*$ is created. In our example, unavailability of expression e_2 is generated at node 18, which is represented by the pair $(*, e_2)$. Node 18 also blocks propagation of data flow values of the form (m, e_2) for expression e_2 whose operand c is defined in node 18. The information associated with $*$ is propagated to all the callers. Thus the pair $(*, e_2)$ is propagated from $19\colon End_T$ to the return node $15\colon Ret_T$. At $15\colon Ret_T$,

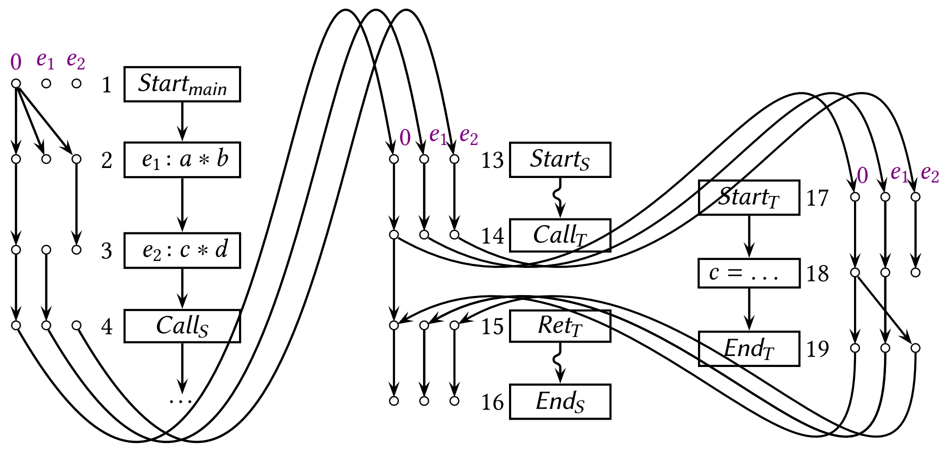


Fig. 8. Exploded CFG used by graph-reachability-based IFDS method for the motivating example in Figure 2. The CFG depicts the complement of the available expression analysis problem. The generation of unavailability of the expression e_2 is shown by the edge from data flow value 0 at node 18 to data flow value e_2 at node 19. The killing of unavailability of the expression e_1 (i.e., generation of the availability of e_1) is depicted by the absence of the edge for data flow value e_1 at node 2 to data flow value e_1 at node 3. The CFG depicts only the relevant part of procedure *main* to highlight the killing of the unavailability of the expression e_1 and e_2 in node 2 and 3, respectively.

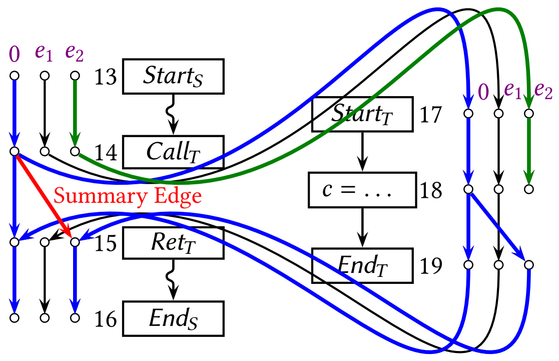
the data flow value at $14{:}\,Call_T$ is consulted and the call sites of the expressions reaching $15{:}\,Ret_T$ are recovered. Thus, the data flow value propagated across $15{:}\,Ret_T$ is $(10, e_1), (*, e_2)$.

3.7 Graph Reachability – IFDS

The IFDS method [24], defined for Interprocedural, Finite, Distributive and Subset-based problems, uses a graph-reachability-based tabulation algorithm. It traverses the call graph top-down. It requires the lattice to be $\mathbb{L} = 2^{\mathbb{D}}$ and the flow functions to be distributive. This allows an efficient representation of distributive functions $f : 2^{\mathbb{D}} \to 2^{\mathbb{D}}$ as their *representation relations* $R_f \subseteq (\mathbb{D} \cup \{\mathbf{0}\}) \times (\mathbb{D} \cup \{\mathbf{0}\})$, see Section 2.2. These relations can be represented by a graph with at most $(|\mathbb{D}| + 1)^2$ edges.

Hence, this method represents the interprocedural control flow of a program by an *exploded CFG* that is obtained by constructing a CFG in which every node is a pair $\langle n, d\rangle$ where $n \in \mathbb{N}$ and $d \in \mathbb{D} \cup \{\mathbf{0}\}$. The exploded CFG has an edge $\langle n, d\rangle \to \langle n', d'\rangle$ if (n, n') is an edge in the CFG and $\langle d, d'\rangle \in R_{f_n}$, i.e. if d and d' are related by the representation relation of the flow function f_n. Thus the effect of symbol $d_1 \in \mathbb{D}$ at program node n_1 can be identified on the effect of symbol $d_2 \in \mathbb{D}$ at program node n_2 by composing the effect of appropriate basis functions along a path in the exploded CFG from node $\langle n_1, d_1\rangle$ to node $\langle n_2, d_2\rangle$. This amounts to finding out if there is a path in the exploded CFG from node $\langle n_1, d_1\rangle$ to node $\langle n_2, d_2\rangle$ and is solved by modelling the problem as a graph reachability problem. A data flow value $d \in \mathbb{D}$ holds at node $n \in \mathbb{N}$ provided there is a path from $\langle m : Start_{main}, 0\rangle$ to $\langle n, d\rangle$.

A natural fallout of this approach is that the analysis to be performed should be amenable to modelling in terms of *some* path reaching a node independently of the other paths reaching the node. Hence available expression analysis cannot be directly mapped to an IFDS problem because IFDS analysis combines values using union operation to capture the effect of a path independently of other paths. Hence, we again consider the complement of available expression analysis that records *unavailability* of expressions which can then be combined using union—if an expression is unavailable along some path reaching a node, then it is unavailable at the node.



Path edges for **0** reaching from 4

$\langle 13, 0\rangle \longrightarrow \langle 13, 0\rangle$
$\langle 13, 0\rangle \longrightarrow \langle 14, 0\rangle$
$\langle 17, 0\rangle \longrightarrow \langle 17, 0\rangle$
$\langle 17, 0\rangle \longrightarrow \langle 18, 0\rangle$
$\langle 17, 0\rangle \longrightarrow \langle 19, 0\rangle$
$\langle 17, 0\rangle \longrightarrow \langle 19, e_2\rangle$
$\langle 13, 0\rangle \longrightarrow \langle 15, 0\rangle$
$\langle 13, 0\rangle \longrightarrow \langle 15, e_2\rangle$
$\langle 13, 0\rangle \longrightarrow \langle 16, 0\rangle$
$\langle 13, 0\rangle \longrightarrow \langle 16, e_2\rangle$

Path edge $\langle 17, 0\rangle \longrightarrow \langle 19, e_2\rangle$ in procedure T becomes Summary edge $\langle 14, 0\rangle \longrightarrow \langle 15, e_2\rangle$ in procedure S.

Path edge $\langle 13, 0\rangle \longrightarrow \langle 16, e_2\rangle$ in procedure S becomes Summary edges $\langle 4, 0\rangle \longrightarrow \langle 5, e_2\rangle$, $\langle 7, 0\rangle \longrightarrow \langle 8, e_2\rangle$ and $\langle 10, 0\rangle \longrightarrow \langle 11, e_2\rangle$ in procedure *main*.

Path edges for $e_2 \in \mathbb{D}$ reaching from 10

$\langle 13, e_2\rangle \longrightarrow \langle 13, e_2\rangle$
$\langle 13, e_2\rangle \longrightarrow \langle 14, e_2\rangle$
$\langle 17, e_2\rangle \longrightarrow \langle 17, e_2\rangle$
$\langle 17, e_2\rangle \longrightarrow \langle 18, e_2\rangle$

Fig. 9. Explaining graph-reachability-based tabulation algorithm for IFDS on the motivating example in Figure 2. Path edges for $e_1 \in \mathbb{D}$ reaching from 10 are not shown in the figure.

For the unavailable expressions analysis, the exploded CFG (Figure 8) would contain the edge $\langle 2, e_1\rangle \longrightarrow \langle 3, e_1\rangle$ if node 2 does not kill the unavailability of the expression e_1 (in other words, does not generate the availability of the expression e_1). Since this is not the case, this edge does not exist in our exploded CFG. Similarly, node 3 kills the unavailability of the expression e_2, which can be seen by the absence of edge $\langle 3, e_2\rangle \longrightarrow \langle 4, e_2\rangle$. Generation of unavailability of the expression e_2 at node 18 is denoted by the edge $\langle 18, 0\rangle \longrightarrow \langle 19, e_2\rangle$.

Figure 9 shows the IFDS algorithm on our example. Path edges represent the paths from start node in a procedure to other nodes within that procedure. A path edge $\langle 17 : Start_T, d\rangle \longrightarrow \langle 18, d'\rangle$ indicates that $d' = f(d)$ where f is the flow function for path from $17 : Start_T$ to 18. The path edges are constructed by traversing the paths in exploded CFG starting from procedure *main* with data flow value **0** and composing appropriate edges. When $14 : Call_T$ is encountered, a path edge reaching call node 14 is extended by including a *summary edge* $\langle 14 : Call_T, 0\rangle \longrightarrow \langle 15 : Ret_T, e_2\rangle$ which represents the path edge from $\langle 17 : Start_T, 0\rangle \longrightarrow \langle 19 : End_T, e_2\rangle$.

The algorithm traverses the exploded CFG to eliminate interprocedurally invalid paths as illustrated in Figure 9. Expressions e_1 and e_2 are not unavailable (i.e., are available) at $13 : Start_S$ from call site 4 in procedure *main*. Since *PathEdges* are paths within a procedure, *PathEdge* $\langle 13, 0\rangle \longrightarrow \langle 13, 0\rangle$ is added by the algorithm along the interprocedural path from 4 to 13 for data flow value 0. Since there exists an edge from 13 to 14 for expression 0, path is extended by recording *PathEdge* $\langle 13, 0\rangle \longrightarrow \langle 14, 0\rangle$. All paths are extended in a similar manner for the entire procedure.

Once *PathEdge* for the entire procedure is computed, summary edges are added at the call sites in the caller to capture the effect of the call. Such summary edges help construct *PathEdge* in the caller procedure. For procedures involved in recursion, summary edges are added repeatedly in the caller until no further *PathEdge* can be computed. Thus it achieves the effect of repeatedly applying the summary of the callee in the caller until a fixed point is achieved.

The call-return matching to exclude interprocedurally invalid paths is handled by the algorithm explicitly by adding a summary edge $\langle 14 : Call_T, d_1\rangle \longrightarrow \langle 15 : Ret_T, d_4\rangle$ in the caller procedure S of procedure T such that all edges described below are present.

