NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Explorations in Static Pointer Analysis: Adaptive Scalability and Strong Guarantees

George S. Kastrinis

ATHENS

JULY 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Εξερευνώντας τον χώρο της Στατικής Ανάλυσης Δεικτών: Κλιμάκωση και Ισχυρές Εγγυήσεις

Γεώργιος Σ. Καστρίνης

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2020

PhD THESIS

Explorations in Static Pointer Analysis: Adaptive Scalability and Strong Guarantees

George S. Kastrinis

SUPERVISOR: Yannis Smaragdakis, Professor NKUA

THREE-MEMBER ADVISORY COMMITTEE:

Yannis Smaragdakis, Professor NKUA

Alex Delis, Professor NKUA

Panos Rondogiannis, Professor NKUA

SEVEN-MEMBER EXAMINATION COMMITTEE

Yannis Smaragdakis,
Professor NKUA

Alex Delis,
Professor NKUA

Panos Rondogiannis,
Professor NKUA

Mema Roussopoulos,
Associate Professor NKUA

Manolis Koubarakis,
Professor NKUA

Panagiotis Stamatopoulos,
Assistant Professor NKUA

Nikolaos Papaspyrou,
Professor NTUA

Examination Date: July 6, 2020

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Εξερευνώντας τον χώρο της Στατικής Ανάλυσης Δεικτών: Κλιμάκωση και Ισχυρές Εγγυήσεις

Γεώργιος Σ. Καστρίνης

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Γιάννης Σμαραγδάκης, Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Γιάννης Σμαραγδάκης, Καθηγητής ΕΚΠΑ

Αλέξης Δελής, Καθηγητής ΕΚΠΑ

Παναγιώτης Ροντογιάννης, Καθηγητής ΕΚΠΑ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Γιάννης Σμαραγδάκης,
Καθηγητής ΕΚΠΑ

Αλέξης Δελής,
Καθηγητής ΕΚΠΑ

Παναγιώτης Ροντογιάννης,
Καθηγητής ΕΚΠΑ

Μέμα Ρουσσοπούλου,
Αναπληρώτρια Καθηγήτρια ΕΚΠΑ

Μανόλης Κουμπαράκης,
Καθηγητής ΕΚΠΑ

Παναγιώτης Σταματόπουλος,
Επίκουρος Καθηγητής ΕΚΠΑ

Νικόλαος Παπασπύρου,
Καθηγητής ΕΜΠ

Ημερομηνία Εξέτασης: 6 Ιουλίου 2020

ABSTRACT

Static program analysis aims to automatically reason about certain properties a given program might exhibit under all possible executions without actually observing such executions. Static *pointer* analysis is a major subcategory that focuses on the objects that program expressions might point to during program executions. The evolution of programming languages has led to the addition of many abstraction layers that, as a result, have made any automatic reasoning about a program a challenging task at best or an infeasible one at worst. Thus, any practical static pointer analysis algorithm has to compromise and aim to approximate results in some way—either computing more or less than what is actually true.

This dissertation shows how we can obtain *precise* yet also *scalable* static pointer analysis algorithms by carefully differentiating policies for different parts of the program. Furthermore, since a static pointer analysis algorithm with global soundness guarantees and meaningful results throughout is not realistic, we show that it is possible to design analyses that offer *strong guarantees* on the soundness of the results for specific parts of the program.

Pointer analyses in the past introduced the concept of *context-sensitivity* in order to tackle the ever growing problem of imprecision versus scalability. Context is used to annotate analysis components so that the analysis can be more precise without at the same time sacrificing scalability. We show beneficial ways to combine different context flavors for different parts of the program without paying the cost that a naive combination would incur.

Another attempt at producing precise yet scalable analyses leads us to an introspective analysis. We employ a common adaptive pattern in which a cheap imprecise analysis is run first so various metrics can be gathered, and then a more precise (and costly) analysis can be used only in parts of the program—under the assumption that more precise handling of the rest would only incur performance penalties.

Subsequently, we shift our attention to an analysis that *under*-approximates results (instead of the norm of *over*-approximating) so that it might report less but can guarantee those properties to always hold. We build upon observations on the properties that such analyses have in order to apply a specialized data structure that speeds up our algorithm by nearly two orders of magnitude.

Finally, in our last contribution, we revisit an analysis formulation that over-approximates results to create an analysis algorithm that is truly sound but at the same time highly efficient. Our analysis is conservative, guaranteeing soundness even in the presence of arbitrary unknown code, but avoids wasting any work on computations that will later be invalidated due to soundness concerns.

SUBJECT AREA: Programming Languages, Static Analysis

KEYWORDS: Pointer Analysis; Alias Analysis; Object-Oriented Programming; Precision; Performance; Context-Sensitivity

ΠΕΡΙΛΗΨΗ

Η στατική ανάλυση στοχεύει στον αυτόματο συμπερασμό ιδιοτήτων που κάποιο πρόγραμμα μπορεί να επιδείξει σε κάθε πιθανή εκτέλεση, χωρίς στην πράξη να εκτελείται. Η στατική ανάλυση *δεικτών* αποτελεί μια μεγάλη υποκατηγορία της που επικεντρώνεται στα δυναμικά αντικείμενα που δύνανται να 'δείξουν' οι εκφράσεις ενός προγράμματος σε κάποια εκτέλεση του. Η εξέλιξη των γλωσσών προγραμματισμού με την πάροδο των χρόνων οδήγησε στην προσθήκη πολλών επιπέδων αφαίρεσης, τα οποία σαν αποτέλεσμα έχουν ο αυτόματος συμπερασμός για κάποιο πρόγραμμα να αποτελεί τουλάχιστον μία πρόκληση αν όχι και μία αδύνατη προσπάθεια. Συνεπώς, κάθε πρακτικός αλγόριθμος στατικής ανάλυσης πρέπει να στοχεύσει σε μια εκτίμηση των πραγματικών αποτελεσμάτων με κάποια μορφή ανακρίβειας—είτε υπολογίζοντας περισσότερα είτε λιγότερα.

Σε αυτή τη διατριβή παρουσιάζουμε πώς μπορούμε να σχεδιάσουμε *ακριβείς* και συνάμα *αποδοτικούς* αλγορίθμους ανάλυσης δεικτών εφαρμόζοντας διαφορετικές πολιτικές σε διαφορετικά σημεία του προγράμματος. Συμπληρωματικά, δεδομένου ότι ένας αλγόριθμος ανάλυσης δεικτών με βεβαιώσεις εγκυρότητας για όλα τα σημεία του προγράμματος καθώς και πρακτικά αποτελέσματα δεν αποτελεί ρεαλιστική κατεύθυνση, δείχνουμε πώς μπορούμε να σχεδιάσουμε αναλύσεις με *ισχυρές βεβαιώσεις εγκυρότητας* για συγκεκριμένα κομμάτια ενός προγράμματος.

Προηγούμενοι αλγόριθμοι για ανάλυση δεικτών εισήγαγαν την έννοια των *συμφραζομένων* (context) για να αντιμετωπίσουν το αυξανόμενο πρόβλημα της ανακρίβειας έναντι της αποδοτικότητας. Τα συμφραζόμενα χρησιμοποιούνται για να επαυξήσουν στοιχεία της ανάλυσης ώστε η ανάλυση να καταφέρει να είναι πιο ακριβής χωρίς ταυτόχρονα να πρέπει να κάνει θυσίες στον τομέα της αποδοτικότητας. Παρουσιάζουμε επωφελείς τρόπους συνδυασμού διάφορων ειδών συμφραζομένων σε διαφορετικά σημεία του προγράμματος, χωρίς αυτοί οι συνδυασμοί να επιφέρουν το κόστος που θα παρουσίαζε μία αφελής προσέγγιση.

Μία δεύτερη απόπειρα για δημιουργία αναλύσεων που παρουσιάζουν υψηλή ακρίβεια και αποδοτικότητα μας οδηγεί σε μια ανάλυση *ενδοσκόπησης* (introspection). Εφαρμόζουμε ένα σύνηθες μοτίβο στο οποίο μια φτηνή ανακριβής ανάλυση εφαρμόζεται πρώτη ώστε να συλλέξει διάφορες μετρικές για το πρόγραμμα, και στη συνέχεια μια δεύτερη πιο ακριβής (και ακριβή) ανάλυση μπορεί να εφαρμοστεί μόνο σε συγκεκριμένα σημεία του προγράμματος—υπό την υπόθεση ότι η πιο ακριβής μεταχείριση των υπολοίπων θα είχε μόνο αρνητικά αποτελέσματα στην συνολική απόδοση.

Εν συνεχεία, μετατοπίζουμε την προσοχή μας προς μια ανάλυση που *υπό*-εκτιμά τα αποτελέσματα της (σε αντίθεση με το σύνηθες των αναλύσεων που υπολογίζουν μία *υπέρ*-εκτίμηση). Με αυτή την αντιμετώπιση, η ανάλυση μας αναφέρει λιγότερα αποτελέσματα αλλά μπορεί να παρέχει ισχυρές βεβαιώσεις ότι αυτά θα ισχύουν πάντα. Βασιζόμενοι πάνω σε παρατηρήσεις για τις ιδιότητες που παρουσιάζουν αναλύσεις αυτού του είδους, εφαρμόζουμε μια ειδική δομή δεδομένων η οποία επιφέρει επιταχύνσεις στον αλγόριθμο μας σχεδόν κατά δύο τάξεις μεγέθους.

Τέλος, στην τέταρτη συνεισφορά της διατριβής, επιστρέφουμε ξανά στην οικογένεια αναλύσεων

που υπερεκτιμούν τα αποτελέσματα τους. Ο στόχος μας είναι η δημιουργία ενός αρκετά αποδοτικού αλγορίθμου που όντως παράγει έγκυρα αποτελέσματα χωρίς περιορισμούς στο υποκείμενο πρόγραμμα. Κατά συνέπεια, αυτό μας οδηγεί σε μία συντηρητική ανάλυση, που μπορεί να παρέχει βεβαιώσεις εγκυρότητας ακόμα και υπό την παρουσία άγνωστου κώδικα, αλλά ταυτόχρονα αποφεύγει την σπατάλη υπολογισμών σε δεδομένα που αργότερα θα χρειαστεί να ανατραπούν για την διατήρηση των βεβαιώσεων αυτών.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Γλώσσες Προγραμματισμού, Στατική Ανάλυση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ανάλυση Δεικτών, Ανάλυση Συνωνύμων, Αντικειμενοστρεφής Προγραμματισμός, Ακρίβεια, Απόδοση

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Yannis Smaragdakis, for being such a great mentor to me, throughout my PhD studies and before. I immensely appreciate his endless encouragement, patient, and motivation; his advice and guidance have been priceless—even during passionate scientific arguments.

I also thank Alex Delis, Panos Rondogiannis, Mema Roussopoulos, Manolis Koubarakis, Panagiotis Stamatopoulos, and Nikolaos Papaspyrou for their valuable comments and advice while serving as members of my dissertation committee.

My sincere thanks to Shan Shan Huang, Martin Bravenboer, and Molham Aref, who gave me the opportunity to join LogicBlox as an intern, during my PhD. Furthermore, I also extend my thanks to Ben Livshits who gave me the opportunity for an internship in Microsoft Research, and later guided me as a mentor throughout the process. Both internships proved memorable experiences and I am indebted to all those who make them a reality.

A special thanks to my labmates throughout the years: Aggelos Biboudis, George Balatsouras, George Fourtounis, George Kollias, Anastasios Antoniadis, Kostas Saidis, Petros Pathoulas, Neville Grech, Christos Vrachas, Sifis Lagouvardos, Kostas Ferles, Efthymios Hadjimichael, Dimitris Galipos, Stamatis Kolovos, Konstantinos Triantafyllou and Ilias Tsatiris. I am grateful for our interactions, discussions, and potential arguments over every possible aspect of programming languages that intrigued us to no end, and for all the fun we had.

Finally, I would like to thank Anna for always being positive and understanding and my parents, Stavros and Aristea, for being so supportive and reassuring over all these years.

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Η διατριβή αυτή ανήκει στον ευρύτερο τομέα της στατικής ανάλυσης προγραμμάτων, η οποία στοχεύει στον αυτόματο συμπερασμό των ιδιοτήτων που παρουσιάζει κάποιο πρόγραμμα, με βάση την εξέταση του πηγαίου του κώδικα (ή κάποιας αντίστοιχης ενδιάμεσης αναπαράστασης), αλλά δίχως να απαιτείται κάποια πραγματική εκτέλεση του. Η δουλειά μας στη διατριβή αυτή επικεντρώνεται σε μια μεγάλη υποκατηγορία της στατικής ανάλυσης, αυτή της ανάλυσης *δεικτών*. Μία ανάλυση δεικτών στοχεύει στο να υπολογίσει τα σύνολα αντικειμένων στα οποία μπορεί να 'δείξει' κάθε έκφραση του προγράμματος (π.χ. τοπική μεταβλητή, πεδίο, κτλ.) σε όλες τις πιθανές εκτελέσεις του.

Σαν αποτέλεσμα, κάθε πρακτικός αλγόριθμος που στοχεύει να παρέχει ουσιαστικά αποτελέσματα αναγκάζεται να κάνει έναν πρώτο συμβιβασμό: χρειάζεται να κατασκευάσει κάποιο *αφηρημένο* μοντέλο της μνήμης, όπου *εικονικά* αντικείμενα αναπαριστούν (μία ή περισσότερες) *διακριτές* δεσμεύσεις πραγματικών αντικειμένων. Ένα κλασικό παράδειγμα αυτού είναι η δέσμευση αντικειμένων από κάποια εντολή μέσα σε μία δομή επανάληψης. Η συνηθισμένη αντιμετώπιση από κάποιον αλγόριθμο ανάλυσης δεικτών είναι να θεωρηθούν όλα τα αντικείμενα που εν δυνάμει θα δεσμευτούν από την ίδια εντολή, σαν ένα μοναδικό, αφηρημένο αντικείμενο. Αυτό αποτελεί μία (από τις πολλές) πήγη ανακρίβειας στα αποτελέσματα της όποιας ανάλυσης. Ταυτόχρονα όμως, συμβιβασμοί σαν αυτόν, αν και οδηγούν σε εκτιμήσεις της συμπεριφοράς ενός προγράμματος και όχι σε απόλυτα αποτελέσματα, επιτρέπουν στις αναλύσεις να κάνουν πολύπλοκους αυτόματους συμπερασμούς. Συμπερασμούς που βοηθούν σε πληθώρα τομέων όπως η μηχανικά υποβοηθούμενη κατανόηση του προγράμματος, η εύρεση σφαλμάτων, και η βελτιστοποίηση της απόδοσης του προγράμματος.

Τα παραπάνω φανερώνουν ένα από τα βαθύτερα προβλήματα κάθε αλγορίθμου στατικής ανάλυσης δεικτών. Δηλαδή ότι, συχνά, ο σχεδιασμός ενός τέτοιου αλγορίθμου είναι αποτέλεσμα ισορροπίας μεταξύ θεμάτων ακριβείας και κλιμάκωσης. Είναι σχετικά απλό μία ανάλυση να επικεντρωθεί στον υπολογισμό αποτελεσμάτων υψηλής ακρίβειας, θυσιάζοντας την γενική απόδοση του αλγορίθμου. Αντίστοιχα, είναι δυνατόν να σχεδιαστούν πολύ αποδοτικοί αλγόριθμοι, οι οποίοι όμως θα υπολογίζουν μεγάλες εκτιμήσεις των (πραγματικών) αποτελεσμάτων οδηγώντας σε τεράστια ανακρίβεια.

Η διατριβή αυτή στοχεύει στην αντιμετώπιση του παραπάνω προβλήματος, με την κύρια θέση της να συνοψίζεται ώς εξής:

> Είναι δυνατόν να σχεδιαστούν αλγόριθμοι στατικής ανάλυσης δεικτών που παρουσιάζουν *υψηλή ακρίβεια* αλλά και *κλιμάκωση*, εφαρμόζοντας προσεκτικά διαφορετικές πολιτικές σε διαφορετικά σημεία του προγράμματος. Συμπληρωματικά, είναι δυνατόν να σχεδιαστούν αναλύσεις που προσφέρουν *ισχυρές εγγυήσεις εγκυρότητας* των αποτελεσμάτων, αλλά για στοχευμένα κομμάτια του προγράμματος.

Στη συνέχεια, θα παρουσιάσουμε διάφορες τεχνικές για την υλοποίηση αποδοτικών αλγορίθ-

μων ανάλυσης δεικτών, στο περιβάλλον της γλώσσας προγραμματισμού Java, προσαρμόζοντας προσεκτικά την στρατηγική του κάθε αλγορίθμου σε διαφορετικά σημεία του προγράμματος. Επιπροσθέτως, θα παρουσιάσουμε δύο αλγορίθμους αμυντικής φύσης που στοχεύουν στον υπολογισμό αποτελεσμάτων υψηλής εμπιστοσύνης, ακόμη και εν μέσω 'εχθρικού' ή άγνωστου κώδικα.

Βασικές Έννοιες της Στατικής Ανάλυσης Δεικτών

Πριν κάνουμε μια συνοπτική αναφορά των επιστημονικών συνεισφορών της διατριβής αυτής, είναι απαραίτητο να γίνει μια μικρή εισαγωγή σε βασικές έννοιες της στατικής ανάλυσης (δεικτών), που αποτελούν το επιστημονικό και τεχνικό υπόβαθρο της δουλειάς μας.

Πλατφόρμα Υλοποίησης & Γλώσσα υπό Ανάλυση. Το μεγαλύτερο μέρος της δουλειάς που θα παρουσιάσουμε στη συνέχεια, είναι υλοποιημένο στην πλατφόρμα DOOP[16], χρησιμοποιώντας τη δηλωτική γλώσσα προγραμματισμού Datalog. Το DOOP αποτελεί εδώ και χρόνια μία καλά εδραιωμένη πλατφόρμα ανάπτυξης αλγορίθμων στατικής ανάλυσης δεικτών, προσφέροντας μία πληθώρα αναλύσεων που στοχεύουν προγράμματα Java. Περισσότερες λεπτομέρειες δίνονται στο Κεφάλαιο 2.11.

Αξίζει να σημειωθεί ότι, αν και οι ιδέες και οι αλγόριθμοι που εξερευνώνται παρακάτω επικεντρώνονται γύρω από την ανάλυση προγραμμάτων Java, είναι αρκετά πιθανή μία γενίκευση τους, σε μικρότερο ή μεγαλύτερο βαθμό, και σε άλλες γλώσσες προγραμματισμού που προσφέρουν παρόμοια χαρακτηριστικά και ακολουθούν παρόμοια μοντέλα/παραδείγματα.

Χρήση Συμφραζομένων. Όπως προαναφέρθηκε, η υλοποίηση κάθε πολύπλοκου αλγορίθμου ανάλυσης δεικτών σύντομα καταλήγει σε μία προσπάθεια εξισορρόπησης μεταξύ ακρίβειας και απόδοσης. Στο παρελθόν, η επιστημονική κοινότητα έχει επεκτείνει το οπλοστάσιο της με διάφορες έννοιες και τεχνικές προς διαχείριση αυτής της κατάστασης. Μία τέτοια τεχνική, που στοχεύει στην καταπολέμηση της ανακρίβειας των αποτελεσμάτων, ελπίζοντας χωρίς ταυτόχρονα επιβάρυνση της απόδοσης, είναι αυτή των *συμφραζομένων* (context). Η χρήση των συμφραζομένων (πρακτικά επιπλέον πληροφορίας) γίνεται επαυξάνοντας στοιχεία της εκάστοτε ανάλυσης (π.χ., τοπικές μεταβλητές, πεδία και μεθόδους) ώστε η ανάλυση να καταφέρει να τα χειριστεί με μεγαλύτερη ακρίβεια.

Η κεντρική ιδέα είναι ότι η ανάλυση θα διαφοροποιήσει τον χειρισμό στοιχείων του προγράμματος όταν αυτά συνδυάζονται με κάποια συμφραζόμενα, ενώ θα τα χειριστεί ομοιογενώς όταν συνδυάζονται με κάποια άλλα. Για παράδειγμα, μία ανάλυση μπορεί να εξετάσει διαφορετικά κάποια μεθόδου όταν η κλήση της έγινε μέσα στη μέθοδο `A`, μέσα στη μέθοδο `B` ή οπουδήποτε αλλού (δηλαδή παρουσιάζοντας τρεις διαφορετικές τιμές συμφραζομένων).

Δύο βασικές κατηγορίες συμφραζομένων έχουν χρησιμοποιηθεί ευρέως στο παρελθόν: τα συμφραζόμενα *σημείων-κλήσης* (που οδηγούν στις λεγόμενες *call-site sensitive* αναλύσεις) όπου τα συμφραζόμενα δομούνται από εντολές κλήσης μέσα στον κώδικα του προγράμματος, και τα συμφραζόμενα *αντικειμένων* (που οδηγούν στις λεγόμενες *object sensitive* αναλύσεις) όπου τα συμφραζόμενα δομούνται από τα αφηρημένα αντικείμενα-παραλήπτες πάνω στα οποία εφαρμόζονται οι τυχόν κλήσεις συναρτήσεων. Περισσότερες λεπτομέρειες δίνονται στο Κεφάλαιο 2.2.

'May' έναντι 'Must' Αναλύσεων. Όπως αναφέραμε στην αρχή, ο στόχος κάθε αλγορίθμου στατικής ανάλυσης είναι ο αυτόματος συμπερασμός για κάποιο σύνολο συμπεριφορών που δύναται να επιδείξει ένα πρόγραμμα *σε όλες* τις πιθανές εκτελέσεις του. Μια τέτοια προσπάθεια είναι ένα μη-αποφασίσιμο πρόβλημα για τα περισσότερα σύνολα συμπεριφορών παρά για τα πιο τετριμμένα από αυτά (για ένα τυχαίο πρόγραμμα προς ανάλυση). Κατά συνέπεια, κάθε πρακτικός αλγόριθμος αναγκάζεται να κάνει κάποια εκτίμηση του συνόλου των συμπεριφορών προς μία από τις δύο εξής κατευθύνσεις: είτε θα υπολογίσει μία *υπέρ*-εκτίμηση των αποτελεσμάτων και θα αναφέρει όλες τις πιθανές συμπεριφορές του προγράμματος καθώς και κάποιες που δεν είναι δυνατόν να προκύψουν ποτέ, είτε θα υπολογίσει μία *υπό*-εκτίμηση και θα αναφέρει μόνο ένα υποσύνολο των πιθανών συμπεριφορών. Μία αδρή κατηγοριοποίηση των αναλύσεων μπορεί να γίνει κάτω από αυτό το πρίσμα σε may-αναλύσεις (που αναφέρουν υπερεκτιμήσεις της πραγματικότητας) και σε must-αναλύσεις (που αναφέρουν υποεκτιμήσεις της πραγματικότητας). Περισσότερα στο Κεφάλαιο 2.6.

Εγκυρότητα Αποτελεσμάτων. Ένας θεωρητικός όρος που συχνά χρησιμοποιείται για να χαρακτηρίσει έναν αλγόριθμο στατικής ανάλυσης είναι αυτός της *εγκυρότητας*. Με απλά λόγια, λέμε ότι ένας αλγόριθμος είναι έγκυρος όταν τα αποτελέσματα που υπολογίζει συνάδουν με τους αρχικούς ισχυρισμούς του. Για παράδειγμα, μία may-ανάλυση δεικτών ισχυρίζεται ότι σκοπεύει να υπολογίσει μία υπερεκτίμηση του συνόλου των αντικειμένων στα οποία μπορεί να δείξει κάθε έκφραση ενός προγράμματος, σε κάθε πιθανή εκτέλεση του. Αν δεν λείπει κάποιο ζευγάρι 'έκφραση/αντικείμενο', που θα μπορούσε πραγματικά να συμβεί στο πρόγραμμα, από τα αποτελέσματα της ανάλυσης, τότε ο αλγόριθμος χαρακτηρίζεται από εγκυρότητα. Έτσι, μία τετριμμένα έγκυρη, αλλά και παντελώς άχρηστη, may-ανάλυση δεικτών είναι μία που υπολογίζει το καρτεσιανό γινόμενο κάθε έκφρασης του προγράμματος με κάθε αφηρημένο αντικείμενο.

Ποικίλοι παράγοντες οδηγούν τους περισσότερους αλγορίθμους may-ανάλυσης δεικτών στο να θυσιάζουν την εγκυρότητα σε κάποιο βαθμό ώστε να καταφέρουν να διατηρήσουν κάποιο ποσοστό κλιμάκωσης. Μια πιο αναλυτική συζήτηση γύρω από το θέμα της εγκυρότητας ακολουθεί στα Κεφάλαια 2.7-2.9.

Δομή Διατριβής και Επιστημονικές Συνεισφορές

Το περιεχόμενο της διατριβής δομείται σε εννέα κεφάλαια. Το πρώτο κεφάλαιο δίνει μία σύντομη αναφορά του γενικού χώρου της στατικής ανάλυσης δεικτών, εδραιώνει την κεντρική θέση της διατριβής καθώς και τις επιστημονικές συνεισφορές της, και τέλος, παρουσιάζει την δόμηση που θα ακολουθηθεί στη συνέχεια του κειμένου.

Το δεύτερο κεφάλαιο περιέχει μία σύντομη περιγραφή χρήσιμων και απαραίτητων εννοιών, τεχνικών και εργαλείων από την υπάρχουσα επιστημονική βιβλιογραφία, που αποτελούν την υποκείμενη βάση για την δουλειά μας.

Υβριδικές Αναλύσεις Συμφραζομένων. Τα συμφραζόμενα αντικειμένων εισήχθησαν το 2002 από την Milanova [106] ως εναλλακτική των συμφραζομένων σημείων-κλήσης. Από τότε υπάρχει πληθώρα ενδείξεων ότι αποτελούν τη βέλτιστη επιλογή είδους συμφραζομένων, όσον αφορά προγράμματα εκφρασμένα σε αντικειμενοστρεφείς γλώσσες, εξασφαλίζοντας υψηλή

ακρίβεια με χαμηλότερο συγκριτικά κόστος. Τόσο μεγάλη ήταν η επιτυχία τους που έχουν πρακτικά αντικαταστήσει την κλασική εναλλακτική των σημείων-κλήσης. Παρ΄ όλα αυτά, τα συμφραζόμενα σημείων-κλήσης δεν είναι πάντα υποδεέστερα καθώς υπάρχουν συγκεκριμένα χαρακτηριστικά γλωσσών και μοτίβα προγραμματισμού που ευνοούν αυτή την επιλογή.

Συνεπώς, δεν είναι παράλογη μία προσέγγιση όπου και τα δύο είδη συμφραζομένων συνδυάζονται, ομοιογενώς, με κάπως αφελή τρόπο, σε κάθε σημείο του προγράμματος στοχεύοντας ώστε τα οφέλη στην ακρίβεια να είναι ακόμα μεγαλύτερα. Όντως, ένας τέτοιος συνδυασμός έχει σαν αποτέλεσμα κάποια βελτίωση στον τομέα της ακρίβειας, αλλά στις περισσότερες των περιπτώσεων μία τέτοια βελτίωση συνοδεύεται με ένα απαγορευτικά υψηλό κόστος.

Απόρροια αυτής της παρατήρησης είναι η πρώτη μας επιστημονική συνεισφορά, που παρουσιάζεται στο τρίτο κεφάλαιο. Εκεί περιγράφουμε μία προσπάθεια προς έναν πιο εκλεπτυσμένο συνδυασμό των δύο ειδών συμφραζομένων. Η *υβριδική* μας προσέγγιση οδηγεί σε μία οικογένεια αναλύσεων όπου τα διαφορετικά είδη συμφραζομένων συνδυάζονται μόνο σε συγκεκριμένα σημεία του προγράμματος, ώστε η ακρίβεια της ανάλυσης να έχει τα οφέλη της ύπαρξης όλων των ειδών χωρίς όμως να χρειάζεται να πληρώσει και το αντίστοιχο κόστος.

Πιο συγκεκριμένα, η κεντρική ιδέα των υβριδικών αλγορίθμων μας έγκειται στη χρήση των συμφραζομένων αντικειμένων σαν το κυρίαρχο είδος, για την ανάλυση αντικειμενοστρεφών χαρακτηριστικών του προγράμματος στα οποία και προσφέρουν τα περισσότερα οφέλη, και τον συνδυασμό τους με την πιο κλασική εναλλακτική των συμφραζομένων σημείων-κλήσης εκεί που τα πρώτα υστερούν. Το πιο χαρακτηριστικό τέτοιο σημείο προγράμματος είναι η κλήση στατικών συναρτήσεων, όπου και δεν υπάρχει η κατάλληλη πληροφορία που χρειάζονται τα συμφραζόμενα αντικειμένων. Οι κλασικοί αλγόριθμοι ανάλυσης δεικτών αντιμετωπίζουν το πρόβλημα αυτό μεταφέροντας πληροφορία από το πιο πρόσφατο κατάλληλο σημείο (που μπορεί να απέχει από το σημείο της στατικής κλήσης), με την ελπίδα ότι η πληροφορία αυτή θα αποβεί χρήσιμη ξανά στο μέλλον. Στο ενδιάμεσο όμως, η επιπλέον αυτή πληροφορία προσφέρει ελάχιστα στη γενική ακρίβεια του αλγορίθμου ενώ η ύπαρξη της δεν έρχεται χωρίς (κάποιες φορές βαρύ) κόστος. Μία υβριδική αντιμετώπιση θα επιλέξει για τα σημεία αυτά (και μόνο) την χρήση συμφραζομένων σημείων-κλήση, τα οποία είναι ικανά να βελτιώσουν την τοπική ακρίβεια της ανάλυσης χωρίς να την επιβαρύνουν σημαντικά.