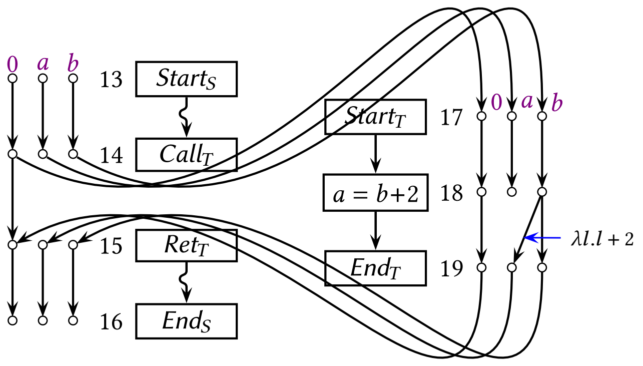


Fig. 10. Example to illustrate IDE algorithm for linear constant propagation problem. The edge functions are all $\lambda l.l$ except where indicated.

- An edge from call node in S to the start node of T: $\langle 14\colon Call_T, d_1\rangle \longrightarrow \langle 17\colon Start_T, d_2\rangle$.
- A *PathEdge* from the start node of T to the end node of T: $\langle 17\colon Start_T, d_2\rangle \longrightarrow \langle 19\colon End_T, d_3\rangle$.
- An edge from the end node of T to the return node in S: $\langle 19\colon End_T, d_3\rangle \longrightarrow \langle 15\colon Ret_T, d_4\rangle$.

For the summary edge $\langle 14\colon Call_T, 0\rangle \longrightarrow \langle 15\colon Ret_T, e_2\rangle$, d_1 is 0, d_2 is 0, d_3 is e_2 and d_4 is e_2.

For unavailable expression analysis, a summary edge indicates that the expression is not available from the start of the callee procedure to the end of the callee procedure. Since the notion of context is not explicit, a lookup key for extensional representation is not required.

3.8 Graph Reachability – IDE

The method of Interprocedural Distributive Environment (IDE) [27] is a generalization of the IFDS method and the functional method both combined into a single method. Like a functional method it computes procedure summaries that are used at call sites. However, it differs from the functional method in that it defines a compact representation for the summaries which is an extension of the representation used for IFDS; the IFDS method takes data flow values to be subsets of $\mathbb{D}$ (i.e. $\mathbb{L} = 2^{\mathbb{D}}$) and hence its flow functions are members of $2^{\mathbb{D}} \to 2^{\mathbb{D}}$ or equivalently $(\mathbb{D} \to \mathbb{V}) \to (\mathbb{D} \to \mathbb{V})$ where $\mathbb{V} = \{0, 1\}$. The IDE method drops the restriction $\mathbb{V} = \{0, 1\}$ and generalizes to allowing $\mathbb{L} = \mathbb{D} \to \mathbb{V}$ where $\mathbb{V}$ can be any complete lattice. Members of the set $\mathbb{L}$ are called *environments* and the IDE method computes *environment transformers* or functions $f : (\mathbb{D} \to \mathbb{V}) \to (\mathbb{D} \to \mathbb{V})$. Given an environment $\mu : \mathbb{D} \to \mathbb{V}$, and $d \in \mathbb{D}$, the function application $f(\mu)(d)$ computes a value in $\mathbb{V}$. Recall that function application here is left-associative. Let μ_i denote the environment obtained by composing the environments along the path $(\pi_i, \sigma_i) \in IVPC(n)$. Then, the method requires that $\forall d \in \mathbb{D}, f\,(\bigsqcap \mu_i)\,(d) = \bigsqcap f(\mu_i)(d)$, i.e., the environment transformers must be distributive.

IFDS stores distributive (flow) functions $f : 2^{\mathbb{D}} \to 2^{\mathbb{D}}$ (or $(\mathbb{D} \to \{0, 1\}) \to (\mathbb{D} \to \{0, 1\})$) as their representation relation $R_f \subseteq (\mathbb{D} \cup \{\mathbf{0}\}) \times (\mathbb{D} \cup \{\mathbf{0}\})$; this relation has at most $(|\mathbb{D}| + 1)^2$ edges. IDE stores environment transformers (flow functions of the form) $(\mathbb{D} \to \mathbb{V}) \to (\mathbb{D} \to \mathbb{V})$ by an extended representation relation $\subseteq (\mathbb{D} \cup \{\mathbf{0}\}) \times (\mathbb{D} \cup \{\mathbf{0}\}) \times (\mathbb{V} \to \mathbb{V})$. Such relations are best thought of graphs of relations like R_f above whose edges (there are at most $(|\mathbb{D}| + 1)^2$ of them) are labelled with functions in $\mathbb{V} \to \mathbb{V}$. These labelling functions are called "micro" functions in [24].

Since we wish to illustrate this method for a data flow framework with an infinite lattice, we take the example of linear constant propagation in Figure 10. In node 18, value of variable a is computed using the value of variable b. This computation is marked on the edge from 18 to 19 by an edge function $\lambda l.l + 2$. Other edges are labelled with the identity function $\lambda l.l$.

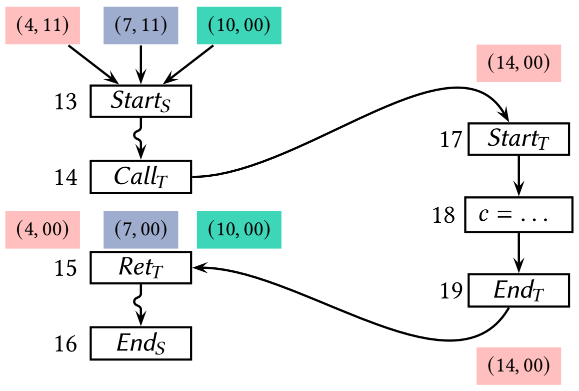


Fig. 11. Explaining k-limited call-strings method for k=1 on the motivating example in Figure 2

Similar to the IFDS method, the IDE method also traverses the edges in the exploded CFG. The *PathEdges* in the IDE method are annotated with path functions and summary edges are annotated with summary functions. Thus, apart from traversing the exploded CFG, the analysis also computes the summary function for the procedure similar to the functional approach [29] but for every symbol $d \in \mathbb{D}$ separately and also by performing a top-down traversal over the call graph.

Unlike IFDS, this method does not require the data flow values to be merged only using union. It can compute the data flow value that must hold along every path, such as in (linear) constant propagation, by performing an additional top-down traversal. This traversal computes values at each node by taking the meet of the data flow values along all paths reaching the node.

This method computes a path edge $\langle 17\!:\!Start_T, \text{b}\rangle \rightarrow \langle 19\!:\!End_T, \text{a}\rangle$ which is annotated with $\lambda l.l+2$ because the value of variable a in procedure T is computed by the expression $b + 2$. The path function for procedure T is incorporated in the caller S as a summary function to compute the value for a. Thus, the value of a is computed using the value of b. Similar to IFDS method, the notion of contexts is not explicit, and hence a key for extensional representation is not required.

Like the functional method, the IDE method also computes the data flow values in the second phase by first computing BI_Q for every procedure $Q \neq main$.

3.9 k-Limited Call Strings

The call-strings method [29] has a popular approximate version in which the length of call strings is restricted to an a-priori fixed bound k such that the analysis is context-insensitive beyond the call-sequences of length k. Thus this is a sound abstraction and appears in many context-sensitive methods such as k-CFA [15, 30] and parameterized object sensitivity [16].

We denote the restriction on call string length by $suffix_k(\alpha{\cdot}n)$ which returns the k-length suffix of $\alpha{\cdot}n$ if $\alpha{\cdot}n$ contains more than k call sites; otherwise it returns $\alpha{\cdot}n$. Figure 11 shows the k-limited call-strings method for $k = 1$ on our motivating example. Procedure S is invoked with the context $4\!:\!Call_S$, $7\!:\!Call_S$ and $10\!:\!Call_S$ from procedure main. Procedure T is invoked from procedure S at call site $14\!:\!Call_T$. Since the length of call sequence to be maintained is 1, the context reaching procedure T is obtained by identifying a suffix of length 1 from the call strings $4 \cdot 14$, $7 \cdot 14$ and $10 \cdot 14$. Thus, these three call strings are approximated to context 14 and this context reaches procedure T with data flow value 00 obtained by taking a meet of the data flow values reaching along the three contexts. At the return node $15\!:\!Ret_T$, the three contexts are reconstructed with the data flow value associated with context 14 at $19\!:\!End_T$.

Method	Notion of context	Definition of context	Inlining strategy		Restrictions on data flow frameworks
			Call graph traversal	Call-return matching	
Functional	None	None	Bottom-up traversal followed by top-down traversal	Inlining procedure summary	Compact representations of meets/compositions of flow functions
Full call strings	Explicit	Sequence of unfinished calls	Top-down traversal	Using context	Finite lattice
VBTCS	Explicit	Sequence of unfinished calls and data flow values	Top-down traversal	Using context	Finite lattice
VASCO	Explicit	Data flow values	Top-down traversal	Using context transition diagram	Finite lattice
IFDS	Implicit	Data flow values	Top-down traversal	Direct matching by the algorithm	Distributive flow functions, finite subset-based lattice
Restricted contexts	Explicit	Call site and data flow values	Top-down traversal	Using context	Separable flow functions over finite lattice
IDE	Implicit	Data flow values	Bottom-up traversal followed by top-down traversal	Inlining procedure summary	Lattice of distributive environments
k-limited call strings	Explicit	Suffixes of sequence of unfinished calls	Top-down traversal	Using context	Lattice with finite height

Table 1. Comparing context-sensitive methods.

Observe that the history of calls from procedure *main* has been discarded to restrict the length of call strings to 1 implying that the data flow value should be propagated back to *every* context reaching procedure S context insensitively. Thus the expression $a{*}b$ which is available at nodes 5 and 8, is now marked to be unavailable by the k-limited call-strings method. In other words, discarding the caller history (by maintaining only call-string suffixes) introduces imprecision by merging the information associated with different call strings.

Note that the notion of k-limiting is different from the length M used in the full call-strings method. The k-limiting method remembers the *suffixes* of length up to k whereas the full call-strings method remembers the *prefixes* of length up to M.

3.10 An Overview of the Features Characterizing Context-Sensitive Methods

Context sensitivity achieves precision by distinguishing between the information for different contexts of a procedure. Since this increases the cost, many context-sensitive methods are motivated primarily by efficiency and employ specialized formalisms for the purpose. In our opinion, this focus on efficiency is the main reason why most methods are presented algorithmically instead of declaratively. In the process, the formal properties that must be satisfied by the analysis (such as insights about soundness or precision) take a back seat and the steps that compute the properties efficiently take prominence. The result is that an absence of coherence between different methods in that the insights gained about a method do not help in understanding the similar formal properties of the other method.

We identify the following set of features that characterize the context-sensitive methods that we have surveyed to bring out the range of variations in the methods. (see Table 1) The table shows that these methods use very different formalisms and appear very dissimilar to a non-expert reader.

- An interprocedural method may define the notion of contexts explicitly [10, 17, 22, 29], may leave it implicit [24, 25, 27], or may not need it at all [29].
- Where the context is explicitly defined, it may be defined in any of the following ways: (a) sequences of unfinished calls [29], (b) sequences of unfinished calls and data flow values [10], (c) data flow values [22], and (d) call sites and data flow values [17].
- The call graph may be traversed bottom-up or top-down [41]. Bottom-up procedure summaries do not depend on the contexts because the information from calling contexts is not available. Hence they use intensional representations. Top-down procedure summaries depend on the contexts because the information from the calling contexts is available and is used for computing the summaries. Hence they use extensional representations.
- The call-return matching may be done using contexts [10, 16, 17, 29] or may use additional abstractions such as context-transition graph [22], or may be directly matched within the algorithm [24, 25, 27], or may be achieved by explicit inlining [29].
- The data flow frameworks supported by the methods have varied requirements. In some cases, $\mathbb{L}$ must be finite [10, 16, 17, 29]; additionally, in some cases, $\mathbb{L}$ should be $(2^{\mathbb{D}}, \supseteq)$ and flow functions must be distributive [24]. Alternatively, $\mathbb{L}$ lattice may be infinite but be definable as $\mathbb{D} \rightarrow \mathbb{V}$ where $\mathbb{V}$ is a complete lattice [25, 27]. Additionally, the meets ($\sqcap$) and compositions ($\circ$) of flow functions must have a compact representation [29].

4 A UNIFIED MODEL OF CONTEXT-SENSITIVE DATA FLOW ANALYSIS

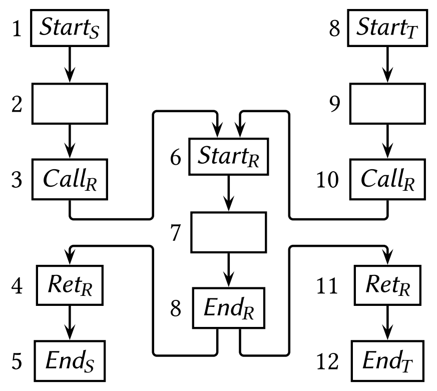
In this section, we generalize the (interprocedural) Maximum Fixed Point solution (*MFPC*, Equation 6 in Section 2.4.2) to define the *Maximum Fixed Point solution using abstract contexts* (*MFPA*). This generalization has two dimensions: the generalization of concrete contexts to abstract contexts, and the generalization of the values computed by *MFPA* from a lattice $\mathbb{L}$ to a set of abstract values $\mathbb{M}$. We formalize these generalizations by defining the notions of an *abstract context structure* denoted $\mathcal{A}$, and an *abstract value structure* denoted $\mathcal{V}$. These generalizations allows us to model many known methods of context-sensitive data flow analysis by defining a pair $(\mathcal{A}, \mathcal{V})$.

4.1 Abstract Context Structure and Abstract Value Structure

4.1.1 Abstract Context Structure. An abstract context structure $\mathcal{A}$ is a generalization of the concrete contexts and is defined as a triple $(\mathbb{A}, \alpha_0, \text{Ncontext})$ where

- $\mathbb{A}$ is a set of *abstract contexts*,
- $\alpha_0 \in \mathbb{A}$ is the initial abstract context corresponding the empty call string ϵ, and
- $\text{Ncontext}_{n \in \mathbb{N}} : \mathbb{A} \rightarrow \mathbb{A}$ is the context transition function associated with *Call* node n.

Ncontext_n is only defined at call nodes $n : Call_Q$ where, given a context α reaching $n : Call_Q$, it determines the corresponding context α' reaching $m : Start_Q$. Context sensitivity requires matching a context reaching a return node $l : Ret_Q$ with the corresponding context reaching the corresponding call node $\widehat{l} : Call_Q$. Our model achieves it by consulting the Ncontext function of $\widehat{l} : Call_Q$.



R	$\mathsf{InA}_6(\epsilon) = \eta_3^{Call}(\mathsf{InA}_3(\epsilon)) \sqcap \eta_{10}^{Call}(\mathsf{InA}_{10}(\epsilon)) = id$ $\mathsf{InA}_8(\epsilon) = \theta_7(\mathsf{In}_6(\epsilon)) = f_7 \circ id = f_7$
S	$\mathsf{InA}_3(\epsilon) = \theta_2(\mathsf{In}_1(\epsilon)) = f_2 \circ id = f_2$ $\mathsf{InA}_4(\epsilon) = \eta_4^{Ret}(\mathsf{InA}_8(\epsilon), \mathsf{InA}_3(\epsilon))$ $= \mathsf{InA}_8(\epsilon) \circ \mathsf{InA}_3(\epsilon) = f_7 \circ f_2$
T	$\mathsf{InA}_{10}(\epsilon) = \theta_9(\mathsf{In}_8(\epsilon)) = f_9 \circ id = f_9$ $\mathsf{InA}_{11}(\epsilon) = \eta_{11}^{Ret}(\mathsf{InA}_8(\epsilon), \mathsf{InA}_{10}(\epsilon))$ $= \mathsf{InA}_8(\epsilon) \circ \mathsf{InA}_{10}(\epsilon) = f_7 \circ f_9$

Fig. 12. Preserving context sensitivity using interprocedural abstract flow functions θ_n, η_n^{Call}, and η_n^{Ret}. The effect of node 2 in procedure S (flow function f_2) reaches node 4 in S but not node 11 in T. Similarly, the effect of node 9 in procedure T (flow function f_9) reaches node 11 in T but not node 4 in S.

The following special instantiations of abstract context structure illustrate its generality.

- The abstract context structure $(\Sigma, \epsilon, \mathsf{Ncontext}_n(\alpha) = \alpha \cdot n)$ recreates $IVPC(n)$ in terms of call strings recording active calls.
- The abstract context structure $(\{\epsilon\}, \epsilon, \mathsf{Ncontext}_n(\epsilon) = \epsilon)$ gives all interprocedural paths, not just the interprocedurally valid ones.
- The abstract context structure $(\Sigma_k, \epsilon, \mathsf{Ncontext}_n(\alpha) = \mathit{suffix}_k(\alpha \cdot n))$ recreates paths under model of the k-limited call strings, assuming that Σ_k is the set of call strings whose length is bounded by k, and suffix_k gives the last k elements of its argument. Note that taking $k = \infty$ or $k = 0$ re-creates the two previous cases.

4.1.2 Abstract Value Structure. An abstract value structure $\mathcal{V}$ is a generalization of the values computed by a data flow analysis and is defined as a tuple $\left(\mathbb{M}, v_0, \theta, \eta_n^{Call}, \eta_n^{Ret}, \mathsf{Project}_Q\right)$ where

- $\mathbb{M}$ is a set of *abstract values*, $v_0 \in \mathbb{M}$ is the initial abstract value that holds at $n\!:\!Start_{main}$,
- $\theta_{n \in \mathbb{N}} : \mathbb{M} \rightarrow \mathbb{M}$ is the intraprocedural abstract flow functions,
- $\eta_n^{Call} : \mathbb{M} \rightarrow \mathbb{M}$ (defined only for $n\!:\!Call_Q$) and $\eta_n^{Ret} : \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{M}$ (defined only for $n\!:\!Ret_Q$) are interprocedural abstract flow functions, and
- $\mathsf{Project}_{Q \in \mathbb{P}\mathsf{roc}} : \mathbb{M} \rightarrow \mathbb{L}$ is the projection function that extracts values in $\mathbb{L}$ from those in $\mathbb{M}$ for nodes associated with procedure Q.

Intuitively, we expect $\mathcal{A}$ and $\mathcal{V}$, taken together, to be rich enough to support precise call-return matching. Two exemplifying instantiations of abstract value structures are as follows, where id is the shorthand for the identity function $\lambda x.x$.

- The call-strings method uses a simple abstract value structure

$$\left(\mathbb{M} = \mathbb{L},\ v_0 = BI,\ \theta_n(v) = f_n(v),\ \eta_n^{Call}(v) = v,\ \eta_n^{Ret}(v, w) = v,\ \mathsf{Project}_Q(v) = v\right)$$

  Here $\mathbb{M}$ is $\mathbb{L}$, η_n^{Ret} ignores its second argument (data flow value at the entry of a call node) and returns the first argument (data flow value at the exit of the end block of the callee) and all other functions are identity functions. Yet, the method is precise because its abstract context structure $(\Sigma, \epsilon, \mathsf{Ncontext}_n(\alpha) = \alpha \cdot n)$ is rich enough to contain all contexts and the transition function is injective so we can distinguish between all contexts reaching a procedure.

- The functional method uses the abstract value structure
$$\left(\mathbb{M} = \mathbb{L} \to \mathbb{L}, v_0 = id, \theta_n(v) = f_n \circ v, \eta_n^{Call}(v) = id, \eta_n^{Ret}(v, w) = v \circ w, \mathsf{Project}_Q(v) = v(BI_Q)\right)$$
This is richer than the abstract value structure used by the call-strings method because $\mathbb{M}$ is $\mathbb{L} \to \mathbb{L}$ in this case. Thus a value $v \in \mathbb{M}$ represents a summary function instead of a data flow value in $\mathbb{L}$. As a consequence, the intraprocedural abstract flow function θ composes its argument with the default flow function f_n of node n. Besides, η_n^{Ret} allows composition of its arguments. As we shall see later, the second argument w is the summary function associated with a call node and the first argument v is the summary function for the procedure called at the call node. Effectively, η_n^{Ret} facilitates call-return matching. Given this rich abstract value structure, the functional method uses the trivial abstract context structure $(\{\epsilon\}, \epsilon, \mathsf{Ncontext}_n(\epsilon) = \epsilon)$. Intuitively, the summaries constructed by a functional method are independent of any context because they can be used for any call to a procedure; they are functions parameterised on the information reaching the call. Unlike context-insensitive summaries that merge all contexts, the summaries constructed by a functional method compute values separately for different information reaching the calls to the procedures. Figure 12 illustrates how the method ensures call-return matching.