Σαν αποτέλεσμα, αυτός ο επιλεκτικός συνδυασμός συμφραζομένων οδηγεί σε αναλύσεις σημαντικά ανώτερες όχι μόνο συγκριτικά με αυτές που ακολουθούν κάποιο αφελή συνδυασμό συμφραζομένων, αλλά και ακόμα σε σχέση με τις κλασικές, 'κανονικές', μη-υβριδικές αναλύσεις. Αυτό προκύπτει συμπερασματικά με τη συλλογή εκτεταμένων πειραματικών δεδομένων από μεγάλα προγράμματα Java. Για παράδειγμα, σε σύγκριση με μία αρκετά διαδεδομένη και χρήσιμη ανάλυση που χρησιμοποιεί συμφραζόμενα αντικειμένων μήκους δύο, η προσέγγιση μας προσφέρει επιταχύνσεις της τάξης του **1.53x** αλλά και καλύτερη ακρίβεια.

Ανάλυση Ενδοσκόπησης. Το τέταρτο κεφάλαιο παρουσιάζει τη δεύτερη επιστημονική συνεισφορά μας, γύρω από την προσπάθεια των αλγορίθμων ανάλυσης δεικτών να επιτύχουν καλή απόδοση και κλιμάκωση χωρίς να εγκαταλείψουν την ακρίβεια των αποτελεσμάτων. Όμως, είναι συχνό φαινόμενο στο χώρο αυτό, οι αναλύσεις να βρίσκονται σε ένα εκ των δύο άκρων του φάσματος: είτε έχουν αρκετή ακρίβεια ώστε το σύνολο των δεδομένων υπό ανάλυση να παραμένει διαχειρίσιμο και σαν αποτέλεσμα να επιτυγχάνουν μία εντυπωσιακή κλιμάκωση, είτε

γρήγορα εκτροχιάζονται στο πρώτο σημάδι σημαντικής ανακρίβειας και καταλήγουν να είναι τάξεις μεγέθους πιο κοστοβόρες σε σχέση με το αναμενόμενο με βάση το μέγεθος του αναλυόμενου προγράμματος.

Προς την αντιμετώπιση αυτού του ζητήματος κινείται και η προσέγγιση μας σε αυτό το κεφάλαιο, προτείνοντας μία ανάλυση *ενδοσκόπησης* που απαρτίζεται από δύο βήματα. Η προσέγγιση αυτή επιτρέπει στην ανάλυση να παρουσιάζει μία ομοιόμορφη κλιμάκωση σε μεγάλα προγράμματα Java, εξαλείφοντας τα προβληματικά, πιθανά φαινόμενα απόδοσης, έχοντας μόνο ένα μικρό αρνητικό αντίκτυπο στην τελική ακρίβεια.

Η ανάλυσή μας εφαρμόζει ένα γνωστό μοτίβο: πρώτα εκτελεί μία ανάλυση που δεν χρησιμοποιεί συμφραζόμενα, και άρα είναι ανακριβής αλλά και φτηνή και γρήγορη, και στη συνέχεια, με βάση τα δεδομένα που συλλέχθηκαν στην πρώτη φάση, εκτελεί μία πιο εκλεπτυσμένη ανάλυση (δηλαδή, με χρήση κάποιου είδους συμφραζομένων) αλλά μόνο για συγκεκριμένα σημεία του προγράμματος. Η κατεύθυνση αυτή ευελπιστεί ότι η επιπλέον ακρίβεια στα σημεία αυτά δεν θα είναι τελικά απαγορευτική για την συνολική απόδοση του αλγορίθμου. Άρα, η πρόκληση του όλου εγχειρήματος βρίσκεται στην κατάλληλη επιλογή αυτών των σημείων. Δείχνουμε ότι μία πειθαρχημένη προσέγγιση μπορεί να προβεί αρκετά αποτελεσματική, επιφέροντας κλιμάκωση με σημαντικές βελτιώσεις στην απόδοση σε προγράμματα που στο παρελθόν δεν ήταν δυνατόν να αναλυθούν με ακρίβεια.

Στο κεφάλαιο αυτό, παρουσιάζουμε διάφορες μετρικές για την αξιολόγηση της πληροφορίας από την πρώτη φάση, και στη συνέχεια δύο ευριστικές για την μετέπειτα επιλογή των κατάλληλων σημείων του προγράμματος που θα αναλυθούν με μεγαλύτερη ακρίβεια στη δεύτερη φάση.

Συλλέγοντας αρκετά πειραματικά δεδομένα, επιβεβαιώνουμε τα οφέλη μίας ανάλυσης ενδοσκόπησης. Εξερευνούμε τις πιθανές απώλειες σε ακρίβεια αλλά και τις βελτιώσεις σε απόδοση και κλιμάκωση, δοκιμάζοντας διαφορετικές παραμέτρους στις μετρικές και ευριστικές μας. Τελικά, εξακριβώνουμε ότι, ακόμα και με παραμέτρους που στοχεύουν σε υψηλή ακρίβεια, η ανάλυση μας είναι αποτελεσματική στην διαχείριση προβληματικών περιπτώσεων, που προηγουμένως ήταν αδύνατον να αναλυθούν χωρίς τρομακτική απώλεια ακρίβειας. Αυτά τα πειραματικά συμπεράσματα εδραιώνουν την εμπιστοσύνη μας στον ισχυρισμό ότι οι αναλύσεις συμφραζομένων μπορούν να χρησιμοποιηθούν ευρέως και όχι απλά μεμονωμένα σε εκείνες τις περιπτώσεις που 'δουλεύουν αρκετά καλά'.

Στη συνέχεια της διατριβής, στρέφουμε την προσοχή μας σε αναλύσεις που επικεντρώνονται στον υπολογισμό αποτελεσμάτων τα οποία συνοδεύονται με μεγάλη εμπιστοσύνη. Αν και αυτό οδηγεί σε συντηρητικές, αμυντικές αναλύσεις, συχνά απρόθυμες να προβούν σε νέους συμπερασμούς, όταν τελικά αναφέρουν κάποιο αποτέλεσμα το κάνουν με ισχυρή βεβαιότητα στην εγκυρότητα του.

Must-Ανάλυση Συνωνύμων - Ένα Λογικό Μοντέλο. Το πέμπτο κεφάλαιο, στο οποίο περιγράφεται η τρίτη επιστημονική συνεισφορά μας, αρχίζει μία εξερεύνηση προς μία διαφορετική κατεύθυνση. Πρώτον, αντί για μία ανάλυση δεικτών, παρουσιάζουμε μία ανάλυση *συνωνύμων* (aliases). Μία ανάλυση αυτού του είδους έχει σκοπό τον υπολογισμό των εκφράσεων ενός προγράμματος που αποτελούν συνώνυμα, δηλαδή δείχνουν στο ίδιο αντικείμενο στη μνήμη. Οι αναλύσεις συνωνύμων συνδέονται στενά με τις αναλύσεις δεικτών, αλλά έχουν

και βασικές διαφορές. Δεύτερον, η ανάλυση που προτείνουμε ανήκει στην οικογένεια των must-αναλύσεων, υπολογίζει δηλαδή μία υποεκτίμηση της πραγματικότητας. Αυτό συνεπάγεται ότι, μία τέτοια ανάλυση αποτυγχάνει να υπολογίσει κάποια ισχύοντα ζευγάρια συνωνύμων, αλλά αυτά τα οποία τελικά θα υπολογίσει είναι σίγουρο ότι ισχύουν.

Οι ισχυρές βεβαιώσεις που συνοδεύουν μία ανάλυση αυτού του είδους, καθιστούν τα αποτελέσματα της ιδανικά για αρκετές εφαρμογές: (1) είναι σημαντικά για πληθώρα βελτιστοποιήσεων σε μεταγλωττιστές, (2) μπορούν να βελτιώσουν την ακρίβεια προγραμμάτων για τον έλεγχο σφαλμάτων, για παράδειγμα, ανιχνευτές μη-τερματισμού ή λάθους δεικτοδότησης (null-reference) σε εκφράσεις του προγράμματος, (3) μπορούν να χρησιμοποιηθούν σαν δομικά στοιχεία πιο σύνθετων και πολύπλοκων αναλύσεων, και (4) μπορούν να προβούν ανεκτίμητα στην άμεση κατανόηση του προγράμματος από τον προγραμματιστή.

Για να καταφέρει μία ανάλυση να επιδείξει τόσο υψηλή εμπιστοσύνη στα αποτελέσματα της, χρειάζεται να σέβεται την ροή του προγράμματος, να διατηρεί σύνολα αποτελεσμάτων ξεχωριστά για κάθε σημείο του προγράμματος, και να μεταφέρει σε επόμενα σημεία μόνο όποιο υποσύνολο της πληροφορίας συνεχίζει να ισχύει. Με μία πρώτη ματιά, μία τέτοια προσέγγιση φαντάζει αυτονόητη, αλλά οι περισσότερες (may-) αναλύσεις δεικτών δεν κάνουν αυτή την επιλογή καθώς κάτι τέτοιο προσφέρει λίγο στην ακρίβεια τους και ταυτόχρονα επιβαρύνει αρκετά την απόδοση τους. Στην περίπτωση μίας αμυντικής ανάλυσης όμως, αυτή η κατεύθυνση είναι παραπάνω από απαραίτητη.

Έτσι, αρχικά, στο πέμπτο κεφάλαιο παρουσιάζουμε ένα μινιμαλιστικό μοντέλο της ανάλυσης μας, εκφρασμένο στη δηλωτική γλώσσα Datalog. Το μοντέλο είναι αρκετά εκλεπτυσμένο για να περιγράψει τα κύρια χαρακτηριστικά που πρέπει να διαθέτει μία must-ανάλυση συνωνύμων, αλλά ταυτόχρονα και αρκετά απλό ώστε να μπορούμε να επικεντρωθούμε στην ουσία της ανάλυσης και όχι στους πολύπλοκους τρόπους με τους οποίους αλληλεπιδρούν τα διάφορα στοιχεία της γλώσσας που αναλύουμε.

Επιπροσθέτως, αξίζει να σημειωθεί ότι το μοντέλο μας παρουσιάζει μία μη συμβατική χρήση των συμφραζομένων. Οι κλασικές αναλύσεις εφαρμόζουν τα συμφραζόμενα σε μία προσπάθεια για βελτίωση της ακρίβειας, δηλαδή σαν ένα επιπλέον, θετικό αλλά προαιρετικό στοιχείο. Η ανάλυση μας χρησιμοποιεί τα συμφραζόμενα σαν ένα εργαλείο εξασφάλισης της εγκυρότητας των αποτελεσμάτων. Όποτε χρειάζεται να εξερευνήσει πέρα από τα τοπικά όρια μίας συνάρτησης, το κάνει επεκτείνοντας τα υπάρχοντα συμφραζόμενα, στο μέτρο που κάτι τέτοιο επιτρέπεται από τις παραμέτρους της ανάλυσης. Όταν κάτι τέτοιο δεν είναι πια δυνατόν, σταματάει κάθε προσπάθεια για συμπερασμούς καθώς αυτοί θα οδηγούσαν σε μη έγκυρα αποτελέσματα. Συνεπώς, στην προσέγγιση μας τα συμφραζόμενα αποτελούν αναπόσπαστο κομμάτι του μοντέλου, επιτρέποντας τους συμπερασμούς της ανάλυσης να υπερβούν τα στενά όρια κάθε συνάρτησης.

Ένα ακόμη θετικό στοιχείο της μοντελοποίησης μας είναι το γεγονός ότι η απουσία κομματιών του υπό-ανάλυση προγράμματος (π.χ., κώδικα βιβλιοθηκών) δεν έχει κάποιο αρνητικό αντίκτυπο στην συνολική εγκυρότητα της ανάλυσης. Η παρουσία πιθανώς επιπλέον κώδικα οδηγεί στο συμπερασμό ακόμα περισσότερων αποτελεσμάτων, χωρίς όμως κάτι τέτοιο να ακυρώνει τους προηγούμενους υπολογισμούς της ανάλυσης.

Must-Ανάλυση Συνωνύμων - Ειδικές Δομές Δεδομένων. Η προσεκτική παρα-

τήρηση του παραπάνω μοντέλου αποκαλύπτει διάφορα σημαντικά χαρακτηριστικά μίας must-ανάλυσης συνωνύμων, αλλά και τις ανάγκες που πρέπει να καλύψει κάθε πιθανή υλοποίηση. Σαν αποτέλεσμα, στο έκτο κεφάλαιο παρουσιάζουμε μία ειδική δομή δεδομένων, η οποία αξιοποιώντας αυτές τις παρατηρήσεις επιφέρει σημαντικές βελτιώσεις στην απόδοση και την κλιμάκωση της ανάλυσης.

Πιο συγκεκριμένα, η πρώτη παρατήρηση είναι ότι όταν μία must-ανάλυση υπολογίζει την πληροφορία συνωνύμων, στην πράξη διατηρεί μία σχέση ισοδυναμίας.[1] Για παράδειγμα, αν η ανάλυση υπολογίσει ότι οι μεταβλητές `x` και `y` αποτελούν συνώνυμα, και το ίδιο αντίστοιχα και οι μεταβλητές `y` και `z`, τότε πρέπει αυτομάτως να συμπεριλάβει και το ζευγάρι {`x` με `z`} (καθώς και όλα τα συμμετρικά, {`y` με `x`}, {`z` με `y`}, και {`z` με `x`}). Μία ρητή αναπαράσταση όλων των ζευγαριών μπορεί να επιφέρει σημαντική επιβάρυνση στην απόδοση του αλγορίθμου, ειδικά όταν αυτό συνδυαστεί και με την επόμενη παρατήρηση.

Η δεύτερη παρατήρηση έχει να κάνει με το γεγονός ότι η ανάλυση μας δεν περιορίζεται σε συμπερασμούς για εκφράσεις του προγράμματος μήκους ένα (δηλαδή, τοπικές μεταβλητές), αλλά συμπεριλαμβάνει και μεγαλύτερες, όπως για παράδειγμα η `obj.fld1.fld2` (μέχρι κάποιο μέγιστο μήκος, παράμετρο της ανάλυσης). Το υποκείμενο πρόβλημα είναι το εξής: για παράδειγμα, αν η ανάλυση έχει υπολογίσει ότι δύο μεταβλητές `x` και `y` είναι συνώνυμα, τότε πρέπει επίσης να υπολογίσει ρητά και ζευγάρια σαν τα {`x.f` με `y.f`}, {`x.g` με `y.g`}, {`x.f.h` με `y.f.h`}, κτλ. Ένας τέτοιος συμπερασμός οδηγεί σε εκθετικό πλήθος ζευγαριών.

Για τους παραπάνω λόγους, εισάγουμε μία ειδική δομή δεδομένων, η οποία αποτυπώνει τη σχέση συνωνύμων που υπολογίζει μία must-ανάλυση, και ταυτόχρονα αξιοποιεί τις παραπάνω παρατηρήσεις με αποτέλεσμα τη σημαντική βελτίωση της συνολικής απόδοσης. Η δομή μας έχει τη μορφή κατευθυνόμενου γράφου, όπου κάθε κόμβος αναπαριστά ομάδες μεταβλητών (δηλαδή, τάξεις ισοδυναμίας), με κάθε μεταβλητή-μέλος να είναι συνώνυμη με όλες τις άλλες που βρίσκονται στον ίδιο κόμβο. Κάθε ακμή αναπαριστά την πρόσβαση σε κάποιο πεδίο, ενώ η φορά της ακμής κωδικοποιεί ποιός κόμβος 'δείχνει' σε ποιόν (με τον ίδιο τρόπο που συναντάται η έννοια σε μία ανάλυση δεικτών). Σύνθετες εκφράσεις του προγράμματος, μήκους μεγαλύτερου του ένα, κωδικοποιούνται έμμεσα στα μονοπάτια που εμφανίζονται μεταξύ των κόμβων του γράφου.

Τέλος, επιβεβαιώνουμε πειραματικά τα θεωρητικά οφέλη της ειδικής δομής που παρουσιάσαμε, παρατηρώντας βελτιώσεις στην απόδοση του αλγορίθμου κατά δύο τάξεις μεγέθους. Με αυτή την προσέγγιση, καθίσταται δυνατή η αποδοτική εφαρμογή της must-ανάλυσης συνωνύμων σε μεγάλα προγράμματα Java, με χρόνους εκτέλεσης συχνά κάτω από μισό λεπτό.

Αμυντική Ανάλυση Δεικτών. Στο έβδομο κεφάλαιο ολοκληρώνουμε την παρουσίαση των επιστημονικών συνεισφορών της διατριβής, περιγράφοντας την τέταρτη και τελευταία από αυτές. Επιστρέφουμε ξανά στην οικογένεια των may-αναλύσεων δεικτών, αυτή τη φορά με κυρίαρχο στόχο τον υπολογισμό πραγματικά έγκυρων αποτελεσμάτων, ακόμα και υπό την παρουσία άγνωστου ή 'εχθρικού' κώδικα. Η ανάλυση μας δεν θέτει κάποιο περιορισμό στα χαρακτηριστικά που χρησιμοποιεί το υπό-ανάλυση πρόγραμμα, και επίσης προσπαθεί να μην κάνει εκπτώσεις στην συνολική απόδοση του αλγορίθμου για να επιτύχει τους στόχους της.

[1] Αντιθέτων, όπως περιγράφουμε πιο αναλυτικά στο κείμενο του κεφαλαίου, η σχέση συνωνύμων σε μία may-ανάλυση *δεν* αποτελεί σχέση ισοδυναμίας.

Η ανάλυση μας, όντας μέλος της οικογένειας των may-αναλύσεων, έχει σαν στόχο τον υπολογισμό μίας υπερεκτίμησης της πραγματικής συμπεριφοράς του προγράμματος. Για να καταφέρει ταυτόχρονα να τηρήσει τους ισχυρισμούς εγκυρότητας που δίνει, θα πρέπει όταν για οποιοδήποτε λόγο δεν είναι σίγουρη για το σύνολο αντικειμένων στα οποία μπορεί να δείξει κάποια έκφραση, να αναφέρει ότι μπορεί να δείξει στα πάντα.

Μία αφελής προσέγγιση του θέματος οδηγεί στον περιττό υπολογισμό αρκετών αποτελεσμάτων, τα οποία στη συνέχεια θα ακυρωθούν ή έμμεσα θα υποσκελιθούν από άλλα, ώστε τελικά να τηρηθεί η εγκυρότητα. Κάτι τέτοιο έχει φυσικά σημαντικές, αρνητικές επιπτώσεις στην συνολική απόδοση της ανάλυσης. Η προσέγγισή μας καταφέρνει να παραμείνει αποδοτική χωρίς να ζημιώνει την συνολική εγκυρότητα, αναβάλλοντας τον υπολογισμό πληροφορίας μέχρις ότου είναι βέβαιη ότι αυτή δεν θα ακυρωθεί σε μετέπειτα στάδιο.

Σαν αποτέλεσμα, οι παραπάνω σχεδιαστικές επιλογές επιτρέπουν στην ανάλυση μας να είναι αρκετά αποδοτική, επιτυγχάνοντας υψηλά επίπεδα ακριβείας, αδύνατα για τις κλασικές, προϋπάρχουσες αναλύσεις. Παρά την αρκετά συντηρητική και αμυντική φύση της, η ανάλυση καταφέρνει να δώσει έγκυρα και άμεσα εφαρμόσιμα αποτελέσματα για ένα μεγάλο υποσύνολο του υπό-ανάλυση προγράμματος. Πειραματικά, κάτω από τις πιο απαισιόδοξες και αμυντικές παραμέτρους, γίνεται κάλυψη του **34-74%** του προγράμματος σε σύγκριση με μία από τις καλύτερες, αλλά μη-έγκυρες, αναλύσεις του χώρου.

Τέλος, στα εναπομείναντα κεφάλαια της διατριβής, δίνεται ο επίλογος της εν λόγω δουλειάς. Στο όγδοο κεφάλαιο διερευνούμε σχετική ερευνητική δουλειά του χώρου, για τις τέσσερεις επιστημονικές συνεισφορές μας. Κλείνοντας, στο ένατο και τελευταίο κεφάλαιο σκιαγραφώνται μελλοντικές ερευνητικές κατευθύνσεις και γίνεται μία τελική εκτίμηση της διατριβής.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

> Don't be scared. All of this is new to you, and new can be scary. Now we all want answers. Stick with me—you might get some.
>
> *The 13th Doctor* - Doctor Who

Static program analysis is the cornerstone of several modern programming facilities and tools for program development and aided program understanding. Nowadays, it is an umbrella term for many different methodologies (Hoare logic [46, 63, 112, 128], model checking [25, 26, 40, 118], symbolic execution [15, 65, 74, 113], abstract interpretation [27–29], data-flow analysis [68, 72, 73, 107, 124, 138], and so on) all with the ultimate goal of inferring a program's properties, without the need of an actual execution. It is routinely employed in many different contexts: compilers, bug detectors, verifiers, security analyzers, IDEs, and a myriad other tools.

The main intention of any *static* program analysis algorithm is to reason about the set of all feasible behaviors (under some abstraction of behaviors) that a given program might exhibit under all possible executions. For example, could this method throw a runtime exception? or is that type cast possible to fail under some program input? etc. As a result, virtually all interesting static program analysis questions are undecidable—indeed the prototypical undecidable problem, the *halting problem*, is a static program analysis question: will a program terminate under all inputs?

Pointer analysis (also known as *points-to analysis*) is a fundamental subdomain of static program analysis that consists of computing some *abstract memory model* for a given program. The essence of such an analysis is to compute a set of possible objects that a program variable or expression may point to during program execution. A straightforward endeavor at first, it quickly gets too complicated in practice due to all of the intricate details one has to take into account and the multitude of different features that mutually depend on each other.[1] Although a challenging task, smart implementations of pointer analysis can bear many benefits to client analyses that will subsequently consume the results to reason about specialized behaviors (e.g., security vulnerabilities or potential optimization opportunities).

A closely related analysis, sometimes confused with pointer analysis, is *alias analysis* in which one computes sets of program expressions that may alias (i.e., point to common objects) with each other. Pointer analysis could—although it is not the only possible alternative—be used to implement an alias analysis algorithm, and vice versa.

At the same time, programming languages are evolving, becoming ever higher-level and more complex. Many abstraction levels are added throughout the years with the aim of making the

[1] The analysis inputs are large and the analysis algorithms are typically quadratic or cubic, but try to maintain near-linear behavior in practice, by exploiting program properties and maintaining precision—more precise (i.e., smaller) inference sets lead to less work.

very task of programming easier for developers allowing them to express more with less effort (e.g., in terms of lines of code). Frequently, new features come with complicated semantics regarding their possible implementations and usually they interact in intricate ways with pre-existing ones.

Additionally, modern software paradigms have evolved as well. Complex design patterns have become the norm for experienced developers, immense libraries and frameworks are accepted as a prerequisite for any non-trivial software, and over-involved build tools often make even the task of understanding all of the program's dependencies a challenge.

It comes as no surprise that any kind of static analysis has struggled to keep up with this ever-increasing complexity both in programming languages and software. Even the seemingly simple task of computing a program's call-graph (i.e., which methods are called at every invocation site) requires sophisticated analysis for achieving acceptable precision. Thus, the main emphasis of pointer analysis algorithms is on combining fairly precise modeling of pointer behavior and memory abstractions with scalability.

Thesis.

> *Precise* yet *scalable* static pointer analysis algorithms can be obtained by careful choice of different policies for different parts of the program. In a complementary fashion, analyses can be designed to offer (uniquely) *strong guarantees* on the soundness of results, but for a part of the program only.

We provide a number of techniques for implementing scalable static pointer and alias analyses in the setting of Java programs by configuring the analysis strategy differently for different code parts. Additionally, we present a couple of defensive algorithms for reporting high-confidence results even in the presence of hostile or unknown program points.

1.1 Pointer Analysis Crash Course

Before enumerating the scientific contributions of this dissertation, it is mandatory to introduce certain concepts related to pointer analysis, that comprise the scientific and technical base of this work. This is by no means a detailed presentation of said concepts—a more elaborate introduction will follow in Chapter 2.

Implementation Platform & Target Language. Most of the following work and algorithms have been expressed in the Doop framework [16]. Doop is a well established pointer analysis framework offering a wide variety of full-fledged algorithms for static pointer analysis of Java programs. More in Section 2.11.

Context Sensitivity. Implementing any sophisticated pointer analysis algorithm quickly turns out to be a balancing act between precision and performance tradeoffs. Any attempt

for a scalable algorithm might inadvertently be accompanied by significant precision losses whereas an endeavour for highly precise results might also enforce huge performance penalties.

Throughout the years, the scientific community has amassed a few tools in its arsenal in order to tackle this conundrum of precision versus performance. Among those tools, a widely employed notion, that aims to improve precision without having to pay an unbearable performance cost, is that of *context* resulting in *context-sensitive* algorithms. An algorithm will use additional information (also known as context) to annotate analysis components with the aim of countering potential precision losses. The key idea is that the analysis will differentiate the handling of program elements under some contexts while it will collapse it under others. For instance, an algorithm might differentiate the analysis of a method when called from method `A` or method `B` or anywhere else (thus under three different contexts).

Two main kinds of context have been widely used in the past; in *call-site-sensitive* analyses call instructions comprise the context elements, whereas in *object-sensitive* analyses context is based on the identity of the calling object at each method invocation. More in Section 2.2.

May vs. Must Analyses. The goal of any static program analysis algorithm is to reason about a set of behaviors under all potential program executions. This endeavour is an undecidable problem for any set of behaviors other than the most trivial ones. As a consequence, any practical algorithm has to *approximate* results in one of two directions; either *over*-approximate and both report all possible behaviors and also some that will never actually arise, or to *under*-approximate and be conservative by reporting only a subset of potential arising behaviors. Analyses are often categorized as *may*-analyses when they over-approximate results, and as *must*-analyses when they under-approximate results. More in Section 2.6.

Soundness. A formal term often used to accompany static pointer analysis algorithms is that of *soundness*. In layman's terms, an algorithm is said to be sound when it actually does what it claims. For instance, a may-pointer analysis claims that it aims to over-approximate the set of objects that various program expressions may point to in all possible program executions. If the results are not missing any such inference that could arise in a program execution, then the algorithm is sound. Due to various factors, most may-pointer analysis algorithms forgo soundness in order to maintain scalability. A more detailed discussion regarding soundness will follow in Sections 2.7-2.9.

1.2 Scientific Contributions

In this section, we will briefly explain the main scientific contributions of this dissertation. As already mentioned, the exploration happens in the context of analyzing Java—mainly by use of the Doop framework—although it is not far-fetched to generalize results to other languages that offer similar features and follow similar paradigms.

Ever since the introduction of object sensitivity by Milanova et al. [106], there has been increasing evidence that it is the superior context choice for programs expressed in object-oriented languages, yielding a high precision to cost ratio. Such has been its success that in practice it has almost superseded the use of more traditional call-site-sensitive analyses in object-oriented languages. Nevertheless, a call-site-sensitive analysis is not always inferior as there are language features and code patterns that may partially favor this kind of context abstraction.

Consequently, one might consider an approach where both context flavors are—naively—combined in every program point with the goal of increasing the precision of the end result. Truly, such a combination would bear some precision benefits but in most cases it would be accompanied by an infeasibly high cost.

First contribution. Our first scientific contribution is a step towards a more sophisticated handling, aiming to achieve a beneficial combination of both context flavors. We propose a *hybrid* context flavor for defining a family of analyses where classical contexts are mixed and combined only in those program points where it is profitable for the analysis. The resulting selective combination of both context kinds vastly outperforms not only analyses following the naive non-selective combination approach, but also their "normal" object-sensitive counterparts. This result holds for a large array of context-sensitive analyses establishing a new set of performance/precision sweet spots.

Second contribution. The second scientific contribution tries to tackle an oft-reported issue with context-sensitive analyses, in that they mostly operate in two extremes: either the analysis is precise enough that it manipulates only manageable sets of data, and thus scales impressively well, or the analysis gets quickly derailed at the first sign of—massive—imprecision and becomes orders-of-magnitude more expensive than would be expected given the program's size. Currently, there is no approach for a *precise*, context-*sensitive* (of any context flavor) analysis that would scale across the board at a level comparable to that of a context-*insensitive* one. Instead, we propose a two step process by means of *introspective analysis*: the approach uniformly scales context-sensitive analyses by eliminating the performance-detrimental behavior, only at a small precision expense.