4.2 Interprocedural *MFP* Solution in the Unified Model

Let $(\mathbb{A}, \alpha_0, \mathsf{Ncontext}_n)$ be an abstract context structure. An *abstract interprocedurally valid path* reaching node n, denoted $IVPA(n)$ is (π, α) where π is a sequence of nodes and α is an abstract context and is defined as follows.

$$\frac{}{(n,\ \alpha_0) \in IVPA(n)} \text{ if } n\!:\!Start_{main} \qquad \frac{(\pi\cdot m,\ \alpha) \in IVPA(m)}{(\pi\cdot m\cdot n,\ \mathsf{Ncontext}_m(\alpha)) \in IVPA(n)} \text{ if } \mathsf{CE}(m,n)$$
$$\frac{(\pi\cdot m,\ \alpha) \in IVPA(m)}{(\pi\cdot m\cdot n,\ \alpha) \in IVPA(n)} \text{ if } \mathsf{IE}(m,n) \qquad \frac{(\pi\cdot m,\ \mathsf{Ncontext}_{\widehat{n}}(\alpha)) \in IVPA(m)}{(\pi\cdot m\cdot n,\ \alpha) \in IVPA(n)} \text{ if } \mathsf{RE}(m,n) \tag{11}$$

For convenience, we define predicate $ReachA_n(\alpha)$ to assert that abstract context α reaches node n through a series of valid transitions starting from α_0:

$$ReachA_n(\alpha) \Leftrightarrow \exists \pi \text{ such that } (\pi \cdot n, \alpha) \in IVPA(n)$$

With this provision, the data flow equations for *MFPA* are obtained from those for *MFPC* (Equation 6) by using the abstract context structure to replace the call strings $\sigma \in \Sigma$ by abstract contexts $\alpha \in \mathbb{A}$ and by replacing the data flow values in $\mathbb{L}$ by abstract values in $\mathbb{M}$.

$$\mathsf{InA}_n(\alpha) = \begin{cases} v_0 & n\!:\!Start_{main} \wedge ReachA_n(\alpha) \\ \displaystyle\bigsqcap_{\substack{m,\alpha' \text{ such that } m\,:\,Call_Q \\ \alpha = \mathsf{Ncontext}_m(\alpha')}} \eta_m^{Call}(\mathsf{InA}_m(\alpha')) & n\!:\!Start_Q \wedge Q \neq main \wedge ReachA_n(\alpha) \\ \eta_n^{Ret}(\mathsf{InA}_{End_Q}(\mathsf{Ncontext}_{\widehat{n}}(\alpha)),\ \mathsf{InA}_{\widehat{n}}(\alpha)) & n\!:\!Ret_Q \wedge ReachA_n(\alpha) \\ \displaystyle\bigsqcap_{p \in pred(n)} \theta_p(\mathsf{InA}_p(\alpha)) & \mathsf{IntraNode}(n) \wedge ReachA_n(\alpha) \\ \top & \neg ReachA_n(\alpha) \end{cases} \tag{12}$$

Observe that the following combination recreates the data flow equations for InC_n (Equation 6) as a special case of InA_n (Equation 12).

$$\mathcal{A} = (\Sigma, \epsilon, \mathsf{Ncontext}_n(\alpha) = \alpha \cdot n)$$
$$\mathcal{V} = \left(\mathbb{M} = \mathbb{L},\ v_0 = BI,\ \theta_n(v) = f_n(v),\ \eta_n^{Call}(v) = v,\ \eta_n^{Ret}(v, w) = v,\ \mathsf{Project}_Q(v) = v\right)$$

For contrast, the following combination models the functional method. Figure 12 illustrates call-return matching performed by the interprocedural abstract flow functions θ_n, η_n^{Call}, and η_n^{Ret}.

$$\mathcal{A} = \left(\{\epsilon\}, \epsilon, \mathsf{Ncontext}_n(\epsilon) = \epsilon\right)$$
$$\mathcal{V} = \left(\mathbb{M} = \mathbb{L} \to \mathbb{L}, v_0 = id, \theta_n(v) = f_n \circ v, \eta_n^{Call}(v) = id, \eta_n^{Ret}(v, w) = v \circ w, \mathsf{Project}_Q(v) = v(BI_Q)\right)$$

The variant $\overline{\mathsf{InA}}$ of InA which is only parameterised by node, is defined as

$$\forall n \in \mathbb{N}.\ \ \overline{\mathsf{InA}}_n = \bigsqcap_{\alpha \in \mathbb{A}} \mathsf{InA}_n(\alpha) \tag{13}$$

Note that both InA_n and $\overline{\mathsf{InA}}_n$ are values in $\mathbb{M}$. The final result of data flow analysis are values $\mathsf{In}_n \in \mathbb{L}$ and are retrieved as follows:

$$\forall Q \in \mathbb{P}\mathsf{roc}, \forall n \in \mathbb{N}_Q.\ \ \mathsf{In}_n = \mathsf{Project}_Q(\overline{\mathsf{InA}}_n) \tag{14}$$

A method instantiated in our model is sound if $\forall n \in \mathbb{N}, \mathsf{In}_n \sqsubseteq \mathsf{InC}_n$ where InC_n is defined using Equation (6); the method is precise if $\forall n \in \mathbb{N}, \mathsf{In}_n = \mathsf{InC}_n$.

4.3 Examples of Instantiating Various Context-Sensitive Methods in the Unified Model

Table 2 instantiates many known interprocedural methods in our unified model. We first explain the methods one by one and then summarize our observations. For each method, the abstract context structure is explained first followed by its abstract value structure.

- For the call-strings method, $\mathsf{Ncontext}_n$ merely appends the call site to the context reaching $n\!:\!Call_Q$. In the absence of recursion Σ is finite and the call-string length is bounded by the maximum length of call chain. For recursive programs, Σ is infinite. If $\mathbb{L}$ is finite, then there is no need to construct call strings longer than M (see Section 3.3). We model it by

  $$\forall n \in \mathbb{N}, \forall \alpha \in \mathbb{A},\ \ length(\alpha) > M \Rightarrow ReachA_n(\alpha) = false$$

  Hence, the call strings longer than M can be ignored because it from Equation (12),

  $$\forall n \in \mathbb{N}, \forall \alpha,\ \ length(\alpha) > M \Rightarrow \mathsf{InA}(\alpha) = \top$$

  Since $\mathcal{A}$ of the call-strings method is rich enough to distinguish between all contexts, its $\mathcal{V}$ is simple: the abstract value set $\mathbb{M} = \mathbb{L}$ and all functions are identity functions.
- The VBTCS method partitions the call strings reaching $n\!:\!Start_Q$ on the basis of the data flow values associated with them (see Section 3.4). Its abstract context structure is same as that of the call-strings method except that the $\mathsf{Ncontext}_n$ function uses the representation function R. Its abstract value structure is same as that of the call-strings method.
- VASCO defines contexts in terms of values in $\mathbb{L}$ (see Section 3.5). Its $\mathsf{Ncontext}_n$ function chooses the data flow value $\mathsf{InA}_n(\alpha)$ as the context reaching $m\!:\!Start_Q$. Its abstract value structure is identical to that of the call-strings method and VBTCS with identical restrictions on the data flow frameworks supported—all of them require the lattice $\mathbb{L}$ to be finite (which translates to the lattice $\mathbb{V}$ being finite); they differ only in their abstract context structures.

Method	Abstract Value Structure $\mathcal{V}$						Abstract Context Structure $\mathcal{A}$			Restrictions	Precise?
	$\mathbb{M}$	v_0	$\theta_n(v)$	$\eta_n^{Call}(v)$	$\eta_n^{Ret}(v, w)$	$\text{Project}_Q(v)$	$\mathbb{A}$	α_0	$\text{Ncontext}_n(\alpha)$		
Full call strings	$\mathbb{L}$	BI	$f_n(v)$	v	v	v	Σ	ϵ	$\alpha \cdot n$	Finite $\mathbb{V}$	Yes
VBTCS	$\mathbb{L}$	BI	$f_n(v)$	v	v	v	Σ	ϵ	$R(\alpha \cdot n, \text{InA}_n(\alpha))$	Finite $\mathbb{V}$	Yes
VASCO	$\mathbb{L}$	BI	$f_n(v)$	v	v	v	$\mathbb{L}$	BI	$\text{InA}_n(\alpha)$	Finite $\mathbb{V}$	Yes
Restricted contexts	$\mathbb{M} = 2^{\mathbb{C}}$	(as defined in Section 4.3)					$\{\epsilon\}$	ϵ	ϵ	$\mathbb{V} = (\{0, 1\}, \sqsubseteq)$, Separable f_n	Yes
IFDS method	$\mathbb{L}$	BI	$f_n(v)$	v	v	v	$\mathbb{L}$	BI	$\text{InA}_n(\alpha)$	$\mathbb{V} = (\{0, 1\}, \sqsubseteq)$, Distributive f_n	Yes
IDE method	$\mathbb{L} \to \mathbb{L}$	id	$f_n \circ v$	id	$v \circ w$	$v\,(BI_Q)$	$\{\epsilon\}$	ϵ	ϵ	Distributive f_n	Yes
Functional method	$\mathbb{L} \to \mathbb{L}$	id	$f_n \circ v$	id	$v \circ w$	$v\,(BI_Q)$	$\{\epsilon\}$	ϵ	ϵ	Compact representations of $\sqcap$ and $\circ$ of f_n	Yes
k-limited call strings	$\mathbb{L}$	BI	$f_n(v)$	v	v	v	Σ_k	ϵ	$suffix_k(\alpha \cdot n)$	$\mathbb{V}$ with finite height	No
Context-insensitive	$\mathbb{L}$	BI	$f_n(v)$	v	v	v	$\{\epsilon\}$	ϵ	ϵ	$\mathbb{V}$ with finite height	No

- The set of symbols, $\mathbb{D}$, is finite; the lattice of values $\mathbb{L} = \mathbb{D} \to \mathbb{V}$; $BI \in \mathbb{L}$ is the boundary value; function $id : \mathbb{L} \to \mathbb{L}$ is a shorthand for $\lambda x.x$.
- Other examples of $\mathbb{V}$ are $\{\top, \bot\} \cup \mathbb{Z}$ for constant propagation and $(2^{Loc}, \supseteq)$ for points-to analysis where Loc is the set of all locations.
- $\mathbb{C}$ is the set of call sites $\{n \mid n : Call_Q\}$ ($\epsilon \notin \mathbb{C}$), Σ is the set of call strings ($\mathbb{C}^*$), Σ_k is k-limited call strings (suffixes of length k), and R is the representation function defined in Section 3.4.

Table 2. Instantiating different methods to the unified model.