Introspective analysis employs a common adaptive pattern: it first performs a context-insensitive analysis and then it uses the results to selectively refine (i.e., analyze context-sensitively) only those program elements that are expected not to cause an explosion in running time or memory space. The technical challenge is to appropriately identify such program elements. We show that a simple but principled approach can be remarkably effective, achieving scalability (often with dramatic speedup) for benchmarks previously completely out-of-reach for deep context-sensitive analyses.

For the last two contributions, we shift our attention towards analyses that aim for the highest confidence in their claims. Although quite reluctant and conservative in making a claim, when they actually do they make certain that it is the correct decision.

Third contribution. The next, third, contribution features a different flavor of static program analysis. Instead of the more commonly researched paradigm of *may*-analyses, we

chose to explore the alternative approach of a *must*-analysis. More specifically, we focus on an instance of a *must-alias* (also known as *definite-alias*) analysis that aims to infer aliasing relationships among program expressions that are guaranteed to always hold.[2] The applications of a must-alias analysis are manifold: (1) it is useful for enabling optimizations such as constant folding and register allocation, (2) it can increase the precision of bug detectors, e.g., greatly benefiting a null-reference detector and a non-termination detector, and (3) it can be used internally as part of more complex analyses, e.g., one that can reason correctly about "strong updates" at instructions that modify the heap. In order to compute high-confidence, non-trivial results, the analysis needs to be flow-sensitive, i.e., compute information at each program point and propagate it forward while respecting the control-flow of the program.

Furthermore, we observe that a must-alias analysis exhibits certain properties that can be exploited in order to achieve a more efficient algorithm without any compromise in the precision or the validity of its results. We present a custom specialized *data structure* that speeds up a must-alias analysis by nearly two orders of magnitude. The data structure achieves its efficiency by encoding multiple alias sets in a single linked structure, and compactly representing the aliasing relations of arbitrarily long program expressions. Under this approach, must-alias analysis can be performed efficiently, over large Java benchmarks, in under half a minute, making the analysis cost acceptable for most practical uses.

Fourth contribution. For our last contribution, we revisit the setting of a may-analysis but this time while aiming to explore the potential of a truly *sound*—instead of just soundy—yet *practical* analysis. We present such an approach in a *defensive* may-point-to analysis, which can guarantee soundness even in the presence of arbitrary opaque code.[3] A key design tenet of our approach is *laziness*: the analysis computes points-to relationships only for program expressions that are guaranteed to never escape into opaque code.

The defensive nature of our analysis means that it might miss some valid inferences, but because of its laziness it will never waste work to compute sets that are not "complete", i.e. that may be missing elements due to opaque code. This frugal approach is what enables the great efficiency of the algorithm, allowing for a highly precise points-to analysis (such as a 5-call-site-sensitive, flow-sensitive analysis). Despite its conservative nature, the analysis yields sound, actionable results for a large subset of the program code, achieving (under worst-case assumptions) **34-74%** of the program coverage of an unsound state-of-the-art analysis for real-world programs.

1.3 Outline

The rest of this dissertation is organized as follows:

[2] As previously mentioned, a must-analysis will aim to compute an under-approximation of behaviors that will happen in every possible program execution.

[3] Code that cannot be analyzed such as dynamically generated or native code, or dynamic language features such as reflection, `invokedynamic`, etc.

- Chapter 2 offers a quick yet non-trivial introduction to certain notions or properties that are important to take under consideration when designing a sophisticated static pointer analysis algorithm.
- Chapter 3 examines how a naive combination of object sensitivity and call-site sensitivity into a single analysis can be massively penalizing in terms of performance. Following that, we presents a hybrid context-sensitive approach for implementing points-to analyses that leverage the benefits of combining both object and a call-site sensitivity while avoiding to pay most of the cost of a naive combination.

  This chapter presents research previously published in *"Hybrid Context-Sensitivity for Points-To Analysis"* [70].
- Chapter 4 examines the well-known, bi-modal nature of classical static program points-to analyses in regards to scalability; they are either quite scalable or not scalable at all. In order to counter that discrepancy, we propose an adaptive approach in introspective analysis, where an imprecise analysis is used as a stepping stone in order to fine-tune program points in which a more precise handling is both beneficial and not detrimental to the overall analysis's performance.

  This chapter presents research previously published in *"Introspective Analysis: Context-sensitivity, Across the Board"* [144].

Both aforementioned contributions aim for more scalable analyses that achieve superior performance without foregoing precision. The next three contributions aim for analyses that although more restrained on what they report, they do so with much more confidence in the accuracy of their claims.

- Chapter 5 examines how to compose a declarative model of a rich family of must-alias analyses, with emphasis on a careful and compact modeling, while at the same time exposing the key points where the algorithm's inference power can be adjusted.

  This chapter presents research previously published in *A Datalog Model of Must-Alias Analysis* [8].
- Chapter 6 builds upon the previous chapter and goes forth to provide a specialized data structure that by exploiting the nature of a must-alias analysis it achieves high performance without any sacrifice on the accuracy of its results. We explore the data structure's performance in both an imperative (implemented in Java) and a declarative (implemented in Datalog) setting and contrast it extensively with prior techniques.

  This chapter presents research previously published in *An Efficient Data Structure for Must-Alias Analysis* [71].
- Chapter 7 examines how a defensive reasoning in the presence of opaque code can be combined along with computational laziness in order to produce a highly efficient, highly precise and truly sound may-points-to analysis.

  This chapter presents research previously published in *Defensive Points-To Analysis: Effective Soundness via Laziness* [143], that also received a *Distinguished Paper* award.

- Chapter 8 first discusses related work that is specific to previous chapters, and then expands to various other interesting subjects in the broader realm of static analysis.
- Chapter 9 concludes this dissertation by assessing our initial thesis and discussing future work.

2. BACKGROUND

> Once upon a tim— No! Once upon several times...
>
> *The Master* - Doctor Who

There are a few important design choices that can affect drastically the properties a static program analysis algorithm will enjoy and the reasoning that is required to achieve such goals. A bird's-eye view is given below.

2.1 Naive Pointer Analysis

Before delving into designing any sophisticated pointer analysis algorithm, it is helpful to have a short view on a naive attempt to track the set of objects possible pointed by program variables in all program executions. This will also give an insight on the issues that arise from such a naive approach.

For instance, in the code snippet of Figure 2.1, the analysis will infer that variable `obj1` may point to memory object `o1` (i.e., `obj1` $\mapsto$ {`o1`}), and variable `obj2` to memory object `o2` (i.e., `obj2` $\mapsto$ {`o2`}). Taking into account the method invocation of `foo` in line 10, the analysis will infer that argument variable `arg` may point to `o1` (since `obj1` also points to the same object). Following the same reasoning in line 12, it will also infer that `arg` may point to `o2` (i.e, `arg` $\mapsto$ {`o1, o2`}). This correctly reflects the fact that the argument of method `foo` will point to either object during different program points.

```
1  class A {
2      void foo(Object arg) { return arg; }
3  }
4
5  class B {
6      void bar(A a1, A a2) {
7          obj1 = new ...  // allocated object o1
8          obj2 = new ...  // allocated object o2
9          ...
10         obj3 = a1.foo(obj1);
11         ...
12         obj4 = a2.foo(obj2);
13     }
14 }
```

Figure 2.1: Code snippet for illustrating pointer analysis algorithms.

But here is one point where the weakness of a naive approach starts to show. When reasoning about the return instruction in line 2, the analysis will infer that method `foo` might return both memory objects, since the return variable was inferred to possibly point to both.

Subsequently, the analysis will infer that both variables `obj3` and `obj4` might point to both objects, given that they are assigned whatever the method invocation might return (i.e., `obj3` $\mapsto$ {`o1, o2`} and `obj4` $\mapsto$ {`o1, o2`}). This is clearly wrong since, for example, there is no possible program execution under which variable `obj3` might point to `o2` (assuming no other code interferes).

In following sections we will present countermeasures one could include when designing a pointer analysis algorithm in order to reclaim portions of lost precision. The aforementioned naive algorithm is an instance of *context-insensitive* analyses. The next section clarifies the notion of *context*.

2.2 Context Sensitivity

Throughout the years, pointer analysis has evolved and has been the focus of intense research. It is widely accepted to be among the most standardized and well-understood interprocedural analyses (i.e., going outside method borders when reasoning about a property).

The emphasis of points-to analysis algorithms is on combining fairly precise modeling of pointer behavior with scalability. The challenge is to pick judicious approximations that will allow satisfactory precision at a reasonable cost. Furthermore, although increasing precision often leads to higher asymptotic complexity, this worst-case behavior is rarely encountered in actual practice. Instead, techniques that are effective at maintaining good precision often also exhibit better average-case performance, since smaller points-to sets lead to less work.

A widely used concept that emerged as a powerful tool for tuning precision while still achieving scalable analyses, is that of *context sensitivity*. It consists of qualifying interesting components of an analysis, such as program expressions, object abstractions or method invocations, with additional *context* information. The core idea being that the analysis will collapse information (e.g., "what objects this method argument may point to") for executions that result in the same context, while keeping separate information for different contexts. In essence, qualifying components with additional context is as if each such component is replaced with multiple versions (one for each different associated context value) and the analysis can reason individually for each version. This approach tries to counter the loss of precision that naturally arises in any static analysis, from conflating information of different dynamic program paths.

Two main flavors of context sensitivity have been explored in past literature: (1) *call-site sensitivity* (also known as *kCFA*) [137, 140] in which call-sites are used to qualify variables and other analysis components, essentially re-analyzing a method for different call-sites that target that method, and (2) *object sensitivity* [105, 106, 142] in which receiver objects of a call are used instead, in a similar manner. Another kind of context sensitivity, known as *type sensitivity*, has also been explored as an approximation of object sensitivity with the aim of preserving high precision at substantially reduced cost. In type sensitivity, upper bounds on the dynamic types of the receiver objects are employed as context elements.

A context-sensitive analysis has a second axis of parameterization besides context flavor—

that of (max) context depth. Consequently, a common way to describe an analysis is using the following naming scheme: X-FLAVOR-sensitive+Y-heap, e.g., as in 3-call-site-sensitive+2-heap. Here FLAVOR denotes the kind of context information being employed, and, X and Y denote the context depth limits being used at invocation sites and at object allocations respectively. In the previous example the analysis is keeping track of the 3 most recent call-sites that led to the current method call, in order to annotate local variables. Similarly, the analysis is using the 2 most recent call-sites that led to the allocation site of an object to annotate the newly allocated object.

2.2.1 Call-Site Sensitivity (k-CFA)

As previously mentioned, a call-site-sensitive analysis uses method call sites (i.e., labels of invocation instructions) as context elements. The analysis separates information on program expressions, such as local variables, per call-stack (i.e., sequence of k most recent call-sites) that led to the current method call. Similarly, the analysis separates information on heap objects per call-stack that led to the object's allocation.

For instance, in the code snippet of Figure 2.1, a *1-call-site-sensitive* analysis (unlike a context-insensitive analysis) will distinguish the two call-sites of method `foo` in lines 10 and 12. This results in an analysis that will treat `foo` separately in two cases: that of its formal argument, `arg`, pointing to anything `obj1` may point to, and that of `arg` pointing to anything `obj2` may point to.

An equivalent mental model is that of two instances of method `foo` being analyzed—`foo_COPY1` and `foo_COPY2`. The invocation in line 10 leads to `foo_COPY1`, whereas the one in line 12 leads to `foo_COPY2`. After the analysis has concluded, if one is to query regarding method `foo`, information from both instances will be collapsed into as single answer-set. Therefore, following the reasoning of Section 2.1, the analysis will infer, among others, that `foo_COPY1::arg` $\mapsto$ {`o1`} and `foo_COPY2::arg` $\mapsto$ {`o2`}, and as a result that `obj3` $\mapsto$ {`o1`} and `obj4` $\mapsto$ {`o2`}. As already mentioned, an implicit inference is that of `arg` $\mapsto$ {`o1, o2`}, regarding the unqualified analysis of method `foo`.

2.2.2 Object Sensitivity

In contrast, an object-sensitive analysis uses allocation sites (i.e., labels of instructions containing a **`new`** statement) as context elements. (Hence, a better name for "object sensitivity" might have been "allocation-site sensitivity".) When a method is called on an object (also known as the call *receiver*), the analysis separates information depending on the allocation site of that object, as well as other, previous allocation sites used as context.

Thus, in the code of Figure 2.1, a *1-object-sensitive* analysis will analyze `foo` separately on all the allocation sites of objects that variables `a1` and `a2` may point to. If, for example, `a1` and `a2` are inferred to potentially point to objects from two and three distinct allocation sites respectively, `foo` will effectively be analyzed under five different contexts. On the other hand, if the analysis has inferred that both `a1` and `a2` may only point to objects from a

single common allocation site, then the method will only be analyzed under one context.

It is not apparent from this code fragment—and this also holds in the general case–neither whether `a1` and `a2` may point to different objects, nor to how many: the allocation site of the receiver object may be remote and unrelated to the method call itself. Similarly, it is not possible to compare the precision of an object-sensitive and a call-site-sensitive analysis in principle. In this example, it is not even clear whether the object sensitive analysis will examine all calls to `foo` as one case, as two, or as many more, since this depends on the allocation sites of all objects that the analysis *itself* computes to flow into `a1` and `a2`.

2.3 Intraprocedural vs. Interprocedural Analyses

Different kinds of static program analysis may define differently which program parts are of interest. An *intraprocedural* analysis makes its reasoning using only the local information that is available in each program function. Multiple analyses commonly found in a classical compiler, such as *type-checking* or the computation of *live-ranges* for program variables are well-known examples of intraprocedural static program analyses. On the contrary, an *interprocedural* analysis is one whose reasoning transcends function boundaries, taking into account how different functions interact with each other. An analysis reasoning about *thrown exceptions*—that flow out of a function—is one such example.

Furthermore, a related categorization is that of a *whole-program* analysis in contrast to a *partial* one. A whole-program analysis examines every part of the program, including any external dependencies that the code may have, and reasons about the effects each part has on the rest of the program. In the setting of Java, for example, a whole-program analysis not only reasons about the application code but additionally about any third-party library used by the program (e.g., from external Java Archives—JARs) and also about code run by the Java Runtime Environment—i.e, library code provided by the language itself. On the other hand, a partial analysis only focuses on specific parts of the program and ignores the effects of the rest (e.g., an analysis focusing on certain Java packages).

A partial analysis usually may afford to implement more complex, more expensive reasoning than a whole-program one, since it only focuses on a very localized part of the program. On the contrary, a whole-program analysis has to constantly balance the complexity of its logic, any potential precision gains but also any scalability penalties. The rest of this dissertation will only focus on a few interesting whole-program analyses.

2.4 Flow Sensitivity

Although counter-intuitive at first, it is not unusual for a static program analysis to be flow-*insensitive*. A flow-sensitive analysis examines a method's instructions while taking into account the order they appear in the source code. On the contrary, a flow-insensitive analysis examines a method's instructions as if they were in a set, without any particular order (i.e., any instruction could happen before any other), and as if they may repeat any

number of times. The latter approach leads to analyses that over-approximate the semantics of the actual code—thus potentially suffering in precision—but it is a common tradeoff when aiming to improve the performance of an analysis.

The penalties on performance for a flow-sensitive analysis mainly stem from the need to keep track of what holds at *every single* program point. Potentially, this could mean that information that remains unchanged will be duplicated on a multitude of instructions. On the contrary, a flow-insensitive analysis will collapse information along all instructions of a method.

For the example on the side, a flow-sensitive analysis reasoning about the values of primitive expressions might report that: after line 1: "`x` has the value `1`", after line 2: "`x` has the value `1`" and "`y` has the value `1`", and after line 3: "`x` has the value `2`" and "`y` has the value `1`".

```
1  x = 1;
2  y = x;
3  x = 2;
```

A flow-insensitive analysis might instead report that: "`x` has either the value `1` or `2`" and "`y` has either the value `1` or `2`", because instructions are examined as if happening in any order.

2.5 Static Single Assignment Form

In compiler design, *static single assignment form* (also known as SSA) is a property of an intermediate representation (IR) of the program in which every local variable is assigned only once. Existing local variables in the original source code are split into *versions* (e.g., variable `x` might be split into `x_1` and `x_2`) with each version being assigned only once. At program points where the value of the original variable is read and there are multiple valid versions, as in the point where the branches of an *if-else* statement merge, a *phi-node* statement is used. This special statement bears the semantics of somehow "choosing" a specific variable version to read.

```
1  if (...) x = 10;
2  else x = 20;
3  y = x;
```

Original source code

```
1  if (...) x_1 = 10;
2  else x_2 = 20;
3  y = phi(x_1, x_2);
```

The equivalent SSA form

Figure 2.2: Example code and equivalent SSA form.

In the context of static program analysis, SSA is often used to approximate the benefits of a flow-sensitive analysis, particularly pertaining to the handling of local variables. This is not the case for other, more complicated language features such as heap accesses and method invocations, but SSA provides an easy way to pick the low-hanging fruit.

The flow-*insensitive* analysis of 2.4 will report quite different results when analyzing the SSA-form analogue of the example code (given on the side): "`x_1` has the value `1`", "`y` has the value `1`", and "`x_2` has the value `2`". The analysis is still examining instructions without taking order into account, and is unable

```
x_1 = 1;
y = x_1;
x_2 = 2;
```

to answer questions such as "what is the value of variable `x` at the end of the method" (whether the value of `x_1` or `x_2` is the final one), but nevertheless it has managed to reclaim some of the precision that was previously lost—e.g., regarding the value of variable `y`.

2.6 May vs. Must Analyses

Given an abstraction of behaviors (e.g., thrown exceptions) one can define two interesting sets regarding the potential behaviors that a given program will exhibit. Set $Any(P)$ is defined as containing all possible behaviors that a program P *will* exhibit in *some* program execution (e.g., method `meth1` will throw exception `e1` in one execution and exception `e2` in another). Respectively, set $All(P)$ is defined as containing behaviors that will appear in *every* program execution (e.g., method `meth2` will always throw exception `e3` during any execution).

$$Any(P) := \bigcup_{e \in Executions} Behaviors(P, e) \quad \text{and} \quad All(P) := \bigcap_{e \in Executions} Behaviors(P, e)$$

Both sets are a mathematical ideal, an answer only an oracle could provide. But, for any realistic analysis such an endeavor is an undecidable problem. Thus, a practical static program analysis will aim to compute an approximation of one of the two sets. Under that definition, analyses are split mainly into two groups depending on the kind of approximation they try to achieve. On one hand, a *may*-analysis is one that aims to *over*-approximate $Any(P)$.

$$May(P) \supseteq Any(P)$$

On the other hand, a *must*-analysis is one that aims to *under*-approximate. In many settings (e.g., when an analysis integrates static and dynamic reasoning as in *dynamic-symbolic execution*), the aim is to under-approximate $Any(P)$. But, this is not always the case. For instance, in the setting of a static pointer or alias analysis, it is more favorable for practical approaches to aim towards an under-approximation of $All(P)$.

$$Any(P) \supseteq Must(P) \quad \text{and} \quad Any(P) \supseteq All(P) \supseteq Must_{PT}(P)^1$$

For the rest of this dissertation, unless otherwise noted, all described analyses aim to compute $May(P)$, i.e., they are may-analyses. This also reflects the fact that may-analyses comprise the norm in related literature.

2.7 Soundness & Completeness

The term *soundness*, and its converse *completeness*, originate from formal mathematical logic where they are used in order to evaluate a *proof* system under a given *model*. The model is

[1] Referring to the common approach of a must-(pointer-/alias-)analysis.

some kind of mathematical structure, such as sets over some domain of interest and the proof system is a set of rules with which *properties* regarding the model are proven. A system is sound if and only if statements it can prove are indeed true in the model (concisely given as "claim implies truth"). A system is complete if and only if what is true in the model can also be proven by the system (concisely given as "truth implies claim").

In the context of static program analysis, the most relevant and widely used term is that of soundness. In this setting, the analysis is making claims regarding potential program behaviors under any program execution and the validity of those claims constitutes whether the analysis is sound or not. More specifically, a may-analysis is sound whenever what it claims is actually true—i.e., the computed behaviors are an over-approximation of $Any(P)$. Respectively, a must-analysis is sound whenever the computed behaviors are an under-approximation of $All(P)$. A trivially sound-may analysis could simply infer top ($\top$), i.e., every possible behavior. A trivially sound-must analysis could simply infer the empty set ($\emptyset$), i.e., no behavior at all.

Contrary to the prevalent use of soundness for evaluating static program analysis algorithms, completeness is scarcely referenced—if ever. One can easily find "proofs of soundness" on many publications but not the analogous "proofs of completeness". This is mainly because of the way analyses (may- or must-) are defined as aiming to compute an *approximation* of potential behaviors. For example, what would the meaning of a complete may-analysis be? Such an analysis has to abide by the definition of "truth implies claim". In this case "truth" is any over-approximation of $Any(P)$. Consequently, a "complete" may-analysis would need to compute all possible over-approximations of $Any(P)$, i.e., for all practial purposes infer top ($\top$), and thus, the term is less relevant in the domain of static program analysis.

Relatedly to soundness, there are two closely related terms characterizing the validity of each claim made by the analysis. If the analysis incorrectly claims that some behavior is among the potential program behaviors, then this constitutes a *false-positive*. E.g., if the analysis claims that method $m1$ might throw an exception $e1$, but there is no program execution under which this will actually occur. Similarly, if an analysis claims that some behavior will never happen but in reality there exists a program execution where such a behavior takes place, then this constitutes a *false-negative*. E.g., if the analysis claims that method $m2$ will never throw an exception, but it actually does. By consequence of previous definitions, a sound may analysis will make no false-negative claims, and a sound must analysis will make no false-positive ones.

It is noteworthy that every static program analysis is also making negative claims, in addition to positive ones, even if only implicitly. This is due to an analysis not making some claim actually implicitly claiming its negation. For example, if a may-analysis for thrown exceptions reports that method $m1$ might throw either exception $e1$ or exception $e2$, then it also implicitly reports that the same method will never throw any other exception—given that the analysis aims to compute an over-approximation of possible behaviors.

2.8 Precision & Recall

Both a may- and a must-analysis operate under the premise of computing an approximation of reality and thus there will always be claims that are either extraneous or missing. Of course, any analysis should do its best to get as close to the truth, even if the actual set of behaviors will always be out of grasp. This calls for some *quantitative* way to measure an analysis's quality. Two such metrics have been proposed. *Precision* indicates how many of the analysis claims are actually part of the truth, whereas *recall* indicates how many of the actual true claims are also reported by the analysis. For a formal definition supposing that:

- X is the number of true (actually happening) interesting behaviors
- $T \leq X$ is the number of correct claims made by the analysis (the *true-positives*)
- F is the number of incorrect claims made by the analysis (the *true-negatives*)

then

$$Precision(P) = \frac{T}{T+F} \quad \text{and} \quad Recall(P) = \frac{T}{X}$$

A value of 1 is a perfect score, and a value of 0 is the worst one. A sound-may analysis will have perfect recall, since it at least claims any behavior that is actually true. A sound-must analysis will have perfect precision, since it makes no incorrect claims (it does not report any behavior that cannot actually occur).

Highly useful as they are, both measures have a major, unavoidable shortcoming. It is rarely the case that there is actual knowledge of the ground-truth. Somewhat a chicken-and-egg problem, having an automatic way to retrieve the ground-truth for arbitrary programs is both what a static analysis aims to achieve at its core and also what is needed to evaluate its claims. Usually, one has to resort to observing a limited amount of actual executions for a given program and—making the assumption that the observations are a good enough representative—interpolate to all possible program executions.

Furthermore, this approach makes both measures *empirical*, in the sense that they measure an analysis's performance in regards to a *specific* given program. They bear no information in regards to the analysis behavior at a theoretical level and cannot be generalized to other programs. An analysis could be perfect for one program and terrible for another. This could be tackled to some extent with the use of well-established *benchmarking suites* during the testing phase, that aim to cover common and interesting code patterns and program behaviors.

2.9 Soundiness

Although soundness seems like an essential property for any static program analysis algorithm to have, and is quite prevalent in academic literature, Livshits et al [94] make a strong claim that there is no practical sound whole-program may-analysis. Most of the time, this

is a conscious compromise on how to handle certain language features and not due to lack of understanding. If one would attempt to soundly model such features in the context of a may-analysis, i.e., over-approximate their effects, this would most probably result in an analysis that is so unscalable or imprecise that is practically *useless*.

At the same time, many academic publications make claims of soundness and may even provide some kind of "proof of soundness" but this is mostly in regards to a *subset* of language features—the analysis might be totally unsound in handling all the rest. Thus, a need arises for a way to differentiate between an analysis that tries its best to be sound and only gives up in a well-defined language subset, and one that is simply unsound.

The term *soundiness*, specific to the context of static program analysis, was coined by Livshits et al [94] for such a purpose. A *soundy* analysis handles most classical, core language features in a sound manner (i.e., over-approximates) and may only fail to do so (i.e., under-approximates) in a small well-accepted subset of highly *dynamic features*, specific to each language. Such features include uses of reflection or native code in Java, `eval` in Javascript, and pointer arithmetic in C/C++.

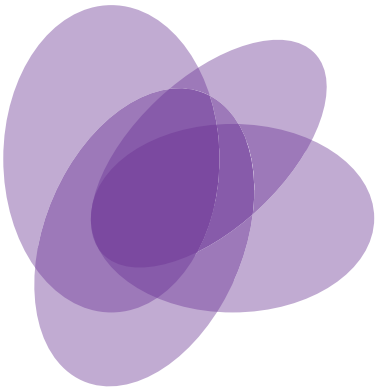
2.10 A Static Program Analysis Mind Map

Figure 2.3 attempts to shed some light on how different kinds of static program analysis relate to each other and to the mathematical ideals. Each set in figure 2.3a represents all behaviors exhibited under a single actual program execution. Any static program analysis will collapse different executions into a single set and will make claims about the given program in general. In figure 2.3b, the ideal $Any(P)$ is represented by the union of all sets and that of $Any(P)$ by the intersection, respectively. Any sound may-analysis will result in a superset of $Any(P)$ (figure 2.3c), whereas any sound must-(pointer-/alias-)analysis (i.e., $Must_{PT}(P)$) will result in a subset of $All(P)$ (figure 2.3d). Finally, any unsound analysis (figure 2.3e) will result in a set with no apparent properties; not entirely covering its appropriate mathematical ideal but also including unrelated elements. More specifically though, as illustrated in figure 2.3f, a soundy analysis will result in a set that starts as a superset of $Any(P)$ but misses some hard (i.e., costly) to over-approximate behaviors in well-known cases.

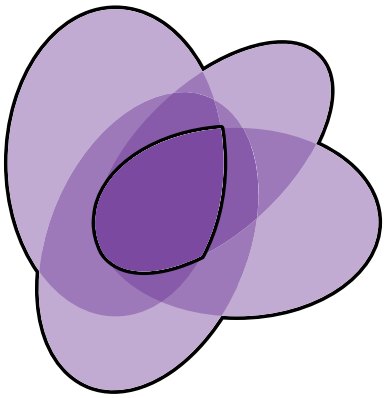
Therefore, the nature of the algorithms described in Chapters 3 and 4 is loosely depicted in Figure 2.3f. The analysis presented in Chapters 5 and 6 falls under what is depicted in Figure 2.3d, whereas the analysis of Chapter 7 follows Figure 2.3c.

2.11 The Doop Framework

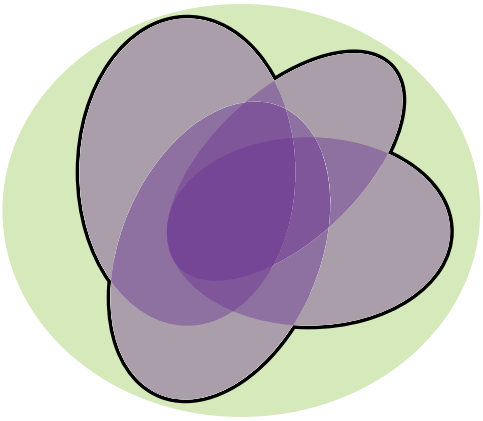
Most of the following work and algorithms have been expressed in the Doop framework [16]. Doop is written in the *declarative* language Datalog, and although Datalog has been used for points-to analyses in the past, this was the first implementation to express full end-to-



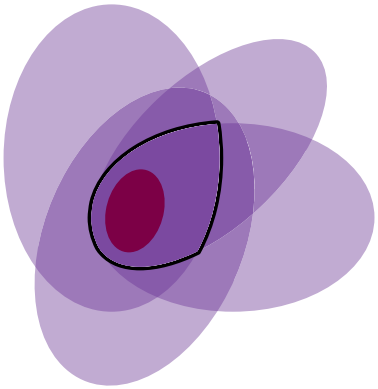
(a) Behavior sets of actual program executions



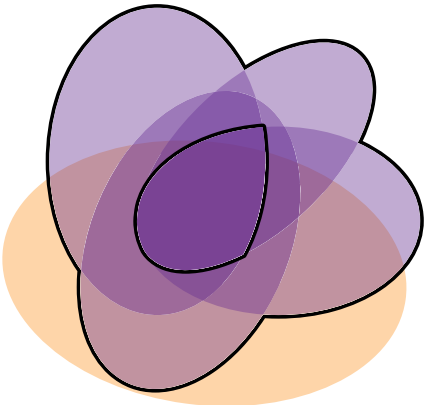
(b) Mathematical ideals $Any(P)$ (union) and $All(P)$ (intersection)



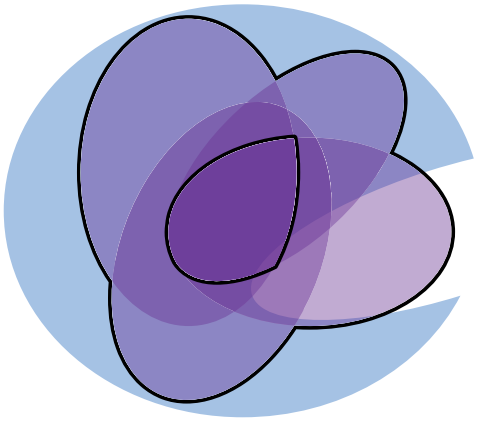
(c) Any sound may-analysis (green) in relation to $Any(P)$



(d) Any sound must-(pointer-/alias-)analysis (red) in relation to $All(P)$



(e) Any unsound analysis (orange) in relation to both ideals



(f) Any *soundy* analysis (blue) in relation to both ideals

Figure 2.3: Venn diagrams visualizing the relations of different static program analysis flavors to each other and to real behaviors.

end context-sensitive analyses for Java[2], declaratively. This includes handling key analysis elements such as call-graph construction as well as logic dealing with various semantic complexities of the Java language such as native code, reflection and exceptions. Nowadays, Doop offers a wide array of intricate analyses displaying a variety of properties.

A Datalog Primer

The declarative power of Doop stems from the expressiveness of Datalog. Datalog has been described in the past, at a higher level, either as Prolog without function symbols or as SQL with support of recursion. Programs written in Datalog are essentially logic specifications that, as a side-effect, are also executable. One simply models the semantics for each language feature of interest along with any reasoning rules of the desired analysis, and the underlying engine is responsible for combining the logic specification with input facts and, after applying a specialized computation, inferring anything that follows given the rules and the respective input.

The aforementioned specialized computation is known as a *fixpoint* computation. All valid Datalog rules are monotonic, i.e., can only reason about the inference of additional facts, and this is exploited by the underlying engine in order to efficiently compute new facts in a repeating fashion until knowledge from previous steps cannot be applied to infer anything new in the current step. Datalog programs are contained in the `PTime` complexity class, i.e., they can be computed in polynomial time, and vice versa, any polynomial algorithm can be implemented as a Datalog program. Additionally, because of the monotonic nature of rules, termination is always guaranteed.[3]

A Datalog program is mainly a collection of rules with each rule contributing to a common knowledge base. Each rule has two parts (separated by "←"); the *rule-body*, on the right, that describes what conditions need to hold in order to infer something new, and the *rule-head*, on the left, that describes what new knowledge is inferred each time. Both parts are collections of *relations* that are conceptually similar to database tables. Relations can be connected to each other either via commas ("`,`"), denoting a logical `AND` connection similar to a database join, or via semicolons ("`;`"), denoting a logical `OR` connection. Finally, relations can be negated by prepending an exclamation mark ("`!`"). Negation is stratified: it is only applied to predicates that are either input predicates or whose computation can complete before the current rule's evaluation. We also permit multiple predicates in a rule head, as syntactic sugar for replicating the rule body.

A classic example is given below. Parent represents the obvious parent relationship between individuals abstracted by the two arguments, whereas Ancestor represents an ancestry relationship of any depth.

[2]More specifically, the soon-to-be presented algorithms operate on Java bytecode—the stack-based intermediate language used in the Java VM—rather than on Java source, and thus they could, in theory, be applied to any programming language that targets the Java VM.

[3]We will later relax these constraints by adding language extensions that will push Datalog programs over the `PTime` class and might invalidate the termination guarantees offered by the language. We will do so in a principled way, allowing for greater flexibility on the algorithms that we can express.

```
Ancestor(x, y) ⟵ Parent(x, y).

Ancestor(x, y) ⟵ Parent(x, z), Ancestor(z, y).
```

The first rule simply states that any parent relationship is also an ancestry one. The second rule is where recursion, the true power of Datalog, shines through. It states that whenever some `x` is the parent of some `z`, and it is already known that `z` is an ancestor of some `y`, then it is also valid to infer that `x` is transitively an ancestor of `y`. This second rule is where the fixpoint computation comes into play. Depending on the input facts, the underlying engine might need to apply the rule multiply times, each time inferring new facts that will in turn be used as the base for further inference.

2.12 Modeling Points-To Analyses: Parameterizable Model

We model a spectrum of flow-insensitive (recall Section 2.4), context-sensitive points-to analyses and joint (also known as *on-the-fly* or *online*) call-graph construction as a parametric Datalog program. Rules in a Datalog program are monotonic logical inferences that repeatedly apply to infer more facts until fixpoint. Our rules do not use negation in a recursive cycle, or other non-monotonic logic constructs, resulting in a declarative specification: the order of evaluation of rules cannot affect the final result. The same abstract model applies to a wealth of analyses. We use it to model a context-insensitive Andersen-style [4] analysis, as well as several context-sensitive analyses, both call-site-sensitive and object-sensitive ones.

The input language is a representative simplified intermediate language that models well the Java bytecode representation[4], but also other high-level intermediate languages. It does not, however, model languages such as C or C++ that can create pointers through an address-of operator. The techniques used in that space are fairly different—e.g., [56, 57]—although our main hybrid approach is likely to be applicable there as well. Also, even though we model regular object fields and static methods, we omit static fields, arrays, exceptions, etc. Their treatment is a mere engineering complexity, as it does not interact with context choice, and indeed the actual implementations in the Doop framework do cover all the intricacies of Java bytecode.

The domain of our intermediate language (i.e., the different value sets that constitute the space of our computation) is presented in Figure 2.4 and the core instructions are abstracted away in the first six Datalog input relations presented in Figure 2.5. We explain the contents of our input relations in more detail below:

- `Alloc` represents instructions for allocating an object `heap`[5] on the heap, assigning it to variable `var` of method `inMeth`. We abstract heap objects by their allocation sites

[4]The Java bytecode is a stack-based intermediate language used in the Java VM, for reasons of compactness. For analysis purposes, however, it is common to translate it into equivalent but more conventional notations, such as the Jimple intermediate language of the Soot framework [158, 159]. Therefore, it might be more accurate to say that out intermediate language is a simplified form of Jimple, rather than a simplified form of the Java bytecode.

[5]For brevity reasons, we will frequently refer to an abstract object allocated on the heap, simply as *heap*.

V is a set of program variables
H is a set of heap abstractions (i.e., allocation sites)
M is a set of method identifiers
S is a set of method signatures (including name and type signature)
F is a set of fields
I is a set of instructions (mainly used for invocation sites)
T is a set of class types
$\mathbb{N}$ is the set of natural numbers
C is a set of contexts
HC is a set of heap contexts

Figure 2.4: The domain of our input intermediate language.

throughout the text, and for simplicity reasons, we name them just by using "heap". Additionally, note that every local variable is appropriately annotated by the exact method signature in which it is defined and is thus uniquely identified just by its name.

- Move represents instructions for copying values between local variables to and from.
- Load represents instructions for reading from the heap, i.e., from field fld of the object stored in variable base and assigning the value back to variable to, whereas Store represents instructions for the inverse flow (i.e., writing the value of variable from to the field fld of the object in variable base).
- VCall represents instructions that call the method of the appropriate signature sig that is defined in the dynamic class of the receiver object stored in local variable base whereas SCall represents instructions that call a statically known target method identified by signature sig. Both VCall and SCall supplementary report the invocation instruction invo itself as well as the enclosing method inMeth.

The last seven Datalog relations encode pertinent symbol table information, that will prove helpful in the analysis rules to come.

- FormalArg states that the i-th argument of method meth is the local variable arg. ActualArg does the same for invocations.
- FormalReturn and ActualReturn convey similar information for when a method returns to its caller.
- ThisVar represents the special local variable this—when applicable—inside method meth whereas HeapType maps object heap to its actual dynamic type.
- LookUp simulates the lookup operations inside a Java VM that given the dynamic type type of a receiver object and a method signature sig, find the appropriate target method meth to call in a virtual invocation.

```
Alloc(var: V, heap: H, inMeth: M)          // var = new ...
Move(to: V, from: V)                       // to = from
Load(to: V, base: V, fld: F)               // to = base.fld
Store(base: V, fld: F, from: V)            // base.fld = from
VCall(base: V, sig: S, invo: I, inMeth: M) // base.sig(...)
SCall(meth: M, invo: I, inMeth: M)         // Class.meth(...)

FormalArg(meth: M, i: ℕ, arg: V)
ActualArg(invo: I, i: ℕ, arg: V)
FormalReturn(meth: M, ret: V)
ActualReturn(invo: I, var: V)
ThisVar(meth: M, this: V)
HeapType(heap: H, type: T)
LookUp(type: T, sig: S, meth: M)
```

Figure 2.5: The input Datalog relations describing the program under analysis.

The specification of our points-to analysis as well as the input language are in line with those in past literature [50, 100], although we also integrate elements such as on-the-fly call-graph construction, static calls, and field sensitivity. Specifying the analysis logically as Datalog rules has the advantage that the specification is close to the actual implementation. Datalog has been the basis of several implementations of program analyses, both low-level [16, 78, 122, 163, 164] and high-level [38, 52]. Indeed, the analysis we show is a faithful model of the implementation in the Doop framework, upon which our work builds. Our specification of the analysis (Figures 2.6-2.7) is an abstraction of the actual implementation in the following ways:

- The implementation has many more rules. It covers the full complexity of Java, including rules for handling reflection, native methods, static fields, string constants, implicit initialization, threads, and a lot more. The Doop implementation[6] currently contains over 1200 rules in the common core of all analyses, and several more rules specific to each analysis, as opposed to the 9 rules we examine here. (Note, however, that these few rules are the most crucial for any points-to analysis. They also correspond fairly closely to the algorithms specified in other formalizations of points-to analyses in the literature [103, 142].)

- The implementation also reflects considerations for efficient execution. The most important are those of defining indexes for the key output relations, or providing multiple execution plans for key rules, all depending on the underlying Datalog engine. Furthermore, implementation details might include designating some relations as functions, defining storage models for relations (e.g., how many bits each variable uses), designating intermediate relations as "materialized views" or not, etc. No such considerations are reflected in our model.

```
VarPointsTo(var: V, ctx: C, heap: H, hctx: HC)
CallGraphEdge(invo: I, callerCtx: C, toMeth: M, calleeCtx: C)
FldPointsTo(baseH: H, baseHCtx: HC, fld: F, heap: H, hctx: HC)
InterProcAssign(to: V, toCtx: C, from: V, fromCtx: C)
Reachable(meth: M, ctx: C)
```

```
Record(heap: H, ctx: C) = ?newHCtx: HC
Merge(heap: H, hctx: HC, invo: I, ctx: C) = ?newCtx: C
MergeStatic(invo: I, ctx: C) = ?newCtx: C
```

Figure 2.6: The core Datalog output relations and constructors of contexts.

Figure 2.6 shows the intermediate and output relations, as well as three *constructor* functions, responsible for producing new contexts. Figure 2.7 shows the points-to analysis and call-graph computation. We explain the contents of both figures in more detail below:

- There are five output or intermediate computed relations (VarPointsTo, . . ., Reachable[7]). Every occurrence of a method or local variable in computed relations is qualified with a context (i.e., an element of set C), while every occurrence of a heap object is qualified with a heap context (i.e., an element of HC). The main output relations are VarPointsTo and , encoding our points-to and call-graph results. The VarPointsTo relation links a variable (var) to a heap object (heap). Other intermediate relations (FldPointsTo, InterProcAssign, Reachable) correspond to standard concepts and are introduced for conciseness. For instance, InterProcAssign (which encodes all parameter and return value passing) unifies much of the treatment of static and virtual method calls.

- We model the parameter space of context-sensitive points-to analysis in a way that allows for entirely different flavors of algorithms. The base rules are not concerned with what kind of context sensitivity is used. The same rules can be used for a context-insensitive analysis (by only ever creating a single context object and a single heap context object), for a call-site-sensitive analysis, or for an object-sensitive analysis, for any context depth. These aspects are completely hidden behind constructor functions Record, Merge, and MergeStatic. The first two follow the usage and naming convention of Smaragdakis et al. [142], while MergeStatic is new and used to differentiate the treatment of static calls—this is a crucial element of our approach.

- Record is the function that creates a new heap context. It is invoked whenever an object allocation site (input relation Alloc) is analyzed. Thus, Record is only used in the rule treating allocation instructions (3rd rule in Figure 2.7). Record takes all available information at the allocation site of an object and combines it to produce a

[6] Doop is publicly available online at `https://bitbucket.org/yanniss/doop/`.

[7] Reachable is somewhat of a special case, since we assume it is also used as an input relation: it needs to initially hold methods that are always reachable, such as the programs's `main` method, the constructor of class `java.lang.ClassLoader`, and more. We ignore this technicality in the model, rather than burden our rules with a separate input relation.

```
InterProcAssign(to, calleeCtx, from, callerCtx) ←
    CallGraphEdge(invo, callerCtx, toMeth, calleeCtx),
    FormalArg(toMeth, i, to), ActualArg(invo, i, from).

InterProcAssign(to, callerCtx, from, calleeCtx) ←
    CallGraphEdge(invo, callerCtx, toMeth, calleeCtx),
    FormalReturn(toMeth, from), ActualReturn(invo, to).
```

```
Record(heap, ctx) = ?hctx,
VarPointsTo(var, ctx, heap, ?hctx) ←
    Reachable(meth, ctx), Alloc(var, heap, meth).

VarPointsTo(to, ctx, heap, hctx) ←
    Move(to, from), VarPointsTo(from, ctx, heap, hctx).

VarPointsTo(to, toCtx, heap, hctx) ←
    InterProcAssign(to, toCtx, from, fromCtx),
    VarPointsTo(from, fromCtx, heap, hctx).

VarPointsTo(to, ctx, heap, hctx) ←
    Load(to, base, fld), VarPointsTo(base, ctx, baseH, baseHCtx),
    FldPointsTo(baseH, baseHCtx, fld, heap, hctx).

FldPointsTo(baseH, baseHCtx, fld, heap, hctx) ←
    Store(base, fld, from), VarPointsTo(from, ctx, heap, hctx),
    VarPointsTo(base, ctx, baseH, baseHCtx).
```

```
Merge(heap, hctx, invo, callerCtx) = ?calleeCtx,
Reachable(toMeth, ?calleeCtx),
VarPointsTo(this, ?calleeCtx, heap, hctx),
CallGraphEdge(invo, callerCtx, toMeth, ?calleeCtx) ←
    VCall(base, sig, invo, inMeth),
    Reachable(inMeth, callerCtx),
    VarPointsTo(base, callerCtx, heap, hctx),
    HeapType(heap, heapT),
    Lookup(heapT, sig, toMeth),
    ThisVar(toMeth, this).

MergeStatic(invo, callerCtx) = ?calleeCtx,
Reachable(toMeth, ?calleeCtx),
(invo, callerCtx, toMeth, ?calleeCtx) ←
    SCall(toMeth, invo, inMeth),
    Reachable(inMeth, callerCtx).
```

Figure 2.7: Datalog rules for the points-to analysis and call-graph construction.

new heap context. The rule merely says that an allocation instruction in a reachable method leads us to infer a points-to fact between the allocated object and the variable it is directly assigned to. We denote variables created by a constructor function with a question mark at the beginning of their name for emphasis (e.g., `?hctx`).

- `Merge` and `MergeStatic` are used to create new calling contexts (or just "contexts"). These contexts are used to qualify method calls, i.e., they are applied to all local variables in a program. The `Merge` and `MergeStatic` functions take all available information at the call-site of a method (virtual or static) and combine it to create a new context. These functions are sufficient for modeling a very large variety of context-sensitive analyses, as we show in Section 2.13 (and later in Section 3.1).

  Note that the use of constructors, such as `Record`, `Merge`, and `MergeStatic`, is not part of regular Datalog and can result in infinite structures (e.g., one can express unbounded call-site sensitivity) if care is not taken. All our later definitions statically guarantee to create contexts of a pre-set depth. Also noteworthy is the fact that, each different combination of parameters in a constructor will only create a new context if one does not already exist—otherwise it just returns the pre-existing one.

- The rules of Figure 2.7 show how each input instruction leads to the inference of facts for the five output or intermediate relations. The most complex rule is the second-to-last, which handles virtual method calls (input relation `VCall`). The rule says that if a reachable method of the program has an instruction making a virtual method call over local variable `base` (this is an input fact), and the analysis so far has established that `base` can point to heap object `heap`, then the called method is looked up inside the type of `heap` and several further facts are inferred: that the looked up method is reachable, that it has an edge in the call-graph from the current invocation site, and that its `this` variable can point to `heap`. Additionally, the `Merge` function is used to possibly create (or look up) the right context for the current invocation.

2.13 Standard Points-To Analyses: Instantiating the Model

By modifying the definitions of the `Record`, `Merge` and `MergeStatic` functions as well as domains `HC` and `C`, one can create endless variations of points-to analyses. We next discuss the most interesting combinations from past literature, before we introduce our own (in Chapter 3). For every analysis variation we also list a common name abbreviation, which we often use later.

Context-insensitive (insens). As already mentioned, our context-sensitive analysis framework can yield a context-insensitive analysis by merely picking singleton `C` and `HC` sets (i.e., `C` = `HC` = `{⋆}`, where ⋆ is merely a name for a distinguished element) and constructor functions that return the single element of the set:

```
Record(heap, ctx) = ⋆
```

```
Merge(heap, hctx, invo, ctx) = ⋆
MergeStatic(invo, ctx) = ⋆
```

Note that the absence of contexts does not mean that the identity of input elements is forgotten. Objects are still represented by their allocation site (i.e., the exact program instruction that allocated the object) and local variables are still distinguished (e.g., by their declaration location in the input program). The absence of context just means that there is no *extra* distinguishing information. This can also be seen in the rules of Figure 2.7, where the `var` and `heap` predicate arguments are present, separately from the context arguments.

1-call-site-sensitive (1call). A 1-call-site-sensitive analysis has no heap context to qualify heap abstractions (`HC` = `{⋆}`) and uses the current invocation site as a context (`C` = `I`). The following definitions describe such an analysis.

```
Record(heap, ctx) = ⋆
Merge(heap, hctx, invo, ctx) = invo
MergeStatic(invo, ctx) = invo
```

In words: the analysis stores no context when an object is created (`Record`) and keeps the invocation site as context in both virtual and static calls.

1-call-site-sensitive with a context-sensitive heap (1call+H). The analysis is defined similarly to 1call.[8] The heap context as well as the main context consist of an invocation site (`HC` = `C` = `I`).

```
Record(heap, ctx) = ctx
Merge(heap, hctx, invo, ctx) = invo
MergeStatic(invo, ctx) = invo
```

In words: the analysis uses the current method's context as a heap context for objects allocated inside the method. The invocation site of a method call is the context of the method for both virtual and static calls.

1-object-sensitive (1obj). Object sensitivity uses allocation sites as context components. A 1-object-sensitive analysis has no heap context (`HC` = `{⋆}`) and uses the allocation site of the receiver object as context (`C` = `H`). The following definitions complete the description.

```
Record(heap, ctx) = ⋆
Merge(heap, hctx, invo, ctx) = heap
MergeStatic(invo, ctx) = ctx
```

[8]The standard convention in the points-to analysis literature is to name an analysis first according to the context of methods, and, if a heap context exists, designate it in a suffix such as *context-sensitive heap* or *heap cloning*.

In words: the analysis stores no context for allocated objects. For virtual method calls, the context is the allocation site of the receiver object. For static method calls, the context for the called method is that of the calling method.

The above definition offers a first glimpse of the possibilities that we explore in this paper, and can serve as motivation. In static calls, the context of the caller method is copied, i.e., the receiver object of the caller method is used as the new context. Why not try `MergeStatic(invo, ctx) = invo`, instead of the current `MergeStatic(invo, ctx) = ctx`? Isn't it perhaps better to use call-sites to differentiate static invocations, instead of blindly copying the context of the last non-static method called? A simple answer is that `invo` is an entity of the wrong type, since `C` = `H`. The only entity of type `H` we have available at a static call-site is the current context, `ctx`. But if we let `C` = `H` ∪ `I`, we have a context type that is a hybrid of both an allocation site and an invocation site, and which allows the above alternative definition of `MergeStatic`. We explore this and other such directions in depth in Chapter 3.

2-object-sensitive with a 1-context-sensitive heap (2obj+H). In this case, the heap context consists of one allocation site (`HC` = `H`) and the context consists of two allocation sites (`C` = `H` × `H`). The definitions of constructor functions are:[9]

```
Record(heap, ctx) = first(ctx)
Merge(heap, hctx, invo, ctx) = pair(heap, hctx)
MergeStatic(invo, ctx) = ctx
```

In words: the context of a virtual method (see `Merge`) is a 2-element list consisting of the receiver object and its (heap) context. The heap context of an object (fixed at allocation, via `Record`) is the first context element of the allocating method, i.e., the receiver object on which it was invoked. Therefore, the context of a virtual method is the receiver object together with the "parent" receiver object (the receiver object of the method that allocated the receiver object of the virtual call). Again, static calls just copy the context of the caller method.

Although there can be other definitions of the `Merge` function, yielding alternative 2-obj+H analyses, it has been shown [142] that the above is the most precise and scalable. In intuitive terms, we use as method context the most precise abstraction of the receiver object available to the analysis.

2-type-sensitive with a 1-context-sensitive heap (2type+H). A type-sensitive analysis is step-by-step analogous to an object-sensitive one, but instead of using allocation sites

[9]We use auxiliary constructor functions like `pair`, `triple` and accessors like `first`, `second`, etc., with the expected meaning, in order to construct and deconstruct contexts with 2 or 3 elements. This has the added advantage that our context-depth is statically bounded—we never create lists of unknown length. Since our most complex constructor is `triple`, the possible number of distinct contexts is cubic in the size of the input program.

(i.e., instructions) a type-sensitive analysis uses the name of the class containing the allocation site. In this way, all allocation sites in methods declared in the same class (though not inherited methods) are merged. This approximation was introduced by Smaragdakis et al. [142] and yields much more scalable analyses at the expense of moderate precision loss (as we also determine in our experiments).

In order to define type-sensitive analyses we need an auxiliary function which maps each heap abstraction to the class containing the allocation.

$\mathcal{C}_{\mathcal{A}} : H \rightarrow T$

Now we can define a 2type+H analysis by mapping $\mathcal{C}_{\mathcal{A}}$ over the context of a 2obj+H analysis. The heap context uses a type instead of an allocation site ($\texttt{HC} = \texttt{T}$) and the calling context uses two types ($\texttt{C} = \texttt{T} \times \texttt{T}$).

```
Record(heap, ctx) = first(ctx)
Merge(heap, hctx, invo, ctx) = pair(C_A(heap), hctx)
MergeStatic(invo, ctx) = ctx
```

But just having a type as a context element does not tell us how good the context will be in improving precision. Thus, the selection of type is of paramount importance. As discussed in [142] using the dynamic type of the heap object would be an awful design decision—the method under analysis already gives us enough information about the type of the receiver object. A better approach is to use an upper bound of the dynamic type of the *allocator* object.[10]

Other Analyses. The above discussion omits several analyses in the literature, in order to focus on a manageable set with practical relevance. We do not discuss a 1-object-sensitive analysis with a context-sensitive heap (1obj+H) because it is a strictly inferior choice to other analyses (especially 2type+H) in practice: it is both much less precise and much slower. We do not present other varieties of type-sensitivity for a similar reason. Deeper contexts or heap contexts (e.g., 2call+H, 2obj+2H, 3obj, etc.) quickly make an analysis intractable for a substantial portion of realistic programs and modern JDKs. In short, we focus on the specific analyses (1call, 1call+H, 1obj, 2obj+H, 2type+H) that are of most practical interest: they are quite scalable over a variety of medium-to-large programs, and no other analysis supplants them by being uniformly better in both precision and performance.

[10] If the allocation occurs in a method of class C, the allocator object must be of type C or a subclass of C that does not override the method containing the allocation site.

Part I

Achieving Scalability

3. HYBRID-CONTEXT SENSITIVITY

> If you wanted to know about the Hybrid, why didn't you just ask me?
>
> *The 12th Doctor* - Doctor Who

Although in principle call-site and object sensitivity are incomparable, in practice the story is quite clear. Call-site sensitivity has a long history, dating back to at least the '80s. For a long time, call-site sensitivity was considered synonymous with context sensitivity as a whole. Object sensitivity was later introduced in 2002 [106] and within a decade it has become the overwhelming choice of context sensitivity for object-oriented programs. Multiple studies [16, 81, 84, 90, 108] have found object-sensitive analyses to yield excellent precision/cost tradeoffs. Compared to call-site-sensitive analyses, an object-sensitive analysis of the same context depth has always been advantageous, in terms of both speed and performance.

Given such past experimental results, it would seem that exploring combinations of call-site and object sensitivity is futile. There is no tradeoff to exploit. Even though call-site-sensitive analyses are occasionally faster to execute, this only comes at the expense of precision. To achieve the same precision level as an object-sensitive analysis, call-site-sensitive analyses have to suffer much higher cost. Additionally, coarser approximations of object sensitivity, such as type sensitivity [142] have filled the performance gap and offered fast options for cases when a full object-sensitive analysis is too expensive.

The question behind this chapter is whether the two kinds of context can be fruitfully combined, given how dissimilar they are. In order to address this question, we map the design space of *hybrid* call-site- and object-sensitive analyses and describe the combinations that arise. Our work shows that the aforementioned conventional wisdom is false. This is one of the rare occasions when the combination of two ideas supplants both: hybrid-context sensitivity outperforms both object and call-site sensitivity in both precision and speed.

Naive hybrid combinations, such as always maintaining as context *both* call and allocation sites, do not pay off, due to extremely high cost. For instance, keeping one call site and one allocation site as context, in all places, yields a very expensive analysis, on average 3.9x slower than a simple 1-object-sensitive analysis. Although such a combination will improve precision, it is still lacking in comparison to, for example, a 2-object-sensitive analysis.

However, we find that more sophisticated hybrids are highly beneficial. Specifically, we show that we can switch per-language-feature between a combined context and an object-only context. For instance, contexts for static method calls are computed differently from contexts for dynamic method calls. This approach yields analyses with both low cost and high precision. Furthermore, adapting contexts per program feature defines a complex design space and allows even further optimization. Design choices arise, such as, how should the context adapt when a dynamic method call, or an object allocation are made inside a static method?

The end result is analyses that closely track the precision of a combined call-site-and-object

sensitivity while incurring none of the cost. In fact, the cost of the resulting analysis is usually *less* (and occasionally much less) than that of just an object-sensitive analysis, due to increased precision. This effect is shown to apply widely, to several variants of analyses. Accordingly, this outcome establishes new sweet spots for the analyses most relevant for practical applications: 1-object-sensitive, 2-object-sensitive with a 1-context-sensitive heap, and analogous type-sensitive [142] analyses. For all of them, a selective hybrid context is typically both more precise and faster than the original analysis.

In all, this chapter describes the following contributions:

- We introduce the idea of hybrid call-site and object sensitivity where the two kinds of context are freely mixed and the mix is adjusted in response to analyzing different program features. The goal is to achieve the precision of keeping *both* kinds of context together, but at the same cost as keeping only one.

- We implement our approach in the Doop framework and apply it to a large variety of algorithms with varying context depth.

- We show experimentally, over large Java benchmarks and the Java JDK, that hybrid-context sensitivity works remarkably well. Our experiments establish that different programming language constructs are best analyzed with different kinds of context. The selective application of a combined context achieves the same *effective* precision as keeping both contexts at all times, at a fraction of the cost, and is typically faster even than keeping only an object context. For instance, in the practically important case of a 2-object-sensitive analysis with a context-sensitive heap, we get an average speedup of **1.53x** *and* a more precise analysis. Similarly, for the simple and popular 1-object-sensitive analysis, we get an average speedup of **1.12x** combined with significant increase in precision.

3.1 Hybrid-Context-Sensitive Analyses

We can now explore interesting combinations of call-site and object sensitivity. The design space is large and we will be selective in our presentation and later experiments. Our choice of analyses in this space leverages insights from past studies on what kinds of context are beneficial.[1] Such insights include:

- A call-site-sensitive heap is far less attractive than an object-sensitive heap. Generally, adding a heap context to a call-site-sensitive analysis increases precision very slightly, compared to the overwhelming cost.

[1]We have validated these insights with extensive measurements on our experimental setup, and have generally explored a much larger portion of the design space than is possible to present in our evaluation section.

- When there is a choice between keeping an object-context or a call-site-context, the former is typically more profitable. This is well validated in extensive past measurements by Lhoták and Hendren [84], comparing call-site-sensitive and object-sensitive analyses of various depths. In other words, call-site sensitivity is best added as *extra* context over an object-sensitive analysis and will almost never pay off as a replacement context, for an object-oriented language.