- The restricted-contexts method is applicable only to the bit vector frameworks and computes a set of pairs (m, d) at node n where $m \in \mathbb{C}$ is the last call site from where the $\perp$ value of $x \in \mathbb{D}$ has reached node n. We model the method using a simple $\mathcal{A}$ but a rich $\mathcal{V}$. Although the call site m in a pair (m, d) for $x \in \mathbb{D}$ plays the role of context, we do not model it through $\mathcal{A}$ because of the presence of a wild card $*$ which propagates across a return node unmodified but not across a call node $m\colon Call_Q$ where the $*$ must transition to m. This asymmetry is different from any notion of context found in any other method and hence cannot be modelled without complicating the model for all methods.
  We use the property of separability of flow functions and model the method as a 'parallel' analysis over all symbols in $\mathbb{D}$. Let $\mathbb{D} = \{d_1, d_2, \ldots, d_{|\mathbb{D}|}\}$. Then,
  $$\mathcal{V} = \left(\mathcal{V}_{d_1} \times \cdots \times \mathcal{V}_{d_{|\mathbb{D}|}}\right)$$
  We use the trivial abstract context $\mathcal{A} = (\{\epsilon\}, \epsilon, \mathsf{Ncontext}_n(\epsilon) = \epsilon)$ for all $\mathcal{V}_{d_i}$, where $1 \leq i \leq |\mathbb{D}|$. Thus the context ϵ flows to all nodes. The abstract value structure $\mathcal{V}_{d_i}$ is defined by viewing data flow values as a set of call sites to identify the callers to which the data flow value of d_i can be propagated to (because it is $\perp$). The universal set represents $\mathbb{C}$ implying that there is no restriction on the callers to which the data flow value of d_i can be propagated to (thus $\mathbb{C}$ models $*$; the online appendix shows an instantiation of our model using an explicit $*$). The empty set indicates that the propagation of the data flow value of d_i should be stopped (because it is $\top$) .
  - The set of abstract values is $\mathbb{M} = 2^{\mathbb{C}}$ and the initial value $v_0 \in \mathbb{M}$ is defined as follows:
  $$v_0 = \begin{cases} \mathbb{C} & \text{if } d_i \text{ has } \perp \text{ value in } BI \\ \emptyset & \text{otherwise} \end{cases}$$
  - The intraprocedural abstract flow function is defined as follows:
  $$\theta_n(v) = \begin{cases} \emptyset & f_n|_{d_i} = \lambda b.\top \\ \mathbb{C} & f_n|_{d_i} = \lambda b.\perp \\ v & \text{otherwise} \end{cases}$$
  - The interprocedural abstract flow functions are defined as follows:
  $$\eta_n^{Call}(v) = \begin{cases} \{n\} & v \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$
  $$\eta_n^{Ret}(v, w) = \begin{cases} v & v = \mathbb{C} \\ w & v \neq \mathbb{C} \wedge \widehat{n} \in v \\ \emptyset & \text{otherwise} \end{cases}$$
  - The projection function is defined as follows:
  $$\mathsf{Project}_Q(v) = \begin{cases} \{d_i\} & v \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$
  - The above $\mathsf{Project}_Q$ function was actually one of a family, one $\mathsf{Project}_Q^i$ per symbol x_i. We obtain the overall $\mathsf{Project}_Q$ function as the union over i of all the $\mathsf{Project}_Q^i$.
- In our model, $\mathcal{A}$ of IFDS is identical to that of VASCO and the methods differ slightly on their abstract value structure. Note that unlike VASCO, IFDS requires the flow functions to be distributive (see Section 3.7). This allows an efficient representation in terms of an exploded

CFG that stores a collection of $(\mathbb{D} \cup \{\mathbf{0}\}) \times (\mathbb{D} \cup \{\mathbf{0}\})$ relations. Our model is oblivious to this level of detail because it is a matter of efficiency than a matter of defining what is computed. The use of relations by IFDS allows procedures to internally define contexts in terms of individual elements of $\mathbb{D}$ rather than in terms of subsets of $\mathbb{D}$ as is done by VASCO. For example, for $\mathbb{D} = \{a, b, c\}$, $\mathbb{A} = \big\{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\big\}$ for VASCO. Thus VASCO could analyze a procedure for each of these contexts separately and thus as many as eight times. Theoretically, IFDS must consider the same $\mathbb{A}$. However, given its assumption of distributive functions, IFDS implements the contexts in the same α as a combination of the following: 0, a, b, c. Thus a procedure would be analyzed at most four times by IFDS. Thus IFDS is more efficient than VASCO but is less general because it requires distributive flow functions.

- The functional method also uses the simplest abstract context structure similar to that of a context-insensitive method. However, it achieves precision by using a more advanced abstract value structure that ensures precise call-return matching using the interprocedural abstract flow functions η_n^{Call} and η_n^{Ret} as illustrated in Figure 12.
  The values $\overline{\mathsf{InA}}_n \in \mathbb{M}$ computed in phase 1 are summary functions $\mathbb{L} \to \mathbb{L}$ using which the values $\mathsf{In}_n \in \mathbb{L}$ are computed in phase 2. By definition, $\overline{\mathsf{InA}}_n$ computes In_n from the data flow values reaching $n\!:\!Start_Q$ where $n \in \mathbb{N}_Q$. Hence, the second phase computes the following:
  - For procedure $main$, $\mathsf{In}_n = \mathsf{Project}_{main}(\overline{\mathsf{InA}}_n) = \overline{\mathsf{InA}}_n(BI)$.
  - For every procedure $Q \neq main$, BI_Q is computed as the meet of the data flow values In_m for all $m\!:\!Call_Q$. Then. $\mathsf{In}_n = \mathsf{Project}(\overline{\mathsf{InA}}_n) = \overline{\mathsf{InA}}_n(BI_Q)$.

  Since the data flow values from all callers are merged to define BI_Q, the method ensures precision by requiring the data flow frameworks to be distributive. The tabulation version of the method additionally requires the lattice $\mathbb{L}$ to be finite.
- The IDE method combines the features of the IFDS and functional methods. Like the functional method it computes procedure summaries that are used at call sites. However, it differs from the functional method in that it defines a representation for the summaries (see Section 3.8). This representation is an extension of the representation used in IFDS. However, since the underlying representation is a matter of detail not captured by our model, the abstract context and value structures IDE are similar to those of the functional method.
- The k-limited call-strings method considers only k-length suffixes of call strings. Thus, the $\mathsf{Ncontext}_n$ function is non-injective in that the call strings that differ in their prefixes at $n\!:\!Call_Q$ may be mapped to the same call string reaching $m\!:\!Start_Q$. For example, for 2-limited call strings, $\mathsf{Ncontext}_n(m' \cdot m) = \mathsf{Ncontext}_n(m'' \cdot m) = m \cdot n$. This leads to merging data flow values of these call strings leading to imprecision—the merged value reaching $l\!:\!Ret_Q$ is propagated back to multiple call strings at $\widehat{l}\!:\!Call_Q$. The $\mathcal{V}$ of this method is same as that of the call-strings method. The only restriction that this method places on the data flow framework is that $\mathbb{L}$ should have finite height for termination of the analysis. Since $\mathbb{L} = \mathbb{D} \to \mathbb{V}$ and $\mathbb{D}$ is finite, this requirement translates to the finiteness of the height of $\mathbb{V}$.
- For contrast, we have also modelled a context-insensitive method using our model. It uses the simplest possible abstract context structure consisting of the lone 0-length call string ϵ. In other words, no callers are remembered, effectively converting calls into simple goto statements but returns are converted to non-deterministic goto to one of several return sites. This admits interprocedurally invalid paths thereby causing imprecision. Its abstract value structure is similar to the methods that use a variant of call strings. The only restriction that this method places on the data flow framework is that the lattice $\mathbb{L}$ should have finite height

for termination so that a meet of all descending chains can be computed. Since $\mathbb{L} = \mathbb{D} \to \mathbb{V}$ and $\mathbb{D}$ is finite, this requires the finiteness of the height of $\mathbb{V}$.

4.4 Soundness and Precision of *MFPA* Relative to *MFPC*

We define soundness and precision of *MFPA* relative to *MFPC* and identify

- A soundness criterion that ensures that the *MFPA* solution computed by a method instantiated in the unified model is an over-approximation of the *MFPC* solution.
- A precision criterion that ensures that the *MFPA* solution computed by a method instantiated in the unified model is same as the *MFPC* solution.

4.4.1 Validity of an Abstract Context Structure and Abstract Value Structure. An abstract context structure $\mathcal{A} = (\mathbb{A}, \alpha_0, \mathsf{Ncontext}_n)$ is *valid* if a mapping $H : \mathbb{N} \to \Sigma \to \mathbb{A}$ exists between the (concrete) context structure $(\Sigma, \epsilon, \mathsf{Ncontext}_n(\sigma) = \sigma \cdot n)$ and $\mathcal{A}$. A valid $\mathcal{A}$ guarantees the existence of an abstract context $\alpha \in \mathbb{A}$ for every concrete context $\sigma \in \Sigma$. Thus it ensures that the abstract data flow equations (Equation (14)) cover all interprocedurally valid paths (defined by (2)).

Observe that all methods in Table 2 have a valid abstract context structure because a mapping H exists such that every $\sigma \in \Sigma$ can be mapped to some $\alpha \in \mathbb{A}$.

In order to define the validity of $\mathcal{V}$, we first define that $\theta : \mathbb{M} \to \mathbb{M}$ *simulates* $f : \mathbb{L} \to \mathbb{L}$ if $f \circ \mathsf{Project}_Q = \mathsf{Project}_Q \circ \theta$. Here θ and f are intended as (respectively) abstract and concrete flow functions for nodes within procedure Q. This is straightforward for non-call nodes, but we want to extend it for call nodes too – those, say, to procedure R. Let $f_R : \mathbb{L} \to \mathbb{L}$ denote the overall flow function for procedure R. We are not interested in obtaining a representation of f_R but want to define f_R mathematically and use the value of $f_R(x)$ for any arbitrary $x \in \mathbb{L}$ to establish a useful property of abstract value structures. Hence we define $f_R(x) = \overline{\mathsf{InC}}_{End_R}$ where $\overline{\mathsf{InC}}_n$ is defined in Equation (7) using $\mathsf{InC}(\sigma)$ defined as the *MFPC* solution of Equation (6) with the following changes:

- $Start_{main}$ in the first case is replaced by $Start_R$.
- *BI* in the first case is replaced by x.
- If R is recursive, the we introduce a new function R_{REC} which clones R to represent the recursive calls to R; the non-recursive calls remain calls to R.

Since the flow function f_n associated with a call node is the identity function in the definition of InC_n (Equation (6)), we get the following identity:

$$\forall v \in \mathbb{M}, Q \in \mathbb{P}\mathrm{roc}, n{:}Call_R \in \mathbb{N}_Q. \quad \mathsf{Project}_R(\eta_n^{Call}(v)) = \mathsf{Project}_Q(v) \tag{15}$$

This follows from the way we have defined $f_R(x)$ above in which x replaced the *BI* value for the callee procedure.