3.1.1 Uniform Hybrid Analyses

The first kind of context combination is a straightforward one: both kinds of context are kept. We term such combinations *uniform hybrid analyses*. In the variants we describe, a uniform hybrid analysis is guaranteed to be more precise[2] than the base analysis being enhanced. The question is whether such precision will justify the added cost.

The insight is that by using every information available at each point we can get more precise analyses. Although that is proved to be true, experimental results also show that this precision gain comes with an infeasibly high cost in most cases. Still, we present some analyses that fully combine call-site and object sensitivity as a middle step for understanding later improvements and also as a baseline when comparing more elaborate ways of context combination.

Uniform 1-object-sensitive hybrid (U-1obj). Enhancing a 1-object-sensitive analysis with call-site sensitivity results in an analysis with an empty heap context ($\texttt{HC} = \{\star\}$) but with a context that consists of both the allocation site of the receiver object and the invocation site of the method ($\texttt{C} = \texttt{H} \times \texttt{I}$). The following definitions describe the analysis:

```
Record(heap, ctx) = ⋆
Merge(heap, hctx, invo, ctx) = pair(heap, invo)
MergeStatic(invo, ctx) = pair(first(ctx), invo)
```

In words: a virtual method has as context the abstraction of its receiver object, extended with the method's invocation site. A static method keeps a context consisting of the most significant part of the caller's context and the method's invocation site. Note that under the above definitions, the context of a U-1obj analysis is always a superset of that of 1obj, hence the analysis is strictly more precise.

Uniform 2-object-sensitive with 1-context-sensitive heap hybrid (U-2obj+H). A 2-object-sensitive analysis with a context-sensitive heap can be enhanced in the same way. A heap context consists of an allocation site ($\texttt{HC} = \texttt{H}$) and a method context consists of two allocation sites and one invocation site ($\texttt{C} = \texttt{H} \times \texttt{H} \times \texttt{I}$). The constructor definitions for the analysis are:

[2] We use the term "more precise" colloquially. Strictly speaking, the analysis is guaranteed to be "at least as precise" and not necessarily "more".

```
Record(heap, ctx) = first(ctx)
Merge(heap, hctx, invo, ctx) = triple(heap, hctx, invo)
MergeStatic(invo, ctx) = triple(first(ctx), second(ctx), invo)
```

In words: an object's heap context is the receiver object of the method doing the allocation. A virtual method's context is its receiver object's allocation site and context (the latter being the allocation site of the object that allocated the receiver), followed by the invocation site of the method. On a static call, the heap part (i.e., first two elements) of the method context is kept unchanged, and extended with the invocation site of the call.

This analysis is also strictly more precise than the "plain" analysis it is based on, 2obj+H. This is achieved partly by placing the receiver object's allocation site in the most significant position of the context triple. In this way, the `Record` function produces the same heap context as 2obj+H on an object's allocation. Alternative definitions are possible for the same sets of contexts, `C` and `HC`. For instance, one could choose to place `hctx` in the most significant position. Similarly, one could produce a hybrid analysis based on 2obj+H but with a different kind of heap context, e.g., `HC` = `I`, therefore using the invocation site in a method's context as an allocation context. These definitions make decisively less sense, however, per the insights mentioned earlier: invocation sites are rarely advantageous as heap contexts, and, similarly, it is not reasonable to invert the natural significance order of `heap` vs. `hctx`. (We have also verified experimentally that such combinations yield bad analyses.)

Note here that in deeper analyses where some elements from the context are used in order to create new heap contexts, it is important which ones we choose to propagate. Different choices might result in analyses that behave quite differently both in terms of performance and precision. We can easily influence which context elements are selected for propagation, by changing their ordering in the context.

Uniform 2-type-sensitive with 1-context-sensitive heap hybrid (U-2type+H). Isomorphically to object sensitivity, we can enhance type-sensitive analyses with call-site information in the same way. When applied to a 2-type-sensitive analysis with a context-sensitive heap, this results in an analysis with a heap context of one type (`HC` = `T`) and a context of two types and an invocation site (`C` = `T` × `T` × `I`)—mirroring the 2-object-sensitive analysis with a context sensitive heap. The definitions are almost identical:

```
Record(heap, ctx) = first(ctx)
Merge(heap, hctx, invo, ctx) = triple(C_A(heap), hctx, invo)
MergeStatic(invo, ctx) = triple(first(ctx), second(ctx), invo)
```

3.1.2 Selective Hybrid Analyses

Another approach to hybrid call-site- and object-sensitive analyses is to maintain a context that varies inside the same analysis. We call such analyses *selective hybrid analyses*, as opposed to the earlier "uniform hybrid" ones. In a selective hybrid analysis, the sets of

contexts, C and HC, will be formed as the cartesian product of unions of sets. Depending on the information available at different analysis points where new contexts are formed, we shall create contexts of a different kind, instead of always keeping a combination of rigid form. We have already hinted at such opportunities in Section 2.13: at a static method call, an object-sensitive analysis does not have a heap object available to create a new context, hence it can at best propagate the context of the caller. Yet, an invocation site is available and can be used to distinguish different static calls, as long as we are allowed to use it as context. This observation generalizes: static invocations are a language feature that benefits highly from the presence of call-site-sensitive elements in the context. This is not hard to explain: For object-sensitive analyses, when analyzing a static invocation, we do not have much information to use in creating a new context, in contrast to a "normal" virtual invocation. Consequently, it is beneficial to be able to use the invocation site as a differentiator of static calls.

Selective hybrid analyses are among the most interesting parts of our work and, to our knowledge, have never before arisen in the literature, far less specified, implemented, and evaluated.

Selective 1-object-sensitive hybrid A (S_A-1obj). Trying to selectively enhance a 1-object-sensitive analysis (HC = {⋆}) with call-site sensitive elements, we are presented with two options, relative to how contexts are created in static invocations. The first option is quite simple: we can keep only a single context element in both virtual and static invocations. Consequently, in virtual invocations the context will be an allocation site, but in static invocations it will be an invocation site (C = H ∪ I). The definitions needed are the following:

```
Record(heap, ctx) = ⋆
Merge(heap, hctx, invo, ctx) = heap
MergeStatic(invo, ctx) = invo
```

Note that this analysis is *not* guaranteed to be more precise than the 1obj analysis it is based on. Nevertheless, it should be an excellent reference point for comparison and insights: it will suggest how much precision can be gained or lost by call-site sensitivity as a replacement of object sensitivity in static method calls.

Selective 1-object-sensitive hybrid B (S_B-1obj). The second option for a selective hybrid enhancement of a 1-object-sensitive analysis is to add extra information to the context of static calls. This means that context in virtual invocations is still an allocation site, but context in static invocations now consists of both the allocation site copied from the caller *and* the invocation site. In this way, C = H × (I ∪ {⋆}). That is, the context can be either just an allocation site or an allocation site and an invocation site. (This could also be written equivalently as C = H ∪ (H × I), but the earlier form streamlines the definitions of constructors, as it makes all contexts be pairs, thus avoiding case-based definitions.) In this way the constructor definitions become:

```
Record(heap, ctx) = ⋆
Merge(heap, hctx, invo, ctx) = pair(heap, ⋆)
MergeStatic(invo, ctx) = pair(first(ctx), invo)
```

This analysis has a context that is always a superset of the 1obj context and, therefore, is guaranteed to be more precise.

Selective 2-object-sensitive with 1-context-sensitive heap hybrid (S-2obj+H). When dealing with deeper analyses, the possible design decisions start to vary. For example, for a 2-object-sensitive analysis with a context-sensitive heap, an interesting choice is to have allocation sites as heap contexts (HC = H), and for method contexts to keep standard object-sensitive information for virtual calls but favor call-site sensitivity for static calls. The constructor definitions for the above analysis are:

```
Record(heap, ctx) = first(ctx)
Merge(heap, hctx, invo, ctx) = triple(heap, hctx, ⋆)
MergeStatic(invo, ctx) = triple(first(ctx), invo, second(ctx))
```

(In this way, we have C = H × (H ∪ I) × (H ∪ I ∪ {⋆}).) Note the interesting behavior of such an analysis: for virtual calls, the context is equivalent to that of 2obj+H. For the first static call (i.e., from inside a virtually called method), the context is a superset of 2obj+H, augmented by an invocation site. For further static calls (i.e., static calls inside statically called methods), however, the analysis favors call-site sensitivity (both the last two elements of context are invocation sites) and otherwise only remembers the most-significant element of the object-sensitive context. (The latter is important for creating high-quality heap contexts, when allocating objects.) It is interesting to see how this analysis fares relative to 2obj+H, since the analyses are in principle incomparable in precision.

Selective 2-type-sensitive with 1-context-sensitive heap hybrid (S-2type+H). Finally, type-sensitive analyses can be enhanced with call-site sensitive information in much the same way. Mirroring our choices in S-2obj+H, the S-2type+H analysis has heap context HC = T and method context C = T × (T ∪ I) × (T ∪ I ∪ {⋆}). The constructor definitions are isomorphic to the S-2obj+H analysis:

```
Record(heap, ctx) = first(ctx)
Merge(heap, hctx, invo, ctx) = triple(C_A(heap), hctx, ⋆)
MergeStatic(invo, ctx) = triple(first(ctx), invo, second(ctx))
```

Other analyses. The above discussion does not nearly exhaust the space of hybrid combinations. Consider selective hybrids for a 2obj+H analysis: Many more design choices are possible than the one shown. One could change the heap context into an invocation site, or into a union of invocation and call-site (HC = H ∪ I). This combination is a bad choice,

due to the poor payoff of call-site heap contexts. One could create context structures that let call-site- and object-sensitive context elements freely merge, e.g., $\mathtt{C} = (\mathtt{H} \cup \mathtt{I}) \times (\mathtt{H} \cup \mathtt{I}) \times (\mathtt{H} \cup \mathtt{I} \cup \{\star\})$. This allows several different definitions of context constructors, but has the drawback of diverging significantly from object sensitivity (i.e., allowing to skip even the most-significant, object-sensitive context element), which misses the well documented precision and performance advantages of object sensitivity, especially as a heap context.

3.2 Evaluation

We implemented and evaluated all aforementioned analyses using the Doop framework. There are interesting and subtle aspects in our measurements, but the executive summary is clear: uniform hybrid analyses are typically not good choices in practice: their precision is offset by a very high performance cost. A relative exception is the uniform type-sensitive hybrid analysis, U-2type+H, which, although higher-cost, is not prohibitively expensive and offers a reasonable precision/performance tradeoff. Selective hybrid analyses, on the other hand, are not just interesting tradeoffs but clear winners: they match or (usually) outperform the object-sensitive analyses they are based on, while offering better precision, closely approaching the precision of the much more costly uniform hybrids. Overall, the best analyses in our evaluation set, both for highest-precision and for high performance with good precision, are selective hybrids.

Our evaluation setting uses the LogicBlox Datalog engine, v.3.9.0, on a Xeon X5650 2.67GHz machine with only one thread running at a time and 24GB of RAM (i.e., ample for the analyses studied). We analyze the DaCapo benchmark programs (v.2006-10-MR2) under JDK 1.6.0_37. This is a much larger set of libraries than earlier work [16, 142], which also results in differences in measurements, since the numbers shown integrate application- and library-level metrics. All runtime numbers are medians of three runs. As in other published work [1, 142], jython and hsqldb are analyzed with reflection disabled and hsqldb has its entry point set manually in a special harness.

For an illustration of the precision and performance spectrum, consider Figure 3.1, which plots analyses on precision/performance axes. The figure plots execution time against precision in the "may-fail casts" metric, i.e., the number of casts that the analysis cannot statically prove safe. Lower numbers are better on both axes, thus an analysis that is to the **left and below** another is better in both precision and performance. Values that are disproportionately high on the Y axis (i.e., large execution times) are clipped and plotted at the top of the figure, with the actual number included in parentheses. (Note that the Y axis starts at zero, while the X axis starts at an arbitrary point—we cannot know what is the "ideal" reference value for this metric.)

In terms of pre-existing analyses, Figure 3.1 illustrates what has been past experience: 2obj+H is the most precise analysis, but often heavy. 1obj and 2type+H are both quite fast, with 2type+H also showing very good precision, often approaching 2obj+H. The two call-site-sensitive analyses (1call, 1call+H) are mostly shown for reference and to demonstrate the insights discussed in Section 3.1. 1call is a fast analysis but vastly imprecise,

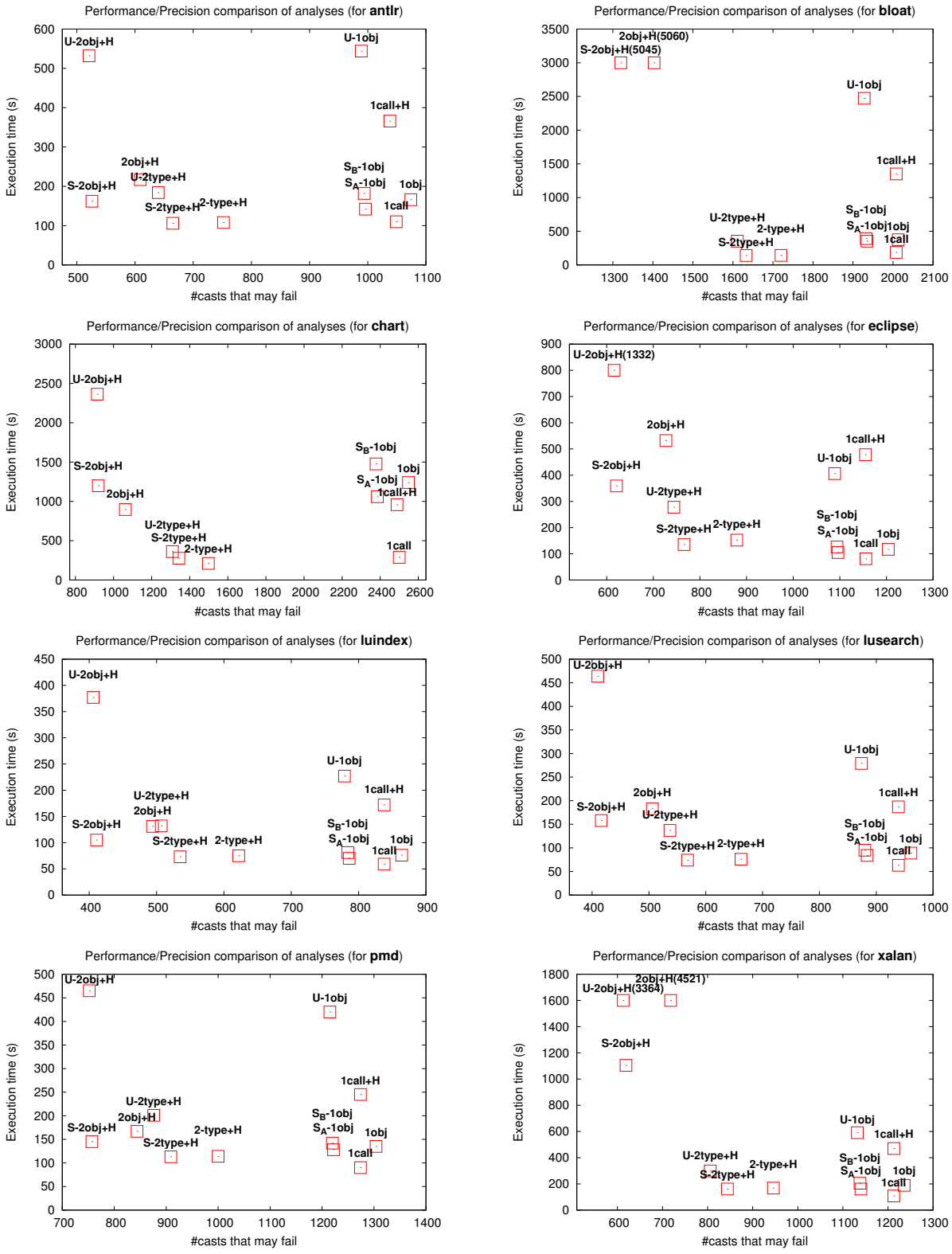


Figure 3.1: Graphical depiction of performance vs. precision metrics for eight of our benchmarks over all analyses. Lower is better on both axes. The Y axis is truncated for readability. Out-of-bounds points are included at lower Y values, with their real running time in parentheses.

while 1call+H is a bad tradeoff: its cost grows quite significantly relative to 1call without much precision added—call-site sensitivity is a bad choice for heap contexts.

As can be seen, the selective hybrid analyses (S_A-1obj, S_A-1obj, S-2obj+H, S-2type+H) usually offer an advantage over the corresponding base analysis (1obj, 2type+H, 2obj+H) in both precision and performance. In fact, selective hybrids are typically imperceptibly less precise than the corresponding uniform hybrid, yet much more precise than the base analysis. For instance, the plot points for S-2obj+H are always barely to the right of those for the theoretically more precise U-2obj+H (but significantly lower—uniform hybrids are very expensive), while they are clearly to the left of 2obj+H.

3.2.1 Detailed Results

Detailed results of our experiments are presented in Tables 3.1 and 3.2. The tables show precision and performance metrics for all analyses. The precision metrics are: (1) the average points-to set size (i.e., average over all variables of their points-to sets sizes), (2) the number of edges in the computed call-graph (which is typically a good proxy for the overall precision of the analysis, in broad strokes), and the results of two client analyses: (3) the number of virtual calls that could not be de-virtualized, and (4) the number of casts that could not be statically proven safe. A combination of these four metrics gives a reliable picture of the precision of an analysis. Note that the average points-to set size alone is not necessarily reliable, because it is influenced by a small number of library variables with enormous points-to sets. For comparison, the ***median*** points-to set size is 1, for all analyses and benchmarks.

Performance is shown with two metrics: (1) time and (2) total size of all *context-sensitive* points-to sets. Although time is the ultimate performance metric, it is brittle: one can argue that our time measurements are influenced by a multitude of implementation or environment factors, among which are the choice of underlying data structures, indexing, and the overall implementation of the points-to analysis, especially since it is running on a Datalog engine, with its own complex implementation choices hidden. The context-sensitive points-to set size metric does not suffer from any such measurement or implementation bias. It is the foremost internal complexity metric of a points-to analysis, and typically correlates well with time, for analyses of the same flavor. Note that analysis implementations that fundamentally differ from ours also try hard to minimize this metric in order to achieve peak performance: Lhoták's Paddle framework [81] is using binary decision diagrams (BDDs) for representing relations. The best BDD variable ordering (yielding "impressive results" [9]) is one that minimizes the total size of context-sensitive points-to sets. In short, it is reasonable to expect that improvements in this internal metric reinforce the verdict of which analysis yields better performance, not just in our setting but generally. Furthermore, the size of context-sensitive points-to sets serves as a valuable indicator of the internally different computation performed by various analyses: Analyses with almost identical precision metrics (e.g., context-insensitive points-to set sizes, call-graph edges) have vastly different context-sensitive points-to set sizes.

Since Tables 3.1-3.2 have a high information density, we guide the reader through some of

		avg. objs per var	call-graph edges	polymorph v-calls	may-fail casts	elapsed time (s)	sensitive VPT (M)		avg. objs per var	call-graph edges	polymorph v-calls	may-fail casts	elapsed time (s)	sensitive VPT (M)
			over 8.8K meths	over 33K calls	over 1.7K casts					over 10.2K meths	over 31K calls	over 2.8K casts		
1call	antlr	29.79	60999	1994	1049	110	16	bloat	43.96	70506	2138	2008	186	32.9
1call+H		29.58	60999	1994	1038	366	54.8		43.94	70506	2138	2008	1351	150.5
1obj		24.86	60194	1933	1074	166	14.3		41.26	69501	2076	2013	374	21.9
U-1obj		24.55	60194	1933	989	544	65		41.16	69501	2076	1928	2473	287.1
S_A-1obj		24.90	60202	1936	996	**142**	**10.5**		41.32	69511	2080	1935	**353**	**20.1**
S_B-1obj		24.61	60194	1933	994	182	17.3		41.18	69501	2076	1933	391	24
2obj+H		6.42	55548	1707	609	217	19.9		13.25	60726	1640	1403	5060	153.5
U-2obj+H		6.26	55548	1707	521	532	39.5		-	-	-	-	-	-
S-2obj+H		6.27	55548	1707	526	**162**	**13.9**		13.10	60726	1640	1320	**5045**	**149.8**
2type+H		17.60	55850	1759	752	108	5.3		16.28	62115	1888	1720	142	11.4
U-2type+H		7.14	55765	1746	640	184	8.9		14.71	61753	1827	1611	353	30.3
S-2type+H		17.39	55850	1759	665	**106**	**4.8**		16.10	62115	1888	1633	**140**	**11**
			over 15K meths	over 35K calls	over 3.5K casts					over 9.3K meths	over 23K calls	over 2K casts		
1call	chart	45.12	82156	2900	2500	288	49.6	eclipse	21.84	53006	1515	1156	81	12.3
1call+H		45.11	82078	2897	2488	957	120.9		21.65	53001	1514	1155	478	61.5
1obj		40.80	81423	2821	2548	1240	62.5		18.65	52114	1429	1204	117	9.4
U-1obj		-	-	-	-	-	-		18.41	51935	1404	1089	406	42.3
S_A-1obj		40.72	81075	2815	2385	**1059**	**39.7**		18.59	51958	1412	1096	**105**	**7.6**
S_B-1obj		40.11	81012	2808	2378	1477	89.7		18.43	51936	1404	1094	126	10.8
2obj+H		5.30	59162	1610	1062	**896**	67.6		5.75	44900	1163	727	532	44.6
U-2obj+H		4.99	59142	1603	915	2363	115.7		5.60	44899	1163	616	1332	89.8
S-2obj+H		5.00	59152	1610	920	1199	**53**		5.61	44900	1163	621	**359**	**32.3**
2type+H		7.02	62290	1775	1498	**211**	**13.3**		7.93	45318	1233	879	152	13.6
U-2type+H		5.89	62172	1756	1309	362	21.3		6.41	45123	1202	744	278	24.4
S-2type+H		6.57	62280	1775	1343	276	16.5		7.61	45235	1229	766	**135**	**11.5**
			over 10K meths	over 26K calls	over 2K casts					over 8.5K meths	over 21K calls	over 1.9K casts		
1call	hsqldb	18.56	54619	1552	1360	90	9.6	jython	20.64	50494	1525	1140	88	10.4
1call+H		18.53	54619	1552	1360	332	39.8		20.57	50480	1524	1140	401	50.6
1obj		15.41	53726	1480	1385	218	13.9		18.21	49622	1448	1157	119	8.7
U-1obj		15.30	53724	1479	1302	1351	74.3		18.01	49614	1448	1087	375	43.2
S_A-1obj		15.58	53730	1482	1320	**183**	**9.6**		18.19	49622	1453	1094	138	**6.7**
S_B-1obj		15.32	53724	1479	1308	329	29.5		18.09	49614	1448	1092	138	10.8
2obj+H		-	-	-	-	-	-		-	-	-	-	-	-
U-2obj+H		-	-	-	-	-	-		-	-	-	-	-	-
S-2obj+H		-	-	-	-	-	-		-	-	-	-	-	-
2type+H		7.92	49421	1276	1031	**195**	**13.7**		8.55	43269	1268	909	731	**52**
U-2type+H		6.71	49319	1263	923	583	42.9		7.18	43138	1236	822	1363	118.4
S-2type+H		7.74	49421	1276	948	238	20.5		8.30	43269	1268	840	**676**	56.5

Table 3.1: Precision and performance metrics for all benchmarks and analyses, grouped by relevance. Continues in Table 3.2.

		avg. objs per var	call-graph edges	polymorph v-calls	may-fail casts	elapsed time (s)	sensitive VPT (M)		avg. objs per var	call-graph edges	polymorph v-calls	may-fail casts	elapsed time (s)	sensitive VPT (M)
			over 7.9K meths	over 18K calls	over 1.4K casts					over 8.4K meths	over 19K calls	over 1.5K casts		
1call	luindex	17.65	41992	1180	838	59	7.8	lusearch	18.64	45270	1360	939	63	8.7
1call+H		17.58	41992	1180	838	172	26.1		18.47	45270	1360	939	187	28.5
1obj		14.94	41103	1119	864	76	5.4		15.71	44371	1299	961	89	6.2
U-1obj		14.81	41103	1119	779	227	26.3		15.57	44365	1299	874	279	30.3
S_A-1obj		14.97	41111	1122	786	**70**	**4.1**		15.79	44379	1302	884	**84**	**5.3**
S_B-1obj		14.83	41103	1119	784	81	6.4		15.60	44371	1299	880	95	7.2
2obj+H		4.77	36580	894	494	131	11.1		4.71	39452	1065	506	183	13.2
U-2obj+H		4.55	36580	894	406	377	22.4		4.49	39446	1065	410	464	26.3
S-2obj+H		4.55	36580	894	411	**105**	**7.2**		4.50	39452	1065	416	**158**	**10**
2type+H		6.20	36889	949	622	75	4.5		6.13	39763	1122	662	76	4.2
U-2type+H		5.15	36796	932	507	132	7.6		5.10	39662	1103	537	137	7.8
S-2type+H		5.92	36889	949	535	**73**	**3.5**		5.86	39763	1122	568	**74**	**3.6**
			over 9.2K methss	over 21K calls	over 2K casts					over 10.5K meths	over 26K calls	over 2K casts		
1call	pmd	19.94	49097	1249	1274	90	11.4	xalan	25.50	57168	1976	1213	108	14.5
1call+H		19.82	49097	1249	1274	245	35.9		25.38	57168	1976	1213	470	59.8
1obj		17.36	48250	1187	1304	135	7.9		21.86	56412	1920	1236	189	15.5
U1obj		17.22	48250	1187	1215	420	42.6		21.59	56158	1905	1132	591	67.5
S_A1obj		17.37	48258	1190	1222	**128**	**6.9**		21.84	56404	1921	1140	**161**	**10.9**
S_B1obj		17.24	48250	1187	1220	142	9.2		21.69	56395	1918	1138	205	18.2
2obj+H		4.87	43068	937	844	167	13.2		5.48	50148	1619	718	4521	166.6
U2obj+H		4.68	43067	937	752	465	30.5		5.22	50054	1615	613	3364	171.7
S2obj+H		4.68	43067	937	757	**145**	**10**		5.23	50054	1615	619	**1105**	**63.3**
2type+H		6.35	43401	988	1000	114	4.5		7.52	50539	1677	946	168	10.2
U2type+H		5.28	43315	976	876	201	9.7		6.19	50432	1660	806	299	17.4
S2type+H		6.10	43400	988	909	**113**	**3.9**		7.16	50526	1677	844	**161**	**9**

Table 3.2: Precision and performance metrics for all benchmarks and analyses, grouped by relevance. In all cases **lower is better**. Dash (-) entries are for analyses that did not terminate in 90 minutes. The 4 precision metrics shown are the average size of points-to sets (how many heap objects are computed to be pointed-to per-var), the number of edges in the computed call-graph, the number of virtual calls whose target cannot be disambiguated by the analysis, and the number of casts that cannot be statically shown safe by the analysis. Reference numbers (e.g., total reachable casts in the program) are shown in parentheses in the metric's heading. These numbers change little per-analysis. Performance is shown as running time and size of context-sensitive var-points-to data (the main platform-independent internal complexity metric). Best performance numbers per-analysis-group are **in bold**.

the most important findings below (see also a partial illustration in Figure 3.1).

- ***General observations.*** The analyses shown are in 4 groups of closely related analyses: call-site sensitive, 1-object-sensitive, 2-object-sensitive with a 1-context-sensitive heap, and 2-type-sensitive with a 1-context-sensitive heap. These analyses span a large performance and precision spectrum. For instance, for the chart benchmark, the least precise analysis, 1call, runs for under 5 minutes and computes an average points-to size of over 45, while the most precise, U-2obj+H, runs for over 53 minutes and computes an average points-to size of under 5. The difference in precision is also vividly shown in the "may-fail casts" metric: the 1call analysis cannot prove 2500 casts safe, while the U-2obj+H fails to prove safe just 915 casts (both numbers from a total of about 3.5K reachable casts—the exact number varies slightly due to method reachability variation per analysis).

  Specifically, 1call, 1obj, and 2type+H are the fastest analyses in our set (for different programs each), with each one offering significant precision enhancements relative to the previous. 2obj+H is typically slower but manageable, and achieves very high precision.

- ***Uniform hybrid analyses.*** Recall that uniform hybrid analyses (U-1obj, U-2obj+H, U-2type+H) were defined to always keep a combination of object-sensitive and call-site-sensitive context. As a result, the analyses are more precise than their respective base analyses (1obj, 2obj+H, 2type+H), especially in the "may-fail casts" metric. However, this precision comes at great cost: uniform hybrid analyses are often 3x or more slower than their base analyses with twice as large, or more, context-sensitive points-to sets. U-1obj and U-2obj+H are plainly bad tradeoffs in the design space: for a slight increase in precision, the performance cost is heavy.

  U-2type+H is a bit more reasonable: it achieves more significant precision gains and its performance toll is often under 2x while still terminating comfortably for all our benchmarks. In fact, a surprising finding was that U-2type+H is a tempting alternative to 2obj+H for applications that need very high precision, given its good scalability. A possible explanation is due to the specific nature of type-sensitive analyses: the class in which a receiver object is allocated forms a good differentiator of behavior when combined with a call site.

- ***1obj hybrids.*** We presented two selective hybrids of a 1-object-sensitive analysis: S_A-1obj (which keeps either an allocation site or a call-site as context, but not both) and S_B-1obj (which always keeps an allocation site as context and occasionally adds a call-site to it). They both turn out to be interesting analyses from a practical standpoint. The former is consistently faster than the base 1obj analysis, with roughly similar precision and occasionally (for the "may-fail casts" metric) higher precision. The size of context-sensitive points-to sets also confirms that this is a "lighter" analysis that is likely to cost less in any context. The S_B-1obj analysis is always more precise than 1obj (as is statically guaranteed) for a slight extra cost. Indeed, S_B-1obj is a good approximation of the uniform hybrid analysis, U-1obj, in terms of precision, for a fraction (typically less than a third) of the cost.

- ***2obj+H hybrids.*** The selective hybrid idea yields even more dividends when applied to the very precise 2obj+H analysis. S-2obj+H is more precise than 2obj+H and only very slightly less precise than the uniform hybrid, U-2obj+H. In terms of performance, however, the analysis is typically well over 3 times faster than U-2obj+H, and significantly faster (an average of **53%** speedup) than 2obj+H. This is interesting, given the practical value of 2obj+H, since it establishes a new sweet spot in the space of relatively scalable but highly precise analyses: S-2obj+H is both more precise than 2obj+H (especially for "may-fail casts") and substantially faster.

- ***2type+H hybrids.*** The 2type+H analysis variations are also highly interesting in practice. This is an analysis space that yields excellent precision relative to its low cost. There are few cases in which one might prefer some other inexpensive analysis over 2type+H given the combination of precision and competitive performance of the latter. As we saw, the uniform hybrid, U-2type+H, is an interesting tradeoff in this space. The selective hybrid, S-2type+H, also performs quite well. It is just as fast or slightly faster than the base analysis 2type+H, while also being more precise.

3.3 Summary

This chapter presented a comprehensive map for the exploration of context combinations in points-to analysis, and used it to discover several interesting design points. Object sensitivity and call-site sensitivity had never been fruitfully combined in the past, although the idea is clearly tempting. We speculate that the reasons for the paucity of hybrid-context sensitivity results have been a) the difficulty of having a good enough model for the space of combinations and a convenient implementation to explore it; b) a belief that nothing fruitful will come out of such a combination, because call-site sensitivity incurs a high performance cost, which is more profitably spent on an extra level of object sensitivity. The latter insight is mostly true, but only if one considers uniform hybrid analyses. As we saw, much of the benefit of call-site and object-sensitive hybrids comes from allowing the context to vary between pure object-sensitive and extended. The result of our work has been new sweet spots, in both precision and performance, for some of the most practically relevant analysis variations.

There are several interesting directions for further work that open up. First, our model gives the ability for further experimentation, e.g., with deeper-context analyses. Furthermore, it is interesting to examine if a hybrid context should perhaps change form more aggressively. The `Merge` and `MergeStatic` functions could examine the context passed to them as argument and create different kinds of contexts in return. For instance, the context of a statically called method could have a different form (e.g., more elements) for a call made inside another statically called method vs. a call made in a virtual method. Similarly, objects could have different context, via the `Record` function, depending on the context form of their allocating method. To explore this space without blind guessing, one needs to understand what programming patterns are best handled by hybrid contexts and how. For deep contexts this remains a challenge, as it is hard to reason about how context elements affect precision.

(E.g., past work had to offer involved arguments for why the allocator object of the receiver object of a method is a better context element than the caller object [142].) This challenge is, however, worth addressing for the next level of benefit in context-sensitive points-to analysis.

4. INTROSPECTIVE ANALYSIS

> Do what I do. Hold tight and pretend it's a plan!
>
> *The 11th Doctor* - Doctor Who

Previous chapters already presented context sensitivity as a common way of pursuing precision and scalability in points-to analysis. An oft-remarked fact about context sensitivity, however, is that even the best algorithms have a common failure mode when they cannot maintain precision. Past literature reports that "the performance of a deep-context analysis is bimodal" [142]; "context-sensitive analyses have been associated with very large numbers of contexts" [83]; "algorithms completely hit a wall after a few iterations, with the number of tuples exploding exponentially" [91]. The experimental results in chapter 3 (Tables 3.1-3.2) show a failure to run a 2-object-sensitive analysis in under **90 minutes** for 2 of 10 DaCapo benchmarks, while 2 more benchmarks take more than **1,000 seconds**, although most other benchmarks of similar or larger size get analyzed in under **200 seconds**.

Thus, when context sensitivity works, it works formidably, in terms of both precision and performance. When it fails, however, it fails miserably, quickly exploding in complexity. In contrast, context-insensitive analyses uniformly scale well, for the same inputs. Figure 4.1 vividly demonstrates this phenomenon for the DaCapo benchmarks, analyzed with the Doop framework [16] under a context-insensitive (insens) analysis and a 2-object-sensitive analysis with a context-sensitive heap (2objH). (The chart truncates the analysis time of the longest-running benchmarks. Two of them, hsqldb and jython, timed out after 90 minutes on a 24GB machine, and would not terminate even for much longer timeouts.) As can be seen, context-insensitive analyses vary relatively little in performance, while context sensitivity often causes running time (and memory usage) to explode.

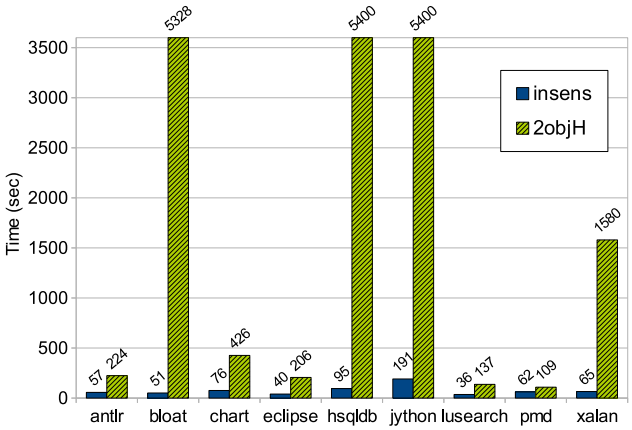


Figure 4.1: Comparison of execution times of context-insensitive analysis vs. 2-object-sensitive with context-sensitive heap. The y-axis is truncated to 1 hour for readability.

Faced with this unpredictability of context sensitivity, a common reaction is to avoid it, favoring context-insensitive analyses, and, consequently, missing significant precision benefits for well-behaved programs. Even worse, for some applications, eschewing expensive context sensitivity is not an option—a context-insensitive analysis is just not good enough and even an often inexpensive context-sensitive analysis (e.g., 2-type-sensitive with a context-sensitive heap) fails to yield good precision. Reports from industry [24] and academic researchers [23] alike reiterate that precise context sensitivity is essential for information-flow analysis, taint analysis, and other security analyses.

We can ask ourselves, why does this scalability barrier arise? The core problem is that, for some objects or methods, the points-to information is imprecise enough that more context does not help, while incurring a heavy overhead [142]. Consider a method argument that was found to point to n objects by a less precise analysis. Further analyzing the method in c different contexts (or, equivalently, increasing context depth by 1) will ideally yield n/c points-to facts per context, perfectly splitting the previous n-object points-to set, thus yielding both precision and scalability. In the worst case, however, increasing the context depth will result in c copies of n points-to facts each: the extra context depth will not have yielded more precision, but will have multiplied the space and time costs. When this occurs, the analysis cost explodes for greater context depths.

The focus of this chapter is on the detection and prevention of pathological behavior in context-sensitive analyses, with minimal intervention. In this way, we achieve many of the precision benefits of context sensitivity without sacrificing scalability. It does not seem possible to know in advance (e.g., by identifying syntactic features of the program) which program elements may be responsible for pathological behavior. Nevertheless, we argue that it is possible to identify such elements with a scalable context-insensitive analysis. We introduce the concept of *introspective context sensitivity*: during a first, context-insensitive, analysis pass, the analysis observes symptoms indicating that the cost may get out of hand for deeper context. This detects exactly the pathology identified above. In its simplest form, the analysis will ask "which program sites currently have points-to information that may grow too large for an extra level of context?" Using a configurable second pass[1], such sites will be re-analyzed with shallow context, even though the rest of the program will be re-analyzed with a deeper context.

In intuitive terms, introspective context sensitivity performs a cost-benefit calculation, with an emphasis on the potential cost of increasing context depth, since cost can be estimated more reliably. Fairly simple—yet not always obvious—heuristics can estimate this cost well.

As a general pattern, this approach is familiar. Even in the context of points-to analysis, the pattern of performing a coarse-grained analysis and using it to tune a finer-grained one has been explored before, as in earlier *refinement-based* [150] or *pruning* [91] techniques. Nevertheless, such past approaches differ from our approach in terms of both external applicability and impact: they fundamentally apply to demand-driven (as opposed to all-points on the entire program) analyses and they either do not target context depth or only apply to specific kinds of context—related work in Section 8.2 includes a detailed discussion.

[1] In theory, this pattern could be repeated for more passes.

The net outcome of our work is not a "first line of defense" analysis, but an "if all else fails" analysis. Users are still better advised to first use traditional context-sensitive algorithms, in the hope that these will scale well and provide good precision. When this fails, however, we show that we can provide a highly reliable knob for filtering out worst-case performance at a small cost in precision. Our experiments demonstrate that the user can "dial-in" scalability, to the exact level required. For instance, as seen in Figure 4.1, a precise 2objH analysis fails to run in under 90 minutes on a 24GB machine for 3 of our experimental subjects. However, we can get an introspective context-sensitive analysis to scale to all benchmarks in under **12 minutes**, while still gaining significant precision over a context-insensitive analysis. Yet another introspective analysis scales to all but one benchmark in under **20 minutes**, while sacrificing a fraction of precision (keeping about **2/3** of the precision gains of a full 2objH analysis). For call-site sensitive analyses, the gains are even more pronounced, with several benchmarks exhibiting at least **300%** speedups, without sacrificing nearly any precision.

Overall, this chapter describes the following contributions:

- We offer an approach to refining a context-sensitive analysis while avoiding its worst-case cost. The approach relies on first running a context-insensitive analysis and using its results to inform the application of context sensitivity. Much of the challenge concerns the question of *how* to use this information, i.e., what heuristics yield good behavior.
- We encode the approach in a simple form, by incremental modifications of a general declarative analysis pattern. Therefore, our approach works on virtually any algorithm expressed in this manner. Our implementation is on the Doop framework and already applies to the over 30 analysis algorithms that the framework has to offer.
- We show experimentally the benefit of introspective context sensitivity. We quantify the precision loss and scalability gains for different parameter settings and show that there is a dial that users can tune, to select points in this spectrum. Even our high-precision settings are effective in eliminating behavior outliers, showing that introspective context sensitivity has core value: previously hopeless analyses suddenly become feasible, for little precision loss. We believe that the result is to give confidence that context-sensitive analyses can be used in virtually any setting and not just in the nebulous "when they work well" case.

4.1 Formulation of Introspective Context Sensitivity

We demonstrate introspective context sensitivity via incremental changes to the existing model for context-sensitive, flow-insensitive points-to analysis algorithms presented in Section 2.12. As previously mentioned, the logical formalism of this model is very close to the core components of our actual analysis implementation.

Input relations. Two additional input relations are added to those presented in Figure 2.5 and are used exclusively for the purposes of introspective context sensitivity. SiteToRefine

and ObjectToRefine encode the program points (invocation site/method combinations, and allocation sites) that will employ a different context abstraction from the rest.

```
SiteToRefine(invo: I, meth: M)
ObjectToRefine(heap: H)
```

Figure 4.2: Additional Datalog input relations expanding those of Figure 2.5.

Computed (output) relations. No additional output relations need to be added to those of Figure 2.6.

Constructors for Context Sensitivity. As previously mentioned, the base rules of any analysis are not concerned with what kind of context sensitivity is used. The context flavor and depth aspects are completely hidden mainly behind constructor functions `Record` and `Merge`. These functions are sufficient for modeling a very large variety of context-sensitive analyses. In addition to those two, introspective context sensitivity adds two more constructor functions to act as counterparts—`RecordRefined` and `MergeRefined`. They are directly analogous to `Record` and `Merge` but just apply to different program points. These constructors are the machinery for introspective context sensitivity: they vary the context sensitivity of the analysis for a subset of the heap objects and methods.

```
RecordRefined(heap: H, ctx: C) = newHCtx: HC
MergeRefined(heap: H, hctx: HC, invo: I, ctx: C) = newCtx: C
```

Figure 4.3: Additional Datalog constructors of contexts that filter which objects and which call-sites/target methods should have a different (i.e., more precise) context in introspective context sensitivity.

Analysis logic. The rules for introspective context sensitivity supplement the existing logic presented in Section 2.12. Out of the core nine rules, only two need to be enhanced with duplicate versions, as shown in Figure 4.4. Each rule pair offers a version for the default handling of context and another for the more precise handling. (In the full implementation, there are some two-dozen rules that construct new contexts, instead of the two in the model, and all are duplicated accordingly.)

The first pair considers the handling of object allocation, with the first rule applying when the allocation is to be treated with no additional precision, whereas the second applies a more refined context—whether the ObjectToRefine holds or not for the current allocation site. The different handling of context is controlled via the `Record` and `RecordRefined` functions, respectively.

The second pair is somewhat more involved but still follows the same reasoning, applied to method invocations. The filtering is controlled via the SiteToRefine relation and the different handling of (calling) context is done via the `Merge` and `MergeRefined` functions.

Therefore, we can effect any change we want to the context sensitivity of an analysis, on a per-object/per-call-site basis, by supplying the right input relations ObjectToRefine or

SiteToRefine and setting the appropriate constructors, RecordRefined and MergeRefined to implement a different flavor/depth of context sensitivity. We discuss such options next.

```
Record(heap, ctx) = ?hctx,
VarPointsTo(var, ctx, heap, ?hctx) ⟵
    Reachable(meth, ctx), Alloc(var, heap, meth),
    ! ObjectToRefine(heap).

// Duplicate rule, for introspective context sensitivity
RecordRefined(heap, ctx) = ?hctx,
VarPointsTo(var, ctx, heap, ?hctx) ⟵
    Reachable(meth, ctx), Alloc(var, heap, meth),
    ObjectToRefine(heap).
```

```
Merge(heap, hctx, invo, callerCtx) = ?calleeCtx,
Reachable(toMeth, ?calleeCtx),
VarPointsTo(this, ?calleeCtx, heap, hctx),
CallGraphEdge(invo, callerCtx, toMeth, ?calleeCtx) ⟵
    VCall(base, sig, invo, inMeth),
    Reachable(inMeth, callerCtx),
    VarPointsTo(base, callerCtx, heap, hctx),
    HeapType(heap, heapT),
    Lookup(heapT, sig, toMeth),
    ThisVar(toMeth, this),
    ! SiteToRefine(invo, toMeth).

// Duplicate rule, for introspective context sensitivity
MergeRefined(heap, hctx, invo, callerCtx) = ?calleeCtx,
Reachable(toMeth, ?calleeCtx),
VarPointsTo(this, ?calleeCtx, heap, hctx),
CallGraphEdge(invo, callerCtx, toMeth, ?calleeCtx) ⟵
    VCall(base, sig, invo, inMeth),
    Reachable(inMeth, callerCtx),
    VarPointsTo(base, callerCtx, heap, hctx),
    HeapType(heap, heapT),
    Lookup(heapT, sig, toMeth),
    ThisVar(toMeth, this),
    SiteToRefine(invo, toMeth).
```

Figure 4.4: Datalog rules for different handling of context creation on a per-object/per-call-site basis.

4.2 How To Selectively Refine

The model of the previous section allows to easily configure context sensitivity in a large variety of ways. For instance, some methods (or some call sites) can be analyzed with object sensitivity while others are analyzed with call-site sensitivity, of any depth. One aspect to determine, therefore, is the two analyses that will be used in different program points.

Another question is how to populate the ObjectToRefine and SiteToRefine input relations. One could attempt to do so by mere syntactic inspection of the program. For example, methods containing cast statements can be analyzed with a higher context depth. In our work, we have failed to identify such syntactic heuristics that would yield benefit.

Instead, our introspective context sensitivity consists of running the analysis *twice*. The first time, ObjectToRefine and SiteToRefine are *empty* and the Merge/Record context constructors are set so that an inexpensive but scalable analysis is performed. In our experimental setting, these constuctor functions return a unique constant value, $\star$, resulting in a context-insensitive analysis:

```
Record(heap,ctx) = ⋆
Merge(heap, hctx, invo, ctx) = ⋆
```

The MergeRefined and RecordRefined constructors are set to implement an expensive context-sensitive analysis, following the techniques of Chapter 3. Yet, these constructors are not relevant in the first analysis run, since the rules employing them are predicated on having elements in SiteToRefine and ObjectToRefine, respectively.

Subsequently, we use the results of the context-insensitive analysis to compute which program elements to refine (i.e., populate the SiteToRefine and ObjectToRefine relations), and run the analysis a second time. The result is that a subset of the program elements are analyzed context-sensitively, while the rest are analyzed context-insensitively even during the second analysis run. In practical terms, the former set is larger than the latter: we focus on identifying a relatively small number of program elements that may disproportionately affect analysis costs and to analyze them context-insensitively, while the majority of program elements are analyzed context-sensitively.

Therefore, the main challenge is to identify a program client analysis (over the results of a context-insensitive points-to analysis) to predict which program elements should not be refined. Our criterion is based on cost rather than expected benefit, since the latter is very hard to estimate in an all-points (as opposed to demand-driven) program analysis. There are several cost metrics that we can mix-and-match to create introspective analysis heuristics:

1. Compute at every invocation site the cumulative size of all points-to sets of actual arguments to the method call. (This is the argument *in-flow* of the method call.)

2. Compute for every method the cumulative (or maximum, for a variant of the metric) size of points-to sets over all local variables. (This is the method's *total points-to volume* or *max var-points-to*.)

3. Compute for each object (i.e., allocation site) the maximum (or total, for a variant of the metric) field points-to set over all of its fields (This is the object's *max field points-to*, or *total field points-to*.)

4. Compute for every method the maximum max field-points-to (metric 3) among objects pointed to by the method's local variables. (This is the method's *max var-field points-to*.)

5. Compute for each object (i.e., allocation site) the number of local variables pointing to it. (This is the object's *pointed-by* metric.)

As can be seen, these metrics can vary in sophistication but all of them attempt to estimate the cost that will be incurred if the same method or allocation site were to be analyzed context-sensitively. Indeed, our emphasis is not on the sophistication of the metrics or on their fine-tuning. Instead, it is on their simplicity and ease of composition so that one can create parameterizable analyses: a knob for adjusting the precision/scalability tradeoff. For example, we propose two heuristic combinations of these metrics:

> **Heuristic-A.** Refine all allocation sites except those with a *pointed-by* (Metric #5) higher than a constant K. Refine all method call sites except those with either an *in-flow* (Metric #1) higher than a constant L or a *max var-field points-to* (Metric #4) higher than a constant M.

> **Heuristic-B.** Refine all method call sites except those that invoke methods with a *total points-to volume* (Metric #2) above a constant P. Refine all object allocations except those for which the product of *total field points-to* and *pointed-by* (Metrics #3 and #5) exceeds a constant Q. The product of these two metrics can be seen as an object's total potential for weighing down the analysis.

These heuristics are themselves tunable, by adjusting the constant parameters. In the rest of the chapter, when we refer to *Heuristic-A* in measurements, the values of K, L, M will be 100, 100, and 200, respectively; when we refer to *Heuristic-B*, the values of P and Q will both be 10000. The point of picking clear-cut reference numbers is to argue that the value of the technique does not come from excessive tuning but from the underlying power of the introspective analysis idea—even relatively large variations of these numbers make scarcely any difference in the total picture of results over multiple programs.

Intuition. The main insight behind our Heuristic-Approach is that there are many program elements whose analysis cost is vastly disproportionate to their importance. If such

elements are analyzed less precisely, the analysis will avoid significant burden without incurring large precision losses. The above two heuristics try to estimate "disproportionate cost" but have no way of estimating the "importance" of a program element. It would be an interesting direction for future work to estimate this importance, i.e., to define metrics that capture the extent of the impact of a program element's precision on all other program elements.

Furthermore, a program element with very large cost, in terms of associated points-to facts, may be hopeless for good enough precision anyway. On the other hand, the above intuition could potentially unfold the other way: it is possible that context sensitivity can productively distinguish seemingly-imprecise program elements (e.g., methods with large cumulative points-to sets, but also many callers) and maintain precision.

Implementation. The above metrics and heuristics can be easily implemented as short analyses over the result of a context-insensitive points-to analysis. For instance, the implementation of the *in-flow* metric (#1) is the following Datalog rules, which define an intermediate predicate and aggregate over it. ("_" is a nameless variable denoting any value, and "{...}" denotes an aggregate operation over the relations inside the curly brackets —in our case a total count of matching tuples.)

```
HeapsPerInvocationPerArg(invo, arg, heap) ←
    CallGraphEdge(invo, _, _, _),
    ActualArg(invo, _, arg),
    VarPointsTo(arg, _, heap, _).

InFlow(invo, ?result) ←
    COUNT{ HeapsPerInvocationPerArg(invo, _, _) } = ?result.
```

4.3 Evaluation

Our evaluation setting uses the LogicBlox Datalog engine, v.3.9.0, on a Xeon E5530 2.4GHz machine with only one thread running at a time and 24GB of RAM. We analyze the DaCapo benchmark programs (v.2006-10-MR2) with Open JDK 1.6.0_24. We run all benchmarks with default Doop settings, including full reflection support. We selected a priori 6 of the Dacapo benchmarks as our experimental subjects: these are the programs that exhibit scalability problems based on past literature (e.g., Chapter 3). Other benchmarks typically run in half the time of the fastest benchmark of our set for deep context-sensitive analyses. Since our technique is explicitly not a "first line of defense", benchmarks that are already certain to scale are out of scope.

The results of our experiments are shown in Figures 4.5, 4.6, and 4.7. We evaluate two variants of introspective context sensitivity corresponding to *Heuristic-A* and *Heuristic-B* from Section 4.2. We test the three main flavors of context sensitivity: object sensitivity [105, 106], call-site sensitivity [137, 140], and type sensitivity [142]. The three flavors have

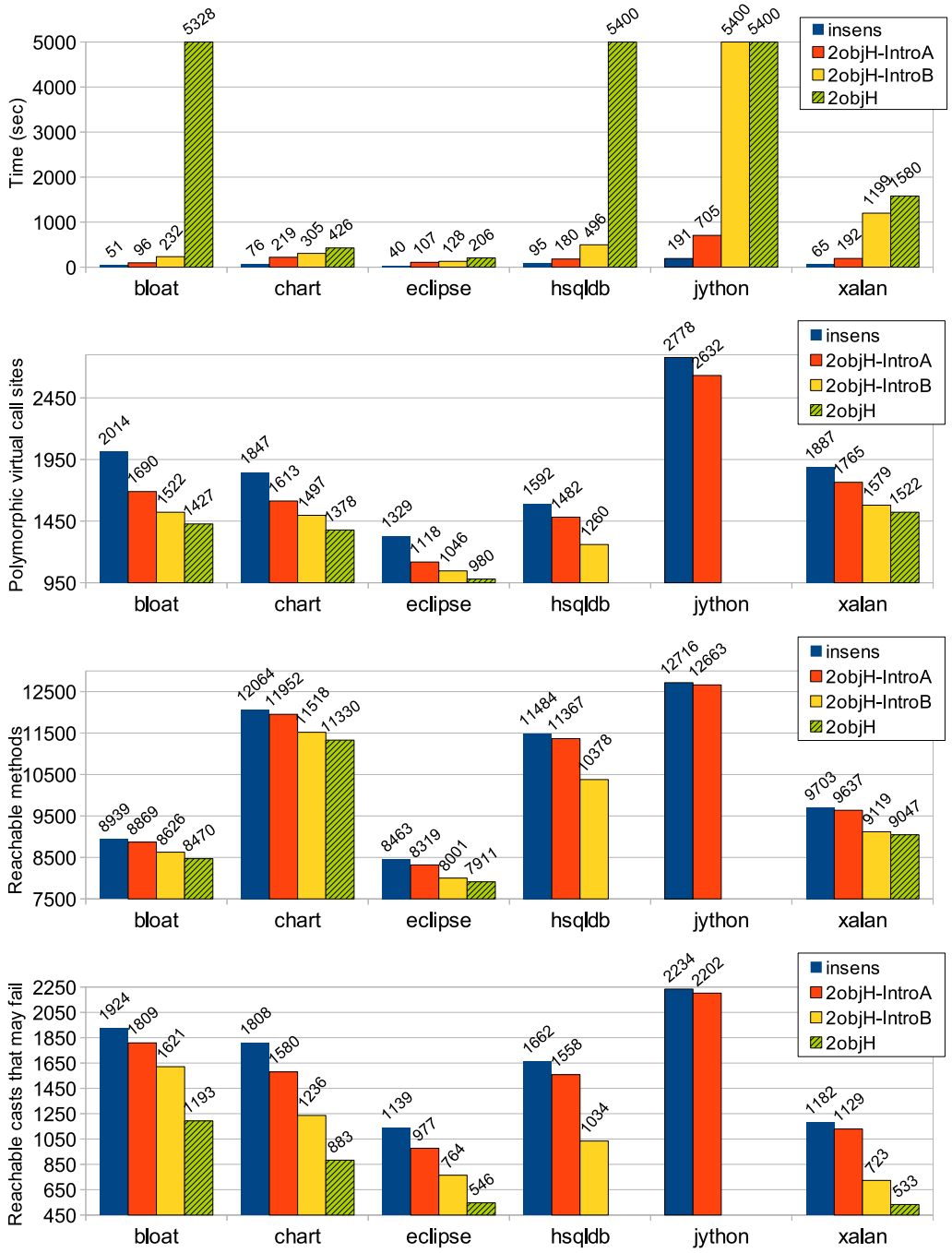


Figure 4.5: Performance and precision (3 separate metrics: calls that cannot be devirtualized, reachable methods, casts that cannot be eliminated) for introspective context-sensitive variants of a **2objH** analysis, compared with baselines (2objH and insensitive).

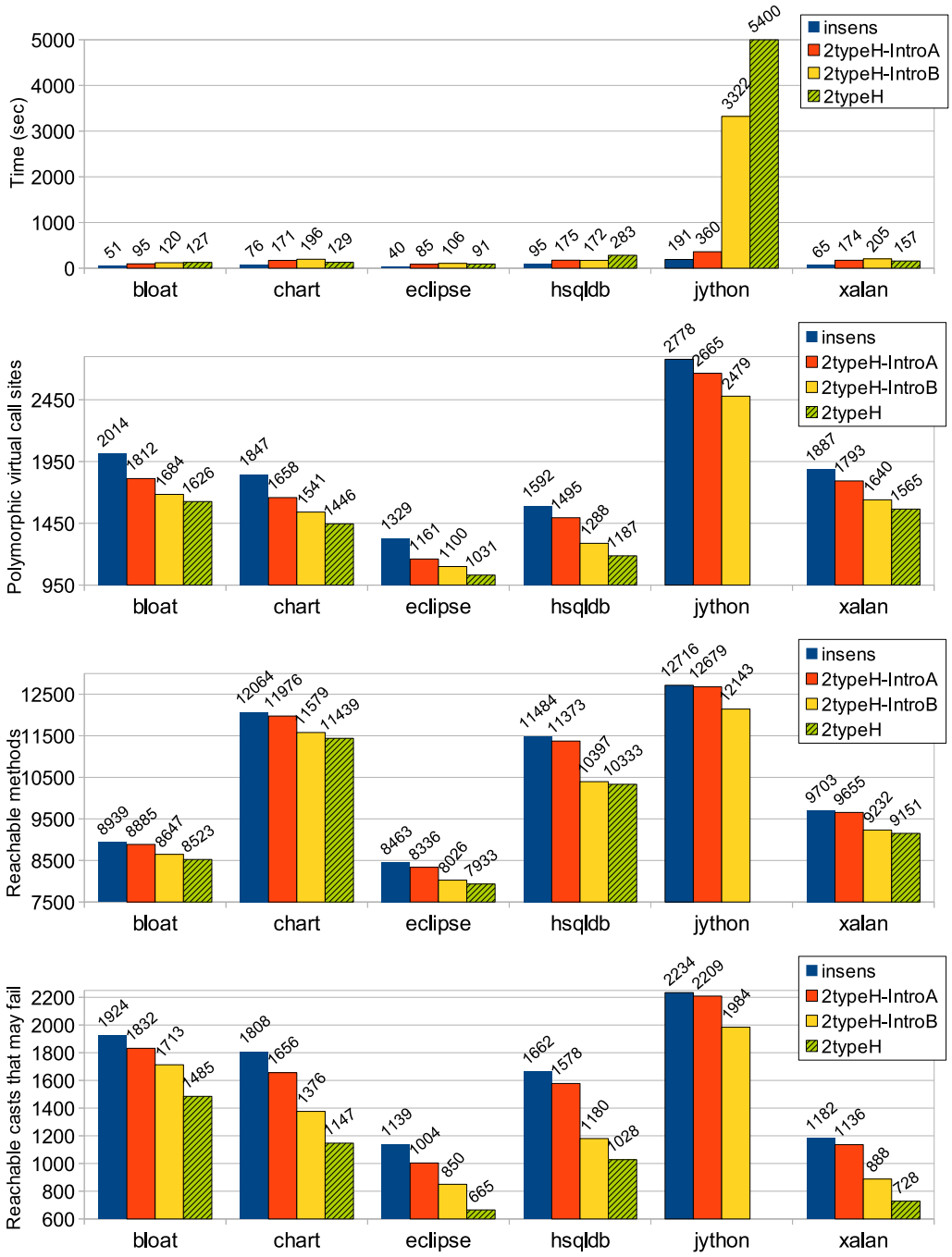


Figure 4.6: Performance and precision (3 separate metrics: calls that cannot be devirtualized, reachable methods, casts that cannot be eliminated) for introspective context-sensitive variants of a **2typeH** analysis, compared with baselines (2typeH and insensitive).