Again suppose R is a procedure called from Q. Having extended the idea of concrete flow functions f_n from individual nodes to function bodies f_R, we similarly extend the abstract flow functions θ_n to function bodies θ_R. Then, an abstract value structure is valid in our model if the abstract flow functions simulate the (concrete) flow functions. This is captured by the following conditions:

$$\forall v \in \mathbb{M}, Q \in \mathbb{P}\mathrm{roc}, n \in \mathbb{N}_Q. \qquad \mathsf{Project}_Q(\theta_n(v)) = f_n(\mathsf{Project}_Q(v)) \tag{16}$$

$$\forall v \in \mathbb{M}, Q \in \mathbb{P}\mathrm{roc}, n{:}Ret_R \in \mathbb{N}_Q. \quad \mathsf{Project}_Q\big(\eta_n^{Ret}\big(\theta_R(\eta_{\hat{n}}^{Call}(v)), v\big)\big) = f_R\big(\mathsf{Project}_Q(v)\big) \tag{17}$$

Figure 13 illustrates these conditions. When they are satisfied, we can be sure that the abstract flow functions model the flow functions faithfully and the abstract value structure $\mathcal{V}$ defined for the method is valid for our model. It is easy to verify that all methods instantiated in Table 2 have valid abstract context and value structures.

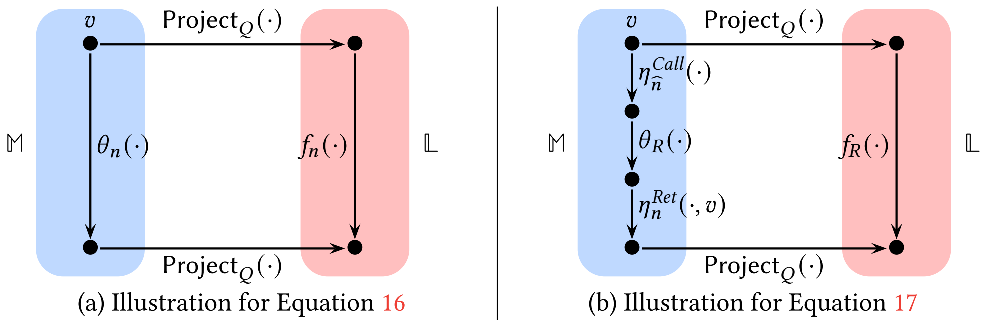

v
$\text{Project}_Q(\cdot)$
$\mathbb{M}$
$\theta_n(\cdot)$
$f_n(\cdot)$
$\mathbb{L}$
$\text{Project}_Q(\cdot)$
(a) Illustration for Equation 16
v
$\text{Project}_Q(\cdot)$
$\eta_{\widehat{n}}^{Call}(\cdot)$
$\mathbb{M}$
$\theta_R(\cdot)$
$f_R(\cdot)$
$\mathbb{L}$
$\eta_n^{Ret}(\cdot, v)$
$\text{Project}_Q(\cdot)$
(b) Illustration for Equation 17

Fig. 13. For the abstract value structure $\mathcal{V}$ to be valid, abstract flow functions must simulate the (concrete) flow functions for all $v \in \mathbb{M}$

4.4.2 *Soundness Criteria.* Intuitively, the soundness of a method is guaranteed if it ensures that no interprocedurally valid path is missed and the method employs a valid abstract value structure.

Our top-level requirement for soundness of *MFPA* is

$$\forall n \in \mathbb{N}.\ \ \text{In}_n \sqsubseteq \overline{\text{InC}_n} \tag{18}$$

From Equations (7), (13), and (14), this follows if

$$\forall Q \in \mathbb{P}\text{roc}, \forall n \in \mathbb{N}_Q, \forall \sigma \in \Sigma, \exists\, \alpha \in \mathbb{A}.\ \ \text{Project}_Q(\text{InA}_n(\alpha)) \sqsubseteq \text{InC}_n(\sigma) \tag{19}$$

The existence of an α corresponding to a σ at every node is guaranteed by a valid abstract context structure. Therefore we can do the reasoning over paths. However, since our focus is on interprocedurally valid paths covered by α, our reasoning is simplified considerably by considering the paths in which finished calls have been abstracted out. This is achieved for a path π containing $\widehat{n}\!:\!Call_Q$ and $n\!:\!Ret_Q$ by making $n\!:\!Ret_Q$ immediately follow the closest $\widehat{n}\!:\!Call_Q$ in the path by adding a *summary* edge between them to represent the effect of the call that has been abstracted out; we denote such paths by $\overline{\pi}$ and adapt the rules in definition (11) to define them as follows:

$$\begin{gathered}
\frac{}{(n,\ \alpha_0) \in \overline{IVPA}(n)}\ n\!:\!Start_{main} \\
\frac{(\overline{\pi}\cdot m,\ \alpha) \in \overline{IVPA}(m)}{(\overline{\pi}\cdot m\cdot n,\ \alpha) \in \overline{IVPA}(n)}\ \text{IE}(m,n) \\
\frac{(\overline{\pi}\cdot m,\ \alpha) \in \overline{IVPA}(m)}{(\overline{\pi}\cdot m\cdot n,\ \text{Ncontext}_m(\alpha)) \in \overline{IVPA}(n)}\ \text{CE}(m,n) \\
\frac{(\overline{\pi}\cdot \widehat{n}\cdot l\cdot \overline{\pi}'\cdot m,\ \text{Ncontext}_{\widehat{n}}(\alpha)) \in \overline{IVPA}(m)}{(\overline{\pi}\cdot \widehat{n}\cdot n,\ \alpha) \in \overline{IVPA}(n)}\ \text{RE}(m,n) \wedge \text{CE}(\widehat{n},l) \wedge l \notin \overline{\pi}'
\end{gathered} \tag{20}$$

We define $\overline{IVPC}(n)$ by modifying (2) similarly. It is easy to see that both $\overline{IVPA}(n)$ and $\overline{IVPC}(n)$ are non-empty for every n that is reachable from $Start_{main}$.

Now we can select an arbitrary $(\overline{\pi}, \sigma) \in \overline{IVPC}(n)$ and find the corresponding $(\overline{\pi}, H(\sigma)) \in \overline{IVPA}(n)$. By abuse of notation, we extend $\text{InA}_n(\alpha)$ to $\text{InA}_n(\overline{\pi}, \alpha)$ and $\text{InC}_n(\alpha)$ to $\text{InC}_n(\overline{\pi}, \sigma)$ as the values computed for node n along the single path $\overline{\pi}$ such that for every edge (m,n) in $\overline{\pi}$, $\text{InA}_n(\overline{\pi}, \alpha)$ is computed only from $\text{InA}_m(\overline{\pi}, \alpha)$ and $\text{InC}_n(\overline{\pi}, \sigma)$ is computed only from $\text{InC}_m(\overline{\pi}, \sigma)$. Note that if (m,n) is an edge $(\widehat{n}, n)$ abstracting out a call to procedure Q, InA_n is computed from $\text{InA}_{\widehat{n}}$ using $\eta_{\widehat{n}}^{Call}$, θ_Q, and η_n^{Ret} as explained in Figure 13 whereas InC_n is computed from $\text{InC}_{\widehat{n}}$ using f_Q.

Then, the soundness claim (19) follows if the following holds:

$$\forall Q \in \mathbb{P}\text{roc}, \forall n \in \mathbb{N}_Q, \forall(\overline{\pi}, \sigma) \in \overline{IVPC}(n).\ \ \text{Project}_Q(\text{InA}_n(\overline{\pi}, H(\sigma))) \sqsubseteq \text{InC}_n(\overline{\pi}, \sigma) \tag{21}$$

This can be easily proved by induction on the number of nodes in a manner similar to the proof in Section 2.4.3. The basis is trivially satisfied for $n\!:\!Start_{main}$ because from Equations (6) and (12),

$$\text{Project}_{main}(\text{InA}_{Start_{main}}(\pi, \alpha_0)) = \text{InC}_{Start_{main}}(\pi, \sigma_0) = BI$$

The inductive step can be proved using the the validity of the abstract value structure that guarantees that that the projection of a value in $\mathbb{M}$ computed by an abstract flow function coincides with the value in $\mathbb{L}$ computed by the corresponding flow function.

Since all methods in Table 2 have a valid abstract context structure, all of them are sound.

4.4.3 Precision Criteria. Intuitively, the precision of a sound method is guaranteed if its abstract context structure is rich enough to ensure that two concrete contexts σ_1 and σ_2 that may have distinct values (i.e., $\text{InC}_n(\sigma_1) \neq \text{InC}_n(\sigma_2)$) are not mapped to the same context by the mapping H. If $H(\sigma_1) = H(\sigma_2) = \alpha$, then in Equation (12), $\text{InA}_n(\alpha) = \text{InC}_n(\sigma_1) \sqcap \text{InC}_n(\sigma_2)$ leading to imprecision.[3] This merging manifests in two ways:

- If the underlying analysis is non-distributive, this merging causes imprecision in a callee procedure[4]. Note that this imprecision is unrelated to interprocedurally invalid paths.
  Hence, for precision, a method with such H, must place the restrictions of distributivity on the data flow framework supported. Out of the five methods with simple abstract context structure in Table 2, the restricted-contexts method requires separability (which implies distributivity, see Section 2.2) whereas the functional method and the IDE method require distributivity.
  We explain the differences caused by non-distributivity in such situations with the help of the examples in Figure 1. Available expressions analysis is distributive. In procedure R, the set of available expressions is calculated as the intersection of $\{a * b\}$ and $\{b * c\}$ implying that no expression is available within the body of procedure R. This result is precise.
  If we change the example to a non-distributive data flow framework such as points-to analysis (Figure 1), we observe that merging the points-to information from the two callers S and T and using it in node 9 leads to two spurious points-to pairs (a, d) and (c, b) that do not arise along any control flow path reaching node 10.
- Even if the underlying analysis is distributive, this merging causes imprecision in a caller procedure by accommodating interprocedurally invalid paths containing mis-matched call return pairs. The example of available expressions in Figure 1 illustrates this making both the expressions unavailable after the call in procedures S and T amounting to traversal of the following interprocedurally invalid paths

$$\pi_1 : 1 \longrightarrow 2 \longrightarrow 3\!:\!Call_R \longrightarrow 7\!:\!Start_R \longrightarrow 8 \longrightarrow 9\!:\!End_R \longrightarrow 13\!:\!Ret_R \longrightarrow 14 \longrightarrow 15$$
$$\pi_2 : 10 \longrightarrow 11 \longrightarrow 12\!:\!Call_R \longrightarrow 7\!:\!Start_R \longrightarrow 8 \longrightarrow 9\!:\!End_R \longrightarrow 4\!:\!Ret_R \longrightarrow 5 \longrightarrow 6$$

  The return node $13\!:\!Ret_R$ in π_1 does not correspond to call node $3\!:\!Call_R$ appearing in it because the two nodes belong to different procedures. Similarly, node $4\!:\!Ret_R$ in π_2 not correspond to call node $13\!:\!Call_R$ appearing in it.