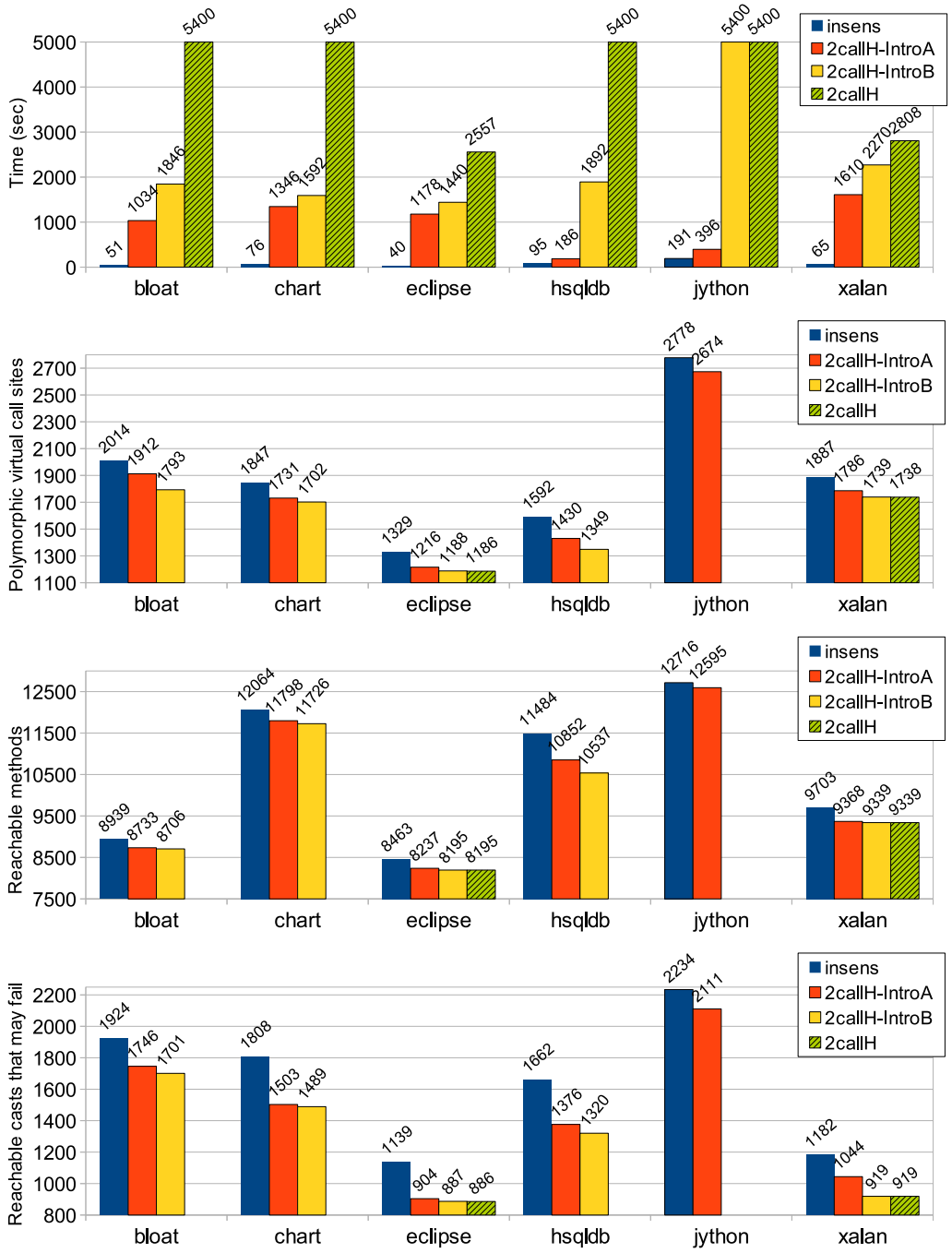


Figure 4.7: Performance and precision (3 separate metrics: calls that cannot be devirtualized, reachable methods, casts that cannot be eliminated) for introspective context-sensitive variants of a **2callH** analysis, compared with baselines (2callH and insensitive).

very different profiles of practical use and scalability, as detailed next.

4.3.1 Object Sensitivity

Deep-context, object-sensitive analyses are the most precise in practice, but do not always scale well. Starting from a 2-object-sensitive analysis with a 1-context-sensitive heap (2objH), we define our two introspective versions (2objH-IntroA and 2objH-IntroB for Heuristic-A and Heuristic-B, respectively). Figure 4.5 plots first the execution time and then three precision metrics for all analyses. In all cases *lower is better*. There is no real "metric" for precision, since each client may have unique needs, but our three metrics together should yield a reasonable projection of precision. Note that since there is no "ground truth" for the ideal value of precision metrics, their chart scales are arbitrary (and differences are not as visually pronounced as could be because of plotting multiple benchmarks on a single chart) but the insensitive/2objH analyses serve as upper/lower reference markers in practice. We use a 90 minutes timeout. The jython and hsqldb benchmarks did not terminate for 2objH, and jython did not terminate for 2objH-IntroB either. We indicate non-termination with full bars in the top (time) chart and the absence of bars in the bottom three (precision) charts.

As can be seen in Figure 4.5, the two introspective variants scale much better than the full 2objH analysis. Indeed, IntroA scales to all benchmarks, while showing significant precision gains over an insensitive analysis. IntroB is even more precise: it covers *more than two-thirds* of the precision advantage of 2objH over an insensitive analysis for most benchmarks and precision metrics, while scaling significantly better.

4.3.2 Type Sensitivity

Type sensitivity is designed with the explicit purpose of providing more scalability than object sensitivity but in a very different manner: instead of avoiding high context depths, type sensitivity makes each context element coarser. Thus it is doubly interesting to see if introspection can add benefit to type-sensitive analyses. Type sensitivity is not immune to the pathologies of object sensitivity: for instance, in our benchmark set it does not scale to jython.

Figure 4.6 shows our results, plotting variants of a 2-type-sensitive analysis with a 1-context-sensitive heap (2typeH), and following the same conventions as earlier. (The insensitive baseline is inherited and not re-run.) As can be seen, the IntroB version scales to all programs while typically maintaining very good precision—often close to the full 2typeH. The IntroA version has the desirable feature of near-perfect scalability: its maximum runtime for *any* benchmark is 360 seconds. At the same time it exhibits precision gains compared to a context-insensitive analysis, although these are noticeably lower than the precision gains of IntroB.

4.3.3 Call-site Sensitivity

Call-site sensitivity is the traditional flavor of context sensitivity—a virtual synonym for the term. In practice, call-site sensitivity is quite good for some analysis clients but almost never scalable at context depths greater than 1.

As Figure 4.7 shows, introspective context sensitivity performs remarkably well when applied to a 2-call-site-sensitive analysis with a 1-context-sensitive heap (2callH). The base 2callH analysis does not terminate for 4 out of 6 of our benchmarks, while introspective analyses terminate either for all (IntroA) or for nearly all (5 out of 6 for IntroB). Furthermore, IntroB seems to achieve the full precision of 2callH for the two benchmarks for which the latter yields results, and for all different metrics! Combined with the across-the-board scalability gains shown in the timing chart, this confirms the effectiveness of introspection for tuning out extreme analysis costs. IntroA is not far behind in precision, obtaining more than two-thirds of the precision gains of IntroB for most metrics and benchmarks.

4.3.4 Discussion

The above timings of introspective context sensitivity do not include the cost of first running a context-insensitive analysis, and other timing overheads (relatively constant at about 100 seconds) related to computing the objects and sites to refine and re-running an analysis.[2] We did not include these numbers in the timings in order to keep the presentation simpler but also because (a) our emphasis is on scalability and not on small-scale speed gains—we consider small differences in timings, e.g., in the chart and eclipse benchmarks of Figure 4.5, to be negligible for our purposes; and (b) these constant overheads can be factored out—e.g., with minor engineering we could have incurred them only once per benchmark and not once per run of every introspective analysis variation.

Based on our experimental results, introspective context sensitivity achieves its goal: it offers a knob for users to select points in the scalability/precision spectrum. The tradeoffs of cost and precision exhibited by Heuristic-A and Heuristic-B are illustrative. Not only do these heuristics yield different options (more precision vs. more scalability) but they are also very consistent in their tradeoff, throughout multiple benchmarks and analysis flavors.

Finally, note that we used identical introspection heuristics (Heuristic-A and Heuristic-B) with the same constants (see Section 4.2) for all three context sensitivity flavors and for all benchmarks. This suggests that there are significant opportunities for further tuning: different heuristics can be used, the constants can be optimized, the constants or the heuristics can be adapted per-benchmark or per-context flavor. However, the goal of our experiments is not to squeeze out a few percentage points of speedup but to show that the simple idea of introspective context sensitivity can easily offer very useful tradeoffs in scalability and precision.

[2]Our current implementation has to export facts to disk and re-import them back in memory, as well as regenerate a program representation in order to re-run an analysis.

4.4 Summary

We introduced introspective context sensitivity: an approach to making context-sensitive analyses scale. The approach consists of defining an analysis with two separate kinds of context. Each program element is analyzed with one kind, selected based on external input. Then, by first running an inexpensive context-insensitive analysis, we can identify program elements that should be treated with a more precise context and others that should be treated less precisely to avoid an explosion in complexity. Our technique applies to any kind of context abstraction and yields scalability *à la carte*: the user can select a scalability profile and achieve it for a price in precision. As shown in our experiments, this price is not too steep. The precision loss of introspective context sensitivity can be minuscule (as is for call-site-sensitive analyses), while the scalability gain is substantial.

We believe that introspective context sensitivity is a big step forward in pointer analysis. It is not just an effective technique, but an effective technique that addresses the major current pain point in practical applications of points-to analyses.

Part II

Achieving Strong Soundness Guarantees

5. MUST-ALIAS ANALYSIS: LOGICAL MODEL

> Logic, my dear Zoe, merely enables one to be wrong with authority.
>
> *The 2nd Doctor* - Doctor Who

As previously mention at the beginning of Chapter 1, alias analysis is closely related to pointer analysis with the difference being that the main goal of alias analysis is finding aliasing among program expressions whereas the main goal of pointer analysis is to reason about the objects that program expressions may point to. In both case, one can be employed to support the reasoning of the other.

The vast majority of pointer analysis techniques that have appeared in the recent research literature (e.g., [9, 57, 69, 105, 150, 163, 167]) are *may*-analyses. That is, the techniques attempt to *over*-approximate an unattainable, fully precise result (recall Section 2.6). All possible aliasing expressions are guaranteed to be included in the outcome of a may-alias analysis. All possible abstract objects that may be referenced by a variable are included in the variable's may-point-to set. However, spurious inferences—which will never occur in program execution—may also be included in the analysis output.

In contrast, an *under*-approximate, *must*-analysis (or *definite*-analysis) is often desirable. A must-analysis computes aliasing or points-to relationships that are guaranteed to always hold during program execution, at the cost of missing some inferences. In practice, a must-alias analysis is more probable than a must-points-to analysis. Aliasing relationships are more generally provable from local inspection of the program text, whereas points-to facts are harder to establish in a conservative must- fashion. The difficulty is dual: First, the allocation sites of objects are often far from their use sites, making the establishment of must-points-to relationships unlikely. Second, must-points-to reasoning requires careful modeling of abstract vs. concrete objects. For instance, for techniques such as *strong updates* (i.e., replacing the value of an object field at a store instruction) it is not sufficient to know the abstract object that the base expression must-points-to, since the abstract object may conflate many concrete objects during program execution (e.g., in a loop), and only one of them will have its field updated.

A must-alias analysis is typically flow-sensitive, i.e., it computes information per-program-point, respecting the control-flow of the program (recall Section 2.4). It offers several applications:

- It is useful for optimizations—e.g., constant folding, common subexpression elimination, and register allocation;
- It can increase the precision of bug detectors (that traditionally have a high false-warnings rate): Nikolić and Spoto [110] report that a must-alias analysis significantly increases the precision of both a null-reference detector (46% fewer warnings) and a

non-termination detector (11% fewer warnings). Earlier work has reported similar benefits [98];

- It can be used as an internal component as part of a more complex analysis. For instance, must-alias results may enable an analysis to perform "strong updates" at instructions that modify the heap. Earlier work has used must-alias analysis to similar benefit [39, 67];

- It can be invaluable for better program understanding; the results of a must-inference are guaranteed facts, of immediate value to the human programmer.

To illustrate must-alias reasoning, consider the code example in Figure 5.1. Even at this size, inspecting the program requires human effort. The output consists of must-alias pairs: expressions that are guaranteed to point to the same object (denoted by "∼"). In this example, `a2.next` and `a1` form an alias pair after line 8. (Other alias pairs include {`a1.next` ∼ `null`} after line 7, {`a2.next.next` ∼ `a2`} after line 11, and more.) Alias pairs are established by direct variable assignments—which are plentiful in a compiler intermediate language, although less so in original source code—as well as heap stores and loads. Other aliasing relationships hold throughout the program. Establishing them often requires some inter-procedural reasoning—e.g., to see the aliasing effects of the constructor call on lines 8, 9, or 10. Constructors feature prominently in the example, since they are one of the best sources of must-alias information in a typical program.

A must-alias analysis has to report aliases only when they are guaranteed to hold, and needs to invalidate them on store instructions or method calls that may change the fields of objects pointed by sub-expressions in an alias pair.

```
1  class Node {
2      Node next;
3      Node(Node next) { this.next = next; }
4      void wrap() { next.next = this; }
5  }
6
7  void main() {
8      Node a1 = new Node(null);
9      Node a2 = new Node(a1);
10     Node a3 = new Node(null);
11     a1.next = a3;
12     a2.wrap();
13 }
```

Figure 5.1: Code snippet for illustrating must-alias reasoning.

For example, line 10 invalidates the alias pair {`a1.next` ∼ `null`}—regardless of whether new alias pairs are established, via inter-procedural reasoning. However, the analysis is sound (i.e., it remains a must-analysis—recall Section 2.7) if it also invalidates alias pairs for expressions involving `a2.next` or `a3.next`. The base specification of a must-alias analysis

has to integrate such soundness safeguards, while interplay with other analyses (e.g., a may-not-alias analysis) can lead to more inferences.

This chapter presents a simple declarative model of a must-alias analysis over access paths (i.e., expressions of the form "`var(.fld)*`"). The model underlies the implementation of must-alias analysis in Doop, in which must-alias analysis is employed as an enhancer of its standard array of may-analyses (e.g., in order to enable "strong updates"). The model is interesting in a few different ways:

- It is an instance of a flow-sensitive analysis in Datalog. As such, it introduces idioms and patterns also used in a multitude of other (current or future) analyses in Doop.
- The analysis is minimal, yet models the core features of a general must-alias analysis in a handful of declarative rules.[1] In this way, the analysis semantics are easily understood and can be further enhanced. The rules allow configurability and employ several techniques for conciseness and power.
- The use of context, in particular, is crucial: the analysis applies context variables, much like in traditional may analyses (e.g., [70, 141] and what was presented in previous chapters), yet uses the context highly unconventionally. Context is used as "fuel", to guarantee the "must" nature of the analysis: must-alias inferences are propagated inter-procedurally, with context extended for every call. When maximum context depth is reached, inferences cannot propagate any further.
- A major benefit of the analysis is its *incrementality*. In a well-specified must-alias analysis, soundness is not compromised if only a portion of the program-under-analysis or its libraries are available. This key element is emphasized in our declarative model. We control the program points where the full analysis applies and leverage context sensitivity to allow analysis of other program points. In essence, our analysis infers normal alias pairs for a user-selected core part of the program, and infers conditional, context-qualified alias pairs for other parts that interact with the core program.
- The analysis gives rise to several observations, concerning the representation of equivalence relations in a Datalog engine, and the need for implicit encoding of aliasing.

5.1 Logical Model

We demonstrate a minimal Datalog model of an inter-procedural must-alias analysis via changes to the existing model for context-sensitive, flow-insensitive points-to analysis algorithms presented in Section 2.12. As in previous chapters, the logical formalism of this model is very close to the core components of our actual analysis implementation.

But contrary to previous chapters, we need to model a flow-*sensitive* analysis and this dictates certain alterations on our input language. Mainly, we assume a static single assignment

[1] The analysis core presented here is the basis of a much larger, full-fledged (over 300 Datalog rules) must-alias analysis implementation in Doop.

(SSA) form on our intermediate representation. Recall from Section 2.5 that every local variable is assigned exactly once and for variables with multiple assignments in the original source code, the merging of their values is indicated by a *phi-node* in the SSA representation. For the purposes of static analyses that do not track path conditions, it is not relevant which of the two (or more) values is actually picked, but it is important (for flow-sensitive variants) that such merging points are indeed modeled accordingly.

Input domain. Figure 5.2 demonstrates the needed alterations in the domain of our input language (presented in Figure 2.4).

`A` is an access path of the form `V`.(`F`)*

Figure 5.2: Additions to the domain of our input intermediate language.

Input relations. Our input relations, shown in Figure 5.3, are similar to those presented in Figure 2.5 with the main difference that they also encode of the actual instruction they represent, in order to enable a flow-sensitive reasoning. Regarding invocations, we mainly focus on virtual calls (simplified to `Call` instead of `VCall`) and assume a program in a single-return form, for each method. In addition, the `Phi` relation captures phi-node instructions, where values of multiple "`from`" vars merge into local var `to`. The `Next` relation expresses directed edges in the control-flow graph (CFG) meaning that instruction `j` is a successor of instruction `i`. Relation `InMethod` encodes the obvious semantics of method `meth` containing instruction `i`.

The last two input relations are somewhat more advanced. `Resolved` is a predicate that can be computed by an external call-graph or may-point-to analysis: it holds variables that are determined to only point to objects with a unique dynamic type, so that virtual method calls are resolved. (Note that the form of the predicate is context-insensitive, yet the analysis that computes it may be context-sensitive, for increased precision—the contexts are merely projected out.) Finally, `RootMethod` is a predicate over methods, in order to start must-alias reasoning from a user-selected set of methods. As will be explained bellow, our analysis algorithm will venture beyond these root methods only to the extent that its context constructor allows.

Computed (output) relations.

Figure 5.4 shows the computed relations of our must-alias analysis. The first relation, `MustAlias`, is the main output of the analysis and is defined on access paths. The semantics are that access path `ap1` aliases access path `ap2` (i.e., they are guaranteed to point to the same heap object, or to both be `null`) right after program instruction `i`, executed under context `ctx`, provided that the instruction is indeed executed under `ctx` at program run-time. The two access paths are said to form an *alias pair*.

The other main computed relation represents intermediate results of the analysis. Relation `MustCallGraphEdge` holds information for fully-resolved virtual calls: invocation site `invo` will call method `meth` under the given (caller and collee) contexts.

```
Move(i: I, to: V, from: V)                  // i: to = from
Load(i: I, to: V, base: V, fld: F)          // i: to = base.fld
Store(i: I, base: V, fld: F, from: V)       // i: base.fld = from
Call(i: I, base: V, sig: S)                 // i: base.sig(...)
FormalReturn(i: I, meth: M, ret: V)         // i: return ret;

Phi(i: I, to: V, from1: V, ...)             // i: to = φ(from1, ...)
Next(i: I, j: I)
InMethod(i: I, meth: M)

Resolved(var: V, type: T)
RootMethod(meth: M)
```

Figure 5.3: The (altered) input Datalog relations describing the program under analysis.

```
MustAlias(i: I, ctx: C, ap1: A, ap2: A)
MustCallGraphEdge(invo: I, callerCtx: C, meth: M, calleeCtx: C)
```

```
AP(access path expression) = ?ap: A
PrimeAP(ap: A) = ?newAp: A
UnprimeAP(ap: A) = ?newAp: A
NewContext(invo: I, ctx: C) = ?newCtx: C
```

Figure 5.4: The core Datalog output relations and constructors of access paths and contexts.

Constructors.

We assume a constructor function `AP` that produces access paths. For instance, the expression "`AP(var.fld1.fld2) = ?ap`" means that the access path `?ap` has length 3 and its elements are given by the values of bound logical variables `var`, `fld1` and `fld2`. We shall also use `AP` as a pattern matcher over access paths. For example, the expression "`AP(_.fld) = ap`" binds the value of logical variable `fld` to the last field of access path `ap`. (`_` is an anonymous variable that can match any value.)

Additionally, we manipulate access paths with two functions, `PrimeAP` and `UnprimeAP`. `PrimeAP` takes an access path and returns a new one by "priming" the base variable of the original. `UnprimeAP` reverses this mapping. For instance, `PrimeAP("v.fld") = "v'.fld"`. `UnprimeAP` only applies to access paths with primed variables as their base—otherwise the rule fails to match. Priming and unpriming of access paths is done at method call and return sites, to mark access paths that arrive from callers. This is necessary for avoiding confusion of variables in recursive calls.

Similarly, we construct new contexts using function `NewContext`. The definition of this constructor serves to configure the analysis for different context settings, as discussed later. If `NewContext` does not return a value (e.g., because the maximum context depth has been reached), the current rule employing the constructor will not produce facts. The constant `All` is used to signify the initial context.

Constructors of access paths and contexts are much like other relations. In practical analyses, the space of access paths and contexts is made finite, by bounding their length. Therefore, all possible access paths and contexts could be computed prior to the analysis start and supplied as inputs. However, this is unlikely to be desirable in practice, for efficiency reasons, and is limiting in principle: by separating constructors, our model also allows analyses with unbounded access paths and contexts.

5.2 Analysis Logic

We have broken down our analysis model in four groups of rules. For conciseness, we have employed some syntactic sugar (also recall Section 2.11 regarding disjunction and negation), which straightforwardly maps to more complex Datalog rules:

- We use the shorthand `P*` for the reflexive, symmetric, transitive closure of relation `P`, which is assumed to be binary. For larger arities, underscore (`_`) variables are used to distinguish variables of a relation that are affected by the closure rule. Specifically, `MustAlias*(i, ctx, _, _)` denotes the reflexive, symmetric, transitive closure of relation `MustAlias` with respect to its last two variables.
- We introduce a for-all syntactic sugar that hides a Datalog pattern for enumerating all members of a set and ensuring that a condition holds universally.[2] An expression "$\forall$ `i`: `P(i)` $\longrightarrow$ `Q(i, ...)`" is true if `Q(i, ...)` holds for all `i` such that `P(i)` holds. Such an expression can be used in a rule body, as a condition for the rule's firing. Multiple variables can be quantified by a $\forall$. Variables not bound remain implicitly existentially quantified, as in conventional Datalog. However, the existential quantifier is interpreted as being outside the universal one. For instance, "$\forall$ `i, j`: `P(i, j, k)` $\longrightarrow$ `Q(i, j, k, l)`" is interpreted as "there exist `k, l` such that for all `i, j` ...".

Base Rules. Figure 5.5 lists six rules: one to initialize interesting analysis contexts and five for must-alias inferences. The former rule employs configuration predicate `RootMethod`. This predicate designates methods that are to be analyzed unconditionally: the inference is made under the special context value `All`. For a non-root method, aliasing inferences can only be made under a specific context, for which the method has been computed to be reachable. The results can then be used by the caller that produced that context. They are not, however, established as unconditional results (in an `All` context), which would be usable independently.

The above mechanism controls the application extent of the analysis. Recall that incrementality is a key benefit of a must-alias analysis. Therefore, it is desirable to be able to apply the algorithm as locally as the user may desire. The context mechanism is then used to

[2] Emulating universal quantification in Datalog requires ordered domains. In practice, this is not a restriction. An arbitrary ordering relation (e.g., by internal id of input facts as assigned by the implementation) can be imposed on all our domains.

explore other code, but only to the extent that such exploration benefits the root methods intended for analysis.

The next four MustAlias rules handle one instruction kind each: Move, Phi, Load, and Store. The Move rule merely establishes an aliasing relationship between the two assigned variables, at the point of the move instruction. The Phi rule promotes aliasing relationships that hold for all the right-hand sides of a phi-node instruction to its left hand side. The Load and Store rules establish aliases between the loaded/stored expression, base.fld, and the local variable used. Finally, the last MustAlias rule makes the MustAlias relation symmetrically and transitively closed.

```
Reachable(meth, All) ⟵ RootMethod(meth).

MustAlias(i, ctx, AP(from), AP(to)) ⟵
    Move(i, to, from),
    InMethod(i, meth), Reachable(meth, ctx).

MustAlias(i, ctx, ap, AP(to)) ⟵
    (∀ from: Phi(i, to, ..., from, ...) ⟶ MustAlias(i, ctx, AP(from), ap)),
    InMethod(i, meth), Reachable(meth, ctx).

MustAlias(i, ctx, AP(to), AP(base.fld)) ⟵
    Load(i, to, base, fld),
    InMethod(i, meth), Reachable(meth, ctx).

MustAlias(i, ctx, AP(from), AP(base.fld)) ⟵
    Store(i, base, fld, from),
    InMethod(i, meth), Reachable(meth, ctx).

MustAlias(i, ctx, _, _) ⟵ MustAlias*(i, ctx, _, _).
```

Figure 5.5: The core, base Datalog rules for a model must-alias analysis.

Inter-Procedural Propagation Rules. Figure 5.6 presents four rules responsible for the inter-procedural propagation of access path aliasing.

The first rule continues the handling of program instructions with a treatment of Call. At a Call instruction, for method signature sig over receiver base, if base has a unique (resolved) type, then the method is looked up in that type, a MustCallGraphEdge is inferred from the invocation instruction to the target method and the method is also marked as Reachable with a callee context computed using constructor NewContext. Recall that the NewContext function may fail to return a new context (e.g., because ctx has already reached the maximum depth and calleeCtx would exceed it) in which case the rule will not infer new facts.

The other three rules handle aliasing induced at a method invocation site. Despite their rather daunting form, the rules are quite straightforward. The first states that, at the first

instruction of a called method, the formal and actual arguments are aliased. In combination with other rules (discussed next, under "Access Path Extension") this is sufficient for transferring all alias pairs from the caller to the callee. The actual argument is "primed" appropriately, to mark that it is received from a caller. For instance, if the analyzed program contains a call "`foo(x)`" to a method defined as "`void foo(Object y)`", the rule will simply infer that `x`' and `y` are aliased. The rule infers the same aliasing for the base variable of the method call and the pseudo-variable `this` inside the receiver method. (Note how the first instruction of the called method is computed as the only instruction in the method that has no CFG predecessors: $\forall$ `k` $\rightarrow$!`Next(k, firstInstr)`. This convention is assumed to hold for our input intermediate language.)

The third rule similarly identifies the first instruction of a called method. It then propagates to it all alias pairs that hold *after all predecessor instructions*, `j`, of the calling instruction, `i`. The base variables of the alias pairs are "primed", as appropriate, to denote that they come from the caller.

The fourth and final rule performs the inverse mapping of access paths from a return instruction to the call site. For alias pairs to propagate back (to the caller, with context `callerCtx`), they need to hold either in the appropriate context (`calleeCtx`, which matches `callerCtx` in the call graph), or unconditionally, i.e., with context `All`. Access paths are "unprimed" when propagating to the caller. Note that this implies that local alias pairs (e.g., among local variables of the callee) do not propagate to the caller.

Crucially, the handling of a method return is the *only* point where a context can become stronger. `MustAlias` facts that were inferred to hold under the more specific `calleeCtx` are now established, modulo unpriming, under `callerCtx`.

Access Path Extension. Figure 5.7 contains a straightforward, yet essential, rule. This rule allows access path extension: if two access paths alias, extending them by the same field suffix also produces aliases. It is important to note that the constructor `AP` is not used in the head of the rule, thus the extended access paths are not generated but assumed to exist. Therefore, the rule does not spur infinite creation of access paths.

This powerful rule is responsible for much of the simplicity of our must-alias analysis specification. For instance, recall how earlier we handled the mapping of actual to formal method arguments quite simply: we merely added an alias between the (primed) actual argument variable and the formal argument. It is the access path extension rule that takes care of also generalizing this mapping to longer access paths whose base variable is the actual argument of the call.