Thus our precision criteria boils down to satisfying either of the following requirements:

[3]If some other σ is mapped to α, then $\text{InA}_n(\alpha) \sqsubseteq \text{InC}_n(\sigma_1) \sqcap \text{InC}_n(\sigma_2)$.
[4]The Ncontext function is defined only for call nodes so merging of contexts can happen only at a call node.

(P1) The abstract context structure should be rich enough to ensure that

$$\forall Q \in \mathbb{P}\text{roc}, \exists H : \mathbb{N}_Q \to \Sigma \to \mathbb{A} \text{ such that } \mathsf{InC}_n(\sigma) = \mathsf{Project}_Q(\mathsf{InA}_n(H_n(\sigma))) \tag{22}$$

This condition is sufficient for ensuring precision in both callees and callers because it does not cause merging of distinct data flow values at $n : Start_Q$.

(P2) If the abstract context structure is not rich enough to satisfy condition (**P1**), then the abstract value structure should be rich enough for precision. However, precision in callees and callers requires separate criteria:

(P2a) Precision in callees requires the data flow framework to be distributive.

(P2b) Precision in callers requires the abstract flow functions to satisfy the following:

(P2b.i) If $\eta^{Call}_{\widehat{n}}$ propagates some value from $\widehat{n} : Call_Q$ to $m : Start_Q$, then $\eta^{Ret}_n(v, w)$ should employ some selection function to select only the part of v that corresponds to the value propagated by $\eta^{Call}_{\widehat{n}}$ and reject the part of v that corresponds to the value propagated by some η^{Call} that is not associated with $\widehat{n}$.

(P2b.ii) If $\eta^{Call}_{\widehat{n}}$ does not propagates any value from $\widehat{n} : Call_Q$ to $m : Start_Q$, then no merging can happen at $m : Start_Q$. Thus, v represents the value resulting from this particular call. Hence, $\eta^{Ret}_n(v, w)$ should combine the effect of v with w because w represents the value before the call.

4.4.4 Precision of Methods in Table 2. The following methods are precise because they satisfy precision criterion (**P1**).

- The call-strings method. The required mapping is $H_n(\sigma) = \sigma$. However, we need to account for the fact that not all call strings are constructed for recursive programs and the length of the call strings is restricted to a known fixed length M. Since our model assumes their values to be $\top$, it does not change the value of the meet across all call strings. Hence it does not matter if condition (22) is not satisfied by H_n for call strings longer than M.
- The VBTCS method. This method is same as the call-strings method except for the use of the representation function and the required mapping is $H_n(\sigma) = R(\sigma \cdot n, \mathsf{InA}_n(\sigma))$. The representation function R, by definition, ensures that condition (22) is satisfied.
- VASCO. The required mapping in this case is $H_n(\sigma) = \mathsf{InC}_n(\sigma)$. Thus, if the data flow values of two concrete call strings σ_1 and σ_2 are distinct, the call strings are guaranteed to get mapped to two distinct abstract contexts by the Ncontext function.
- IFDS. The reasoning is similar to VASCO.

The following methods are precise because although they do not satisfy precision criterion (**P1**), they do satisfy precision criterion (**P2**).

- The restricted-contexts method satisfies (**P2a**) and (**P2b.i**).
- The functional and IDE methods satisfy (**P2a**) and (**P2b.ii**).

The remaining two methods are k-limiting and context-insensitive method (which is k-limiting with $k = 0$). They do not satisfy (**P1**) because of the following reasons.

- In context-insensitive method, all call strings get mapped to ϵ.
- In k-limited call-strings method. multiple call strings with different data flow values may get mapped to the same call string because of k-limiting.

Further, They do not satisfy (**P2**) because although they have a valid $\mathcal{V}$, it is not sufficiently rich.

5 BEYOND THE BASIC CONTEXT-SENSITIVE METHODS

In this section we present some other investigations that model context sensitivity and then describe several extensions of the context-sensitive methods described in the earlier sections.

5.1 Modelling Context Sensitivity

Formal semantics of different kinds of sensitivities (such as flow-, context-, and value-sensitivity) using abstract interpretation has been used to formalize the effect of these sensitivities (and their combinations) on the sets of states in a program [12]. Context sensitivity is one of the sensitivities that they consider and rely primarily on call-strings-based context sensitivity and object sensitivity can be found in the form of *record* and *merge* functions [31] whose combinations model these forms of context sensitivities.

5.2 Improving the Efficiency of Context-Sensitive Methods

Some improvements in VASCO [22] were presented in [36]. A comparison of contexts in VASCO comparing the data flow values to decide whether or not a new context is required. This could be expensive for analyses such as points-to analysis. Hence, instead of comparing the entire data flow values, it is proposed to compare only the relevant data flow values for a callee procedure [36]. The method also proposes to defer analyzing methods which do not affect the data flow value in the callers (referred as caller-ignorable methods). These improvements make VASCO more efficient without affecting the precision of the analysis.

Some extensions of graph-reachability-based algorithms [24, 27] were presented in [19]. For analyses where set $\mathbb{D}$ is large, the exploded supergraph used for graph reachability becomes very large. One of the proposed extensions is to construct the supergraph on demand as required by the analysis. Our unified formalism also shows that there is no need to construct the exploded supergraph and ideas from VASCO [22] can be used to avoid construction of large supergraphs. Another extension is useful for analysis which have subsumption relationships between the elements of data flow values. Instead of maintaining multiple data flow values, only the one which subsumes others can be maintained. This makes the analysis highly efficient. Some interesting insights about the efficiency of context-sensitive data flow analysis can be found in [3].

5.3 Improving the Precision of Graph-Reachability-Based Methods

When two calls are invoked on the same object they are said to be *correlated*. When such correlated calls are polymorphic and different types are considered for such calls, it results in infeasible paths being introduced in the interprocedural control flow graph as described in [37]. Such infeasible paths cannot be eliminated by the current notion of contexts using the context-sensitive methods described in this paper. The work in [23] presents a solution to eliminate such infeasible paths in IFDS method [24]. The proposed solution transforms the IFDS problem to an IDE problem by annotating the edges with the type of the object on which the call is made. Type of the object is then propagated along the interprocedural path and is used to eliminate such infeasible paths arising due to correlated calls from the control flow graph.

An extension proposed in [19] improves the precision of the graph-reachability-based algorithms [24, 27] for the analysis for programs in the SSA form. The imprecision in the SSA form of the program occurs due to merging immediately before the ϕ instruction. The extension delays the merge until after the ϕ instruction which increases the precision of the original method to a level similar to the original non-SSA form of the program. Another interesting twist to IFDS is an attempt to automatically transform an IFDS problem into an IDE problem for precise analysis of event-driven applications [40]. This increases the precision of the underlying analysis of an event-driven application by incorporating information about infeasible paths in IFDS.

5.4 Extending the Data Flow Frameworks Supported

IFDS method requires a data flow framework to be distributive—a requirement that is violated by practical analyses such as points-to analysis. Boomerang [34] provides an extension to the IFDS method to support non-distributive data flow frameworks. The part of the program which is distributive is modelled using the IFDS method. Non-distributive statements in pointer analysis are handled by adding additional nodes and edges in the exploded supergraph to accommodate the cartesian product of data flow values that a non-distributive flow function causes.

A functional method for points-to analysis [5] achieves precision without requiring distributivity by obviating the need of phase 2 for computing points-to information in callees. It uses the observation that the application of callee summaries in callers gives the points-to information of the pointers used in the callees. A small book-keeping for remembering statement numbers is sufficient to collect points-to information within the callees without needing phase 2. Since the BI_Q are not computed, there is no imprecision even if the data flow framework is not distributive.

5.5 Improving the Precision of Approximate Call-Strings Method

The approximate call-strings approach is popular [21, 35], but entails imprecision. Interestingly, it can also cause inefficiency by introducing spurious interprocedural cycles (called "butterfly" cycles) [21]. Such imprecision is mitigated by an extension that additionally records a call-site and enforces that the called procedure is analyzed for one call site at a time. Recording such additional information is similar to the restricted contexts method [17]. It also helps avoid the spurious cycles being introduced and thus makes the analysis efficient as well as more precise.

5.6 Other Context-Sensitive Methods

In this section, we mention further investigations that explore different possibilities to increase the effectiveness of context sensitivity.

Context Sensitivity Using Pushdown Systems. Context-sensitive analysis using weighted pushdown systems (WPDS) [25, 26] is similar to graph-reachability-based methods [24, 27]. They represent the exploded supergraph using a pushdown system and traverse it using the rules of a weighted pushdown system and in the process create an automaton. The transitions in the automaton correspond to the traversals of edges in the exploded supergraph. Since there is a one-to-one correspondence between WPDS and graph-reachability-based methods as far as context sensitivity is concerned, we have not discussed it separately.

Synchronized pushdown system [33] synchronizes the two pushdown systems, one to achieve context sensitivity (which is similar to WPDS [25, 26]) and the other to achieve field sensitivity.

Approximation in Case of Recursion. The methods discussed in [4, 38, 39] are context-sensitive for non-recursive procedures but consider recursive procedures context-insensitively. The use of invocation graphs for context-sensitive interprocedural points-to analysis [4] is similar to the use of call strings [29]. However, they introduce approximation in case of recursion [9].

The effect of call inlining for context sensitivity has also been attempted by cloning callee procedures context-sensitively and representing them using binary decision diagrams (BDDs) [38]. In the presence of recursion, all nodes in a strongly connected component are represented by a single node to get an acyclic graph. The analysis then uses a context-insensitive algorithm on the cloned call-graph to achieve context sensitivity.

Partial transfer functions are used to summarize the effect of a procedure in [39]. This is similar to analyzing a procedure only once for a data flow value reaching it and reusing the already computed procedure summary when the same data flow value is encountered, as done by value-based termination of call strings [10], VASCO [22] and graph-reachability-based methods [24, 27]. These methods are fully context-sensitive whereas the method [39] approximates in case of recursion by combining the information for all the nodes in a strongly connected component.

Use of Types to Define Contexts. Type-sensitive analysis [31] defines contexts using types of the objects on which call is made. Since the distinctions based on types cause a coarser approximation than those based on objects [16], type-sensitive analysis is more efficient than object-sensitive analysis. Both of them are beyond the scope of our work as explained in Section 6.

Tuning Context Sensitivity. The work presented in [20] selectively chooses the number of call sites to be distinguished based on a pre-analysis that decides where to use context sensitivity for precision gain. It also identifies the length of the call strings that need to be distinguished to achieve context sensitivity. A subsequent work tries to achieve the same using machine learning techniques [7]. Instead of suffixing a call site to a call string at every call, *context tunneling* updates contexts selectively and decides when to propagate the context without modification. This is achieved by developing a specialized data-driven algorithm, which is able to automatically search for high-quality heuristics for context tunneling.

Introspective analysis [32] proposes to refine context-sensitive analysis using metrics computed by a context-insensitive pass. The information generated by these heuristics estimates potential cost which will be incurred by a context-sensitive analysis. This is then used to identify the part of the program which will be performed context sensitively. Another thread of tuning context sensitivity is to restrict it to a subset of procedures that can benefit from context sensitivity [13] where a pre-analysis identifies suitable procedures.

Combining Different Context-Sensitive Methods. Hybrid context sensitivity [8] combines the call-strings method with object sensitivity. For static method calls the contexts are defined using the call strings method whereas for dynamic method calls they are defined using object-allocation sites. These contexts are then combined when a static call is made inside a dynamic call. Different variants of these hybrid context sensitivity are proposed to understand the precision and efficiency impact of the hybrid context sensitivity.

Similarly, context insensitivity and variants of context sensitivity (such as object-sensitivity or type-sensitivity) can be adopted for different procedures: Scaler [14] automatically adapts them at the level of individual procedures to maximize precision without compromising scalability.

6 CONCLUSIONS AND FUTURE WORK

In order to achieve precision, an analysis needs to compute context-sensitive information at the interprocedural level. The main goal of context sensitivity is to achieve the effect of inlining during the analysis and is achieved by ensuring proper call-return matching in interprocedural paths.

Table 1 (Section 3.10) has provided a summary of salient features of different context-sensitive methods. It is clear from the table and the preceding descriptions in Section 3 that these context-sensitive methods appear very dissimilar because they are algorithmically defined using different formalisms and it is difficult to compare their key ideas. This leads to a blind-men view in that it precludes forming a coherent and consistent view of various methods that solve essentially the same problem. However, when we model the same methods in our unified formalism, they cease to look all that different because of the well-identified rubrics of our model. This is evident from Table 2 where there are many more similarities in the methods than there are differences. Thus, our model provides a vocabulary for a meaningful comparison of these methods. Another strength of our model is that it facilitates reasoning about soundness and precision of methods that can be instantiated in our model by defining valid $\mathcal{A}$ and $\mathcal{V}$. While our soundness and precision criteria consists of only sufficient conditions, they are quite general to inspire a similar reasoning for methods that use very different $\mathcal{A}$ and $\mathcal{V}$ to make our criteria inapplicable.

A *necessary* requirement to achieve context sensitivity is to ensure call-return matching. In order to render the method practical, a *desirable* requirement is to reuse procedure summaries. We describe our take-aways from the process of a unified modelling of these methods.

- Contexts can be defined in terms of data flow values or some abstractions of call strings. The use of data flow values to form contexts enables reusability of procedure summaries.
- Call return matching may be done explicitly through an abstract context structure thereby keeping the abstract value structure simple. If we keep a simple $\mathbb{M}$ that coincides with $\mathbb{L}$, then we need a complex mechanism of call return matching. Otherwise, we can keep a very simple mechanism of call-return matching but then $\mathbb{M}$ has to be much richer because it has to share the burden of call-return matching. As an extreme variation of Equation (12) that computes mappings $\mathbb{A} \to \mathbb{L}$, we can push both $\mathbb{A}$ and $\mathbb{L}$ into $\mathbb{M}$ thereby completely doing away with the need of context matching.

Thus this model opens up a spectrum of possibilities by (a) distilling the essential features required by a context-sensitive method, (b) showing different possibilities of supporting them, (c) formalizing a mechanism of modelling the interplay between them, and (d) providing well-defined soundness and precision criteria in terms of sufficient conditions. A designer of a context-sensitive method can judiciously choose an appropriate point on this spectrum to design a suitable method.

We conclude by listing three possible extensions of the proposed unified model:

- *Handling indirect calls.* In our model, a call node $n : Call_Q$ has a single callee which is known before the analysis starts and remains fixed through the analysis. Thus, handling indirect calls through function pointers or through receiver objects of method calls, requires additional information supplied externally. This would require enhancing Equation (12) to handle multiple callees at a return node (the third case).
- *Handling object sensitivity.* Although there is some similarity in object sensitivity and handling indirect calls in that both of them need pointer-pointee relationships, in the former case this additional information merely adds more nodes to the call graph. The pointer data used for discovering the new caller-callee relationship does not become a part of the context of an analysis unless the method is used to perform pointer analysis. However, in object sensitivity, a context is defined jointly by the caller and the receiver object of the call; in some variants, the caller may be omitted from the context. Supporting object sensitivity in our model would require enriching call-string abstraction with points-to information.
- *Handling bidirectional flows.* The proposed model is currently restricted to unidirectional analyses. There are some important bidirectional analyses such as liveness-based points-to analysis [11], taint analysis [2] that have left context sensitivity implicit and a host of demand-driven methods some of which are context-sensitive and some context-insensitive [6]. We would like to extend our notion of context to uniformly extend all methods in our model to bidirectional analyses.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their incisive comments which pushed us to strengthen our model. We also thank Deepak D'Souza for suggesting the possibility of using our model to characterize soundness and precision of context-sensitive methods.

REFERENCES

[1] (Online Resource, Accessed on 3 May 2020). Blind Men and An Elephant. . https://en.wikipedia.org/wiki/Blind_men_and_an_elephant.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[3] Eric Bodden. 2018. The Secret Sauce in Efficient and Precise Static Analysis: The Beauty of Distributive, Summary-based Static Analyses (and How to Master Them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA '18)*. ACM, New York, NY, USA, 85–93. https://doi.org/10.1145/3236454.3236500

[4] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94)*. ACM, New York, NY, USA, 242–256. https://doi.org/10.1145/178243.178264

[5] Pritam M. Gharat, Uday P. Khedker, and Alan Mycroft. 2020. Generalized Points-to Graphs: A New Abstraction of Memory in Presence of Pointers. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 8:1–8:78.

[6] Swati Jaiswal, Uday P. Khedker, and Supratik Chakraborty. 2020. Bidirectionality in flow-sensitive demand-driven analysis. *Science of Computer Programming* 190 (2020), 102391. https://doi.org/10.1016/j.scico.2020.102391

[7] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and Scalable Points-to Analysis via Data-driven Context Tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276510

[8] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

[9] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. 2009. *Data Flow Analysis: Theory and Practice* (1st ed.). CRC Press, Inc., Boca Raton, FL, USA.

[10] Uday P. Khedker and Bageshri Karkare. 2008. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: resurrecting the classical call strings method. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction (CC08/ETAPS08)*. 213–228.

[11] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. 2012. Liveness-Based pointer analysis. In *Proceedings of the 19th international conference on Static Analysis (SAS12)*. 265–282.

[12] Se-Won Kim, Xavier Rival, and Sukyoung Ryu. 2018. A Theoretical Foundation of Sensitivity in an Abstract Interpretation Framework. *ACM Trans. Program. Lang. Syst.* 40, 3, Article 13 (Aug. 2018), 44 pages. https://doi.org/10.1145/3230624

[13] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided Context Sensitivity for Pointer Analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276511

[14] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first Pointer Analysis with Self-tuning Context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

[15] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-oriented Program Analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 305–315. https://doi.org/10.1145/1806596.1806631

[16] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41.

[17] Eugene M. Myers. 1981. A Precise Inter-procedural Data Flow Algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. ACM, New York, NY, USA, 219–230. https://doi.org/10.1145/567532.567556

[18] Nomair A. Naeem and Ondrej Lhotak. 2008. Typestate-like Analysis of Multiple Interacting Objects. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 347–366. https://doi.org/10.1145/1449764.1449792

[19] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. 2010. Practical Extensions to the IFDS Algorithm. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10/ETAPS'10)*. Springer-Verlag, Berlin, Heidelberg, 124–144. https://doi.org/10.1007/978-3-642-11970-5_8

[20] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-sensitivity Guided by Impact Pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318

[21] Hakjoo Oh and Kwangkeun Yi. 2010. An Algorithmic Mitigation of Large Spurious Interprocedural Cycles in Static Analysis. *Softw. Pract. Exper.* 40, 8 (July 2010), 585–603. https://doi.org/10.1002/spe.v40:8

[22] Rohan Padhye and Uday P. Khedker. 2013. Interprocedural Data Flow Analysis in Soot Using Value Contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis (SOAP13).* 31–36.

[23] Marianna Rapoport, Ondrej Lhoták, and Frank Tip. 2015. Precise Data Flow Analysis in the Presence of Correlated Method Calls. In *SAS.*

[24] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95).* ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[25] Thomas Reps, Stefan Schwoon, and Somesh Jha. 2003. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. In *Proceedings of the 10th International Conference on Static Analysis (SAS'03).* Springer-Verlag, Berlin, Heidelberg, 189–213. http://dl.acm.org/citation.cfm?id=1760267.1760283

[26] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming* 58, 1 (2005), 206 – 263. https://doi.org/10.1016/j.scico.2005.02.009 Special Issue on the Static Analysis Symposium 2003.

[27] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. In *Selected Papers from the 6th International Joint Conference on Theory and Practice of Software Development (TAPSOFT '95).* Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 131–170. http://dl.acm.org/citation.cfm?id=243753.243762

[28] John Godfrey Saxe. (Online Resource, Accessed on 3 May 2020). The Blind Men and the Elephant. . https://en.wikisource.org/wiki/The_poems_of_John_Godfrey_Saxe/The_Blind_Men_and_the_Elephant.

[29] M. Sharir and A. Pnueli. 1981. Two approaches to interprocedural data flow analysis.. In *Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications.* Prentice-Hall Inc., Englewood Cliffs.

[30] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages.* Technical Report.

[31] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL11).* 17–30.

[32] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14).* ACM, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

[33] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-sensitive Data-flow Analysis Using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290361

[34] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

[35] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Aliasing in Object-Oriented Programming. Springer-Verlag, Berlin, Heidelberg, Chapter Alias Analysis for Object-oriented Programs, 196–232. http://dl.acm.org/citation.cfm?id=2554511.2554523

[36] Manas Thakur and V. Krishna Nandivada. 2019. Compare Less, Defer More: Scaling Value-contexts Based Whole-program Heap Analyses. In *Proceedings of the 28th International Conference on Compiler Construction (CC 2019).* ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/3302516.3307359

[37] Frank Tip. 2015. Infeasible paths in object-oriented programs. *Sci. Comput. Program.* 97 (2015), 91–97.

[38] John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04).* ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

[39] Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95).* ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/207110.207111

[40] Ming-Ho Yee, Ayaz Badouraly, Ondrej Lhoták, Frank Tip, and Jan Vitek. 2019. Precise Dataflow Analysis of Event-Driven Applications. *CoRR* abs/1910.12935 (2019). arXiv:1910.12935 http://arxiv.org/abs/1910.12935

[41] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. Hybrid top-down and bottom-up interprocedural analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014.* 249–258. https://doi.org/10.1145/2594291.2594328