Frame Rules: From One Instruction To The Next. The rule in Figure 5.8 determines how must-alias facts can propagate from one instruction to its successors. The rule simply states that all aliases are propagated if the instruction is not a store or a call. (Because of SSA, access paths cannot be invalidated via move instructions.)

Comments. The model we just presented is carefully designed to encompass a minimal, highly-compact but usefully representative must-alias analysis. There are several extensions that can apply, but all of them are analogous to features shown. For instance, we are missing

```
MustCallGraphEdge(i, callerCtx, toMeth, ?calleeCtx),
Reachable(?calleeCtx, toMeth) ⟵
    Call(i, base, sig),
    InMethod(i, inMeth), Reachable(inMeth, callerctx),
    Resolved(base, type), LookUp(type, sig, toMeth),
    NewContext(i, ctx) = ?calleeCtx.

MustAlias(firstInstr, calleeCtx, ?ap1, ?ap2) ⟵
    MustCallGraphEdge(i, _, toMeth, calleeCtx),
    InMethod(firstInstr, toMeth), (∀ k ⟵ !Next(k, firstInstr)),
    ((FormalArg(toMeth, n, toVar), ActualArg(i, n, fromVar)) ;
     (ThisVar(toMeth, toVar), Call(i, fromVar, _))),
    PrimeAP(AP(fromVar)) = ?ap1, AP(toVar) = ?ap2.

MustAlias(firstInstr, calleeCtx, ?ap1, ?ap2) ⟵
    MustCallGraphEdge(i, callerCtx, toMeth, calleeCtx),
    InMethod(firstInstr, toMeth), (∀ k ⟵ !Next(k, firstInstr)),
    (∀ j: Next(j, i) ⟶ MustAlias(j, callerCtx, callerAp1, callerAp2)),
    PrimeAP(callerAp1) = ?ap1, PrimeAP(callerAp2) = ?ap2.

MustAlias(i, callerCtx, ?ap1, ?ap2) ⟵
    MustCallGraphEdge(_, callerCtx, toMeth, calleeCtx),
    FormalRet(retInstr, toMeth, _),
    (MustAlias(retInstr, calleeCtx, calleeAp1, calleeAp2) ;
     MustAlias(retInstr, All, calleeAp1, calleeAp2)),
    UnprimeAP(calleeAp1) = ?ap1, UnprimeAP(calleeAp2) = ?ap2.
```

Figure 5.6: Datalog rules for inter-procedural propagation of alias pairs.

```
MustAlias(i, ctx, ?ap3, ?ap4) ⟵
    MustAlias(i, ctx, ap1, ap2),
    AP(ap1.fld) = ?ap3, AP(ap2.fld) = ?ap4.
```

Figure 5.7: Datalog rule for access path extension.

```
MustAlias(i, ctx, ap1, ap2) ⟵
    !Store(i, _, _, _), !Call(i, _, _),
    (∀ j: Next(j, i) ⟶ MustAlias(j, ctx, ap1, ap2)).
```

Figure 5.8: Datalog frame rule propagating alias pairs from one instruction to the next.

a rule for propagating back to the caller complex access paths (i.e., of length greater than 1) that are based on the formal return variable. Similarly, store or call instructions do not invalidate aliasing between local variables—an extra rule could allow further propagation. Furthermore, it is not always necessary for an alias pair to hold in all predecessors: it could hold in one and others may be dominated by the instruction and not invalidate the alias pair. Our actual implementation contains the handling of such cases, but these complexities do not affect the discussion of our model.

These rules liberally employ negation. They establish that must-alias facts are propagated if some disabling conditions do *not* hold. Therefore, for a full-fledged analysis, the rules need to be enriched with more preconditions, to cover all different kinds of program instructions that may invalidate access paths.

5.3 Discussion

There are several parts of the model and its implementation that are worth emphasizing.

Access Path Creation. Our access path constructor, `AP`, hides the details of the space of access paths and their construction. There are several different policies that an analysis can pick. In theory, we could up-front populate the entire combinatorial space of "`var(.fld)*`") up to a certain depth. However, the large sizes of the domains of variables and fields make this prohibitive. An efficient way to create access paths lazily (also used in our full implementation) is to initially generate all primitive access paths (variables and variable-single-field combinations) that appear explicitly in the program text, and then close the set of access paths by employing the rule:

```
AP(ap2.fld) = ?newAp ⟵ MustAlias(_, _, ap1, ap2), AP(ap1.fld) = _.
```

Note that one use of the constructor `AP` is in the head of the rule (thus generating new access paths on the fly) and one in the body (checking that the access path already exists). That is, extended access paths (`base.field`) are generated only if their base access path is found to be aliased with another path, which already exists with the `field` suffix.

Context Sensitivity in Must-Alias. The use of context in our must-alias analysis is subtle. Context in a pointer analysis is used to distinguish different dynamic execution flows when analyzing a method. That is, the same method gets analyzed once per each applicable context, under different information. The context effectively encodes different scenarios under which the method gets called, allowing more faithful analysis in the specialized setting of the context.

Yet, the use of context in a must-alias analysis has been explored very little in the past. The reason is that there is little benefit to be gained from specializing incoming information (i.e., alias pairs) for a must-alias analysis. Pairs of aliasing access paths already offer a symbolic summary of a function's behavior, so that there is less need to analyze the function separately for different contexts: the access paths can be merely returned to the caller and they will be specialized there. Our analysis model employs context to transmit alias pairs

from a caller to a callee, yet qualify them with the context identifier to which they pertain. This enables producing more alias pairs, however, their validity is conditional on the context used.

Still, this conditional information can be used for further inferences. For instance, MustAlias inferences could be combined with allocation instructions (allocation sites can be viewed as global access paths) in order to determine, when possible, which objects an access path must point to. In turn, this can inform virtual method resolution, which our current rules (Figure 5.6) only perform via a may-analysis by use of relation Resolved. In this way, specialized alias relations for a given context can result in more inferences (since a method call may now have a known target). These inferences can be propagated back to the caller, where they hold unconditionally. (Recall that the rules handling returns can remove access path assumptions.)

Generally, the use of a deeper context in a must-analysis can extend its reach (or recall as discussed in Section 2.8), allowing *more inferences*, i.e., a larger result, whereas deeper context in a may-analysis it results in *more precision*, i.e., a smaller result.

What can our context be, however? In previous chapters, or typical context-sensitive pointer analyses in the literature, a variety of context creation functions can be employed. There are context flavors such as call-site sensitivity [137, 140], object sensitivity [105, 106], or type sensitivity [142]. Our `NewContext` constructor (employed at method calls) could be set appropriately to produce such context variety. However, the current form of our rules restricts our options to call-site sensitivity, with potential extra information adding to, but not replacing, call sites. Given the signature of constructor `NewContext` the assumption is that the new context produced uniquely identifies both invocation site `invo` and *its* context, `ctx`. Effectively, if `NewContext` produces a `?newCtx` at all, it can do little other than push `invo` onto `ctx` and return the result.

The analysis then propagates MustAlias pairs from (all predecessors of) call site `invo` under context `ctx` to the first instruction of a called method, `toMeth`, under context `?newCtx`. Thus, `?newCtx` should be enough to establish that these inferences *must* hold. There is no room for conflating information from multiple execution paths (i.e., callers and calling contexts).[3]

The requirement that `NewContext(invo, ctx)` produce contexts that uniquely identify both `invo` and `ctx` means that context can only grow from an original source in our analysis. Consider a set of three methods, `meth1`, `meth2`, and `meth3`, each calling the next. If we allow `NewContext` to produce contexts that are stacks of invocation sites, `i`, each starting with All and growing up to depth 2, then starting from `meth1` we will propagate its aliases to `meth2`, which will propagate the resulting combined aliases to `meth3`. The propagation will stop there, i.e., the aliases of `meth1` cannot influence inferences for callees of `meth3`. However, `meth3` (assuming it is included in the root methods) will itself also be analyzed with a context of All, allowing its own aliases (independently derived from those of `meth1` or `meth2`) to be a source of a similar propagation.

[3]One could imagine doing so under the premise that all such calling contexts agree on the aliases they establish at the beginning of the callee function. However, this is unlikely to arise often in practice.

Representation of Equivalence Classes. Relation MustAlias encodes equivalence classes on access paths. Datalog inherently has no such notion and any attempt to compute a must-alias relation has to explicitly encode all aliasing pairs. E.g., if variable `v1` is an alias for variable `v2`, and `v2` of variable `v3`, we have to explicitly record the following pairs: {`v1` ∼ `v2`}, {`v2` ∼ `v1`}, {`v2` ∼ `v3`}, {`v3` ∼ `v2`}, {`v1` ∼ `v3`}, {`v3` ∼ `v1`}. This effect is exacerbated for longer access paths.

In theory, this redundancy will greatly hinder performance. In practice, it is often affordable because of keeping access paths short and computing must-alias information where needed, due to the locality of such information. (Aliasing pairs rarely propagate much deeper than the point in code where they were first established.) The analysis is fully modular and can be applied to any subset of the program code. Still, future work should address this shortcoming in the general setting of Datalog computation of equivalence relations.

5.4 Summary

The literature on must-alias analyses is sparse and the distance of specification to implementation is typically large. In our literature survey we have not found a single must-alias analysis publication that concretely refers to another and shows how its approach differs. Thus, we believe that our declarative model can offer a reference point for future work.

We believe that our model is clear yet concrete enough to spur further development and a better understanding of the comparative features of different must-alias analysis algorithms. In practical terms, must-alias analysis is valuable and woefully under-exploited in the literature. Our experiments show concrete value for (human) program understanding and (automatic) optimization.

6. MUST-ALIAS ANALYSIS: DATA STRUCTURES

> I love humans. Always seeing patterns in things that aren't there.
>
> *The 8th Doctor* - Doctor Who

The previous chapter presented an elegant minimal model of a must-alias analysis in the declarative setting of Datalog (and the Doop framework). Although attractive in its expressiveness, if the model is to be implemented as is it will incur serious performance penalties. As mentioned in the discussion Section 5.3, an important prerequisite for a must-alias analysis is that of encoding equivalence classes on access paths. Datalog inherently offers no feature to support such notion and thus, in the previous chapter, we had to resort to an explicit representation of all alias pairs.

In this chapter, we present a data structure that can dramatically speed up the performance of must-alias analysis. The insights behind the data structure are quite general. First, the fact that must-alias sets are equivalence classes and the need to avoid explicitly computing each alias pair. In contrast, an optimized implementation will encode aliasing implicitly: as membership in the same sub-structure. This is a technique also employed in past static analysis approaches in different settings (e.g., in the use of union-find trees in Steensgaard-style [153] points-to analysis). Additionally, aliasing can be implicitly extended to longer access paths and this inference should be readily computable in the course of the analysis. For instance, if two program expressions `x` and `y.next` are aliases, then so are all their extensions (e.g., `x.prev` and `y.next.prev`). Such "derivative" relationships should be represented compactly. In our data structure, we represent complex program expressions implicitly until expansion is needed and up to the extent that must-alias information exists for them.

Our data structure is effectively a symbolic abstraction of the program's heap—as a directed graph. We invent for each variable a graph node: an abstract object that represents "the object that the variable points to". Although several abstractions of the heap have appeared in the literature, ours is distinguished by several elements—e.g., a mere "load" operation introduces a new abstract object. An abstract object in our structure represents at most one concrete object, unlike traditional abstractions that map multiple concrete objects to one abstract. Whenever a must-alias inference is made, the corresponding abstract objects are merged: the two abstract objects have to correspond to the same concrete one. Access paths are represented implicitly, as regular paths that follow object fields through our symbolic heap. All operations over the graph arising during a must-alias analysis (especially the intersection of graphs) are performed highly efficiently.

We implement the data structure in two settings: imperatively, in Java, with destructive updates (upon aliasing, abstract objects are collapsed together) and purely functionally (upon aliasing, abstract objects are related in an associative structure). The latter is suitable for a declarative implementation, in the Datalog language. We show that the data structure yields large performance improvements compared to an explicit representation of alias pairs. The imperative version achieves a speedup of up to **two orders** of magnitude, with the

declarative implementation nearly matching it in most cases. As a result, the running time of a realistic must-alias analysis becomes small—a few tens of seconds for large benchmarks and the full Java library.

Overall, in this chapter we:

- Describe the primitive operations (e.g., set intersection) that a must-alias analysis needs to perform.
- Present an efficient data structure for representing must-alias analysis inferences and efficiently encode operations over that structure.
- Apply the new data structure on an must-alias analysis implemented in an existing framework and quantify the benefits in different implementation settings.

6.1 Must-Alias Analysis Needs

Before presenting our optimized data structure for a must-alias analysis, it is important to ponder upon the properties of such an analysis and the algorithmic needs that arise if one is to implement it with performance in mind. The previous section offered a minimal yet representative Datalog model of a must-alias analysis. A great benefit of a distilled declarative model is the ability to reason about its properties. Directly executing the Datalog rules of Section 5.1 is a realistic proposition. Indeed, the must-alias analysis in the Doop framework is well-captured by the model. However, several inefficiencies arise from the rules. We next examine the model and discuss how it leads to an optimized data structure, and its associated algorithms, for must-alias analysis.

Representing Equivalence Relations. We have already commented on the effects of a naive implementation of the must-alias relation in previous sections. For instance, four program expressions aliasing with each other would require the explicit representation of twelve alias pairs (ignoring the trivial pairs of each expression aliasing with itself) in order for the relation to stay reflexive, symmetric and transitively closed.

Since must-alias is an equivalence relation, it induces a partitioning of the space of access paths: every access path can only belong in one alias (equivalence) class. This means that we can represent the contents of each class compactly, by grouping together all aliased access paths. An access path can denote that it belongs in a certain alias class, e.g., by having a unique identifier, or by being a member in a linked data structure. The goal is to represent an alias class using linear space and time (in the number of its access paths) instead of enumerating all pairs of aliased access paths (and taking up quadratic space and time).

It is important to note that the concrete (i.e., dynamic) "alias" relation is also an equivalence relation, but most *may*-alias relations in the static analysis literature are *not*. For instance, in a typical subset-based pointer analysis, access path `ap1` may-alias `ap2` by pointing to the same abstract object (among others). Similarly, `ap2` may-alias `ap3`. However, it may not be the case that `ap1` and `ap3` may-alias: the common elements in the points-to sets of `ap1`

and ap2 may not be among the common elements in the points-to sets of ap2 and ap3. This highly influences all data structure operations. Notably, the main algorithm that we will describe (intersection of data structures when joining control-flow paths) is not present in a may analysis.

Extending Access Paths. A less obvious observation concerns the representation of aliasing in extended access paths. A naive implementation would, once again, have to represent aliases explicitly. For instance, two aliased program variables `x` and `y` will also induce alias pairs `x.f` and `y.f`, as well as `x.g` and `y.g`, `x.f.g` and `y.f.g`, etc., up to the maximum access path length (and modulo valid field accesses). This is an exponential number, $\Omega(c^k)$, of aliased access paths, for c valid fields and access path length limit of k. The access path length can be easily limited (e.g., $k = 3$ does not restrict the vast majority of useful alias inferences), so the burden is not insurmountable, but it can still be significant.

Ideally, we would like a data structure that only explicitly maintains the initial aliasing relationship and can implicitly derive the aliasing of all extended access paths.

Algorithms. Once we have a data structure that satisfies the above requirements, what algorithms should we implement efficiently on this data structure? The basic algorithms behind most must-alias analysis inferences are straightforward. The analysis needs to copy alias classes (equivalence classes), add a single access path, remove a single access path, or rename variables in a set of alias classes. The only case that introduces some complexity is the one dealing with multiple predecessors of an instruction in the control-flow graph.

In a must-alias analysis setting, in order for an alias pair to be valid at the instruction where multiple control-flow paths meet, it should hold in each path. The operation we need here is that of taking the intersection of alias classes from many different sets (one for each predecessor instruction). For instance, in the code snippet of Figure 6.1, on line 19, we can infer that {`a2.member` ∼ `b1`} since it holds in both paths, but not that {`a2.next` ∼ `a1`}.

```
1  class A {
2      A next;
3      B member;
4      A(A next, B member) {
5          this.next = next;
6          this.member = member; } }
7  class B {
8      A container;
9      B(A container) {
10         this.container = container; } }
11 void main(String[] args) {
12     B b1 = new B(null);
13     A a1 = new A(null, b1);
14     A a2;
15     if (args != null)
16         a2 = new A(null, b1);
17     else
18         a2 = new A(a1, b1);
19     b1.container = a2;
20 }
```

Figure 6.1: Code snippet to illustrate certain points of our algorithms.

More than a Union-Find. Before describing our data structure in full detail, it is important to note how it differs from an implementation of the well known *Union-Find* data structure. First, it extends on the idea of keeping sets of equivalent elements, by connecting equivalence classes in order to form complex access paths in a compact way. Secondly, it allows for deletions of elements from an equivalence class, potentially producing sets without

elements (but significantly different to an empty set—as explained later). Thirdly, and more importantly, our data structure introduces an additional operation on equivalence classes—that of *intersection*—in supplement to that of *union*.

Notably, in contrast to a typical data structure for equivalence classes, unions of (non-singleton) equivalence classes do not arise: if an expression is newly aliased with others, it is because it is no longer aliased with its previous aliases. The corresponding operation is a single access path addition and removal (from a different class). Conversely, intersections of alias classes are central to our structure.

6.2 An Optimized Data Structure and Algorithms

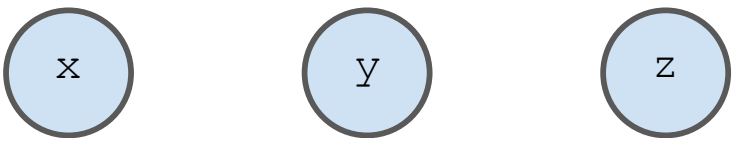
Based on the above requirements, we propose an *alias graph* data structure (and associated algorithms) for representing all alias sets of access paths that hold at a certain program point. In a typical must-alias analysis, a program point is a possibly context-qualified instruction. Each such *instruction-and-context* combination maintains an alias graph and the analysis updates it until fixpoint. An updated alias graph depends on the earlier graph for the same program point, on the graphs of its predecessor instructions, and on the current instruction's semantics.

We begin with a description of the easier case: how the current instruction affects the alias graph. This will also help illustrate the data structure.

The intuition is that an alias graph abstractly represents local variables and the heap, with abstract objects as placeholders for concrete objects. Nodes (abstract objects) are alias classes, edges are field-points-to relationships. Every abstract object, however, corresponds to (at most) a single concrete object at the current program point: our data structure is isomorphic with a part of the concrete heap. This property is true only because the data structure represents definite (must) aliasing.

We illustrate with simple examples. It is worth noting once again that every program instruction will maintain a different alias graph. The following examples focus on the situation at a certain instruction.

All variables conceptually begin with their own node in the graph. (In practice, such nodes need not be represented, unless connected to others.) The node represents "the object that the variable points to at this program point".



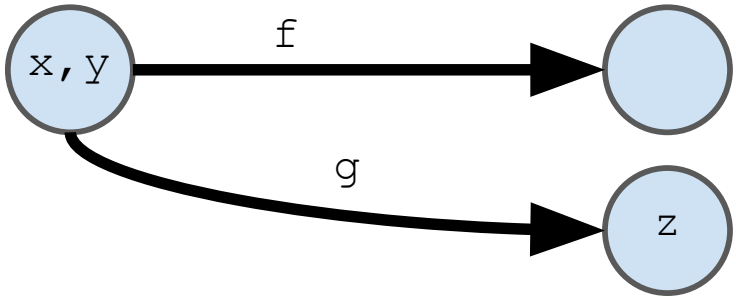
Aliasing can be induced by various program operations (e.g., `MOVE`, `LOAD`, and `STORE`), as seen in our earlier model. Since we are interested in must-alias, two aliased variables have to point to the same object—their nodes can be merged if a `MOVE` instruction, `x = y`, is encountered:

This collapsing of nodes is responsible for compact encoding of equivalence relations: two variables are computed to be aliases iff[1] they belong in the same node of the alias graph.

If the next instruction is a STORE, `x.f = z`, the previous graph will get propagated—i.e., copied. Subsequently, the STORE will add an edge to the graph, signifying that the field, `f`, of the object pointed by `x` will point to the object that `z` points to:

A subsequent LOAD operation, `z = y.g`,[2] will inherit the alias graph of its predecessor and will modify it. Variable `z` is removed from its old node (`z` no longer points to this abstract object), a new node for `z` is created, and the nodes are linked, to indicate that `z` now points to the same object as `y.g`. The empty, former node of `z` will be garbage collected if no other paths can reach it in the alias graph.

On the other hand, when an access path can still reach the empty node, the empty node provides useful information. It represents "the object that an access path points to at this program point". Empty here doesn't describe the lack of information—just the lack of a *local variable* pointing to this abstract object.



The LOAD operation shows that our alias graph, although intended to abstractly represent a real heap, behaves quite differently: a load from a field can introduce new objects, as well as update fields of existing objects.

Generally, the alias graph captures compactly all aliasing relationships among access paths. Maintaining the graph across program instructions is simple, as in the above examples.

[1] The statement refers to the must-alias analysis results, not to actual aliasing during program execution.

[2] The example sequence of actions described is contracted for brevity. Our implementation works on a static single assignment (SSA) intermediate form, so this exact scenario will never arise, since `z` has had its value read earlier and its single assignment has to dominate its use.

Graph manipulation merely has to observe some invariants:

- Two variables are in the same graph node iff the analysis reports them to be aliased. (Since alias classes are disjoint, the variables in different nodes are also disjoint.)
- A path in the graph represents a set of access paths, starting from a non-empty node (that denotes the base variables of the access paths) and extended with the field labels along the path's edges. If two paths in the graph reach the same node, all access paths they represent must be aliased.

For illustration, consider Figure 6.2, which shows the alias graph after line 19 of our code example in Figure 6.1.

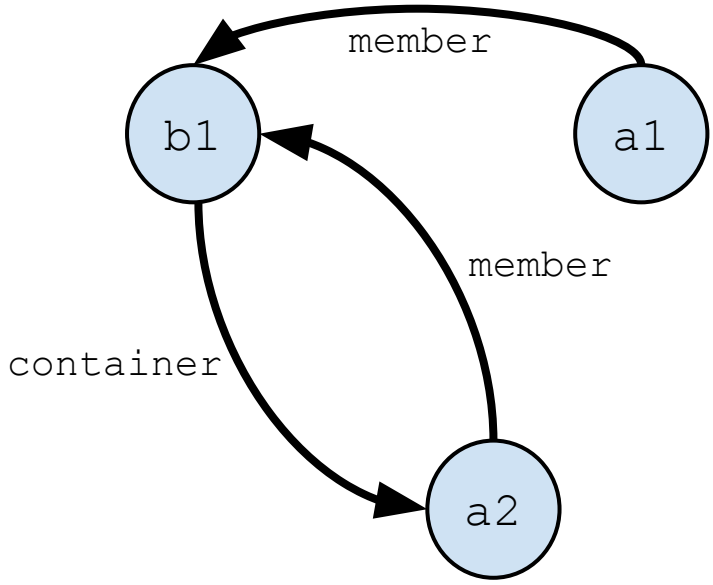


Figure 6.2: Example alias graph data structure.

The graph concisely represents a set of alias relationships that hold at that program point: {`b1.container` ∼ `a2`}, {`a1.member` ∼ `b1`}, and {`a2.member` ∼ `b1`}. An infinite number of other alias pairs are represented implicitly: {`a1.member` ∼ `a2.member`}, {`a1.member.container` ∼ `a2`}, {`a1.member.container.member` ∼ `b1`}, {`a1.member.container.member` ∼ `a1.member`}, etc.

Overall, the alias graph satisfies both of our requirements of Section 6.1 for an efficient representation. Equivalence relations are represented compactly: an alias class with n members does not need $O(n^2)$ space and time for its computation. Instead, it is represented implicitly, as all the variables in a node ($O(n)$ space) and all alias graph paths that can reach a node. Similarly, long (and even infinite) access paths are represented implicitly as graph paths. The implicit representation is sufficient for any specific queries (i.e., "are two given program expressions aliases?") and for subsequent aliasing computations, per the algorithms we detail next.

6.2.1 Main Algorithms

Most of the required must-alias analysis actions (per the discussion of Section 6.1) over our data structure are straightforward, consisting of copying, additions and removals of variables and edges, and variable renamings. Standard mappings for efficient indexing are required: each target of a directed edge needs to be able to quickly retrieve its source, and each program variable needs to quickly map to the node in which it appears.

For instance, according to the earlier definition of the data structure, finding all aliases of an access path is simple (but requires a transitive computation—our graph is a condensed representation of alias classes):

Algorithm: `all-aliases(ap)`

- Find the node for the base variable of access path `ap`, traverse in the forward direction the labeled edges that match each of the fields of `ap` to reach a target node.
- Any graph path that reaches the same node corresponds to an aliased access path, from a base variable adding the fields labeling the edges. (I.e., traverse $k-1$ directed edges backwards to find access paths of length up to k.)

For instance, in Figure 6.2, we can find all aliases of length 3 of access path `a2.member` by traversing edge `member` from node `a2` (thus reaching the node containing `b1`) and finding all paths of length 2 that can reach the same node, also including the variable(s) in the starting node of the path (e.g., `b1.container.member`).

The more interesting algorithm, as suggested earlier, is that of intersecting alias graphs—necessary for merging alias information from predecessor instructions. This is easy to see as a repeated intersection of two graphs (which is then iterated by intersecting a third with the result, then a fourth, and so on). Note that the graphs do not need to contain a single connected component.

Algorithm: `intersect(g1, g2)`

- The domain of possible nodes for the result of the intersection is the cartesian product of nodes of `g1` and `g2`. For every two nodes i, j of `g1` and `g2`, respectively, node (i, j), if it exists in the intersection result, will contain the intersection of the variables of i and j.
- Nodes are materialized incrementally, according to the rules below.
  1. For every two nodes i, j of `g1` and `g2`, if the intersection of the variables of i and j is non-empty, add to the intersection result a new node (i, j).
  2. (Repeatedly) If node (i, j) exists in the intersection result, then for every label f, if `g1` has an edge $i \to k$ with label f, and `g2` has an edge $j \to l$, also with label f, then add to the intersection result (if not already present):

- a node (k, l) (possibly empty);
- an edge $(i, j) \rightarrow (k, l)$ with label f.

Note that the first step is of linear complexity in the number of nodes, since empty nodes can be eagerly skipped and indexing from a variable to the, up to one, node that may contain it in a different graph is constant-time.

The algorithm considers all possible pair-wise node combinations and all possible edges out of node intersections. It maintains the property that any aliasing relationship (either variables belonging in the same node, or paths reaching the same node) in the result also exists in both input alias graphs.

Notably, the intersection of two alias graphs can produce nodes with empty variable sets, due to the second step of the algorithm. Empty nodes with no in-edges can be eliminated eagerly. Empty nodes with in-edges are meaningful in the output and need to be maintained. To illustrate, consider the example in Figure 6.3.

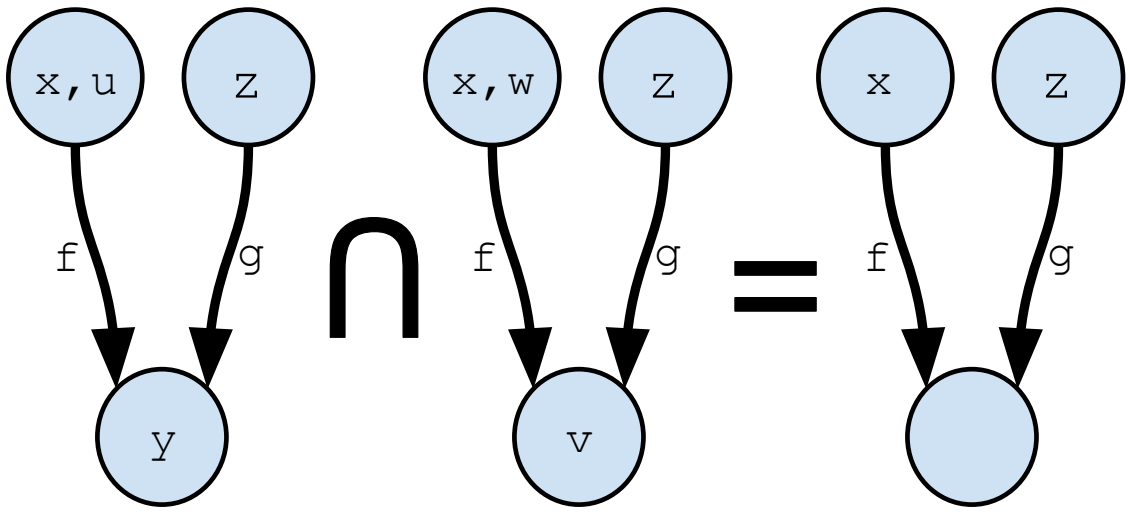


Figure 6.3: Intersecting alias graphs.

In this case, the empty note denotes that access paths `x.f` and `z.g` are still aliased in the intersection alias graph, even though they are no longer aliased with any single-variable access path.

For upper bounds n, v, e in the number of nodes, variables, and edges in the input alias graphs, respectively, the algorithm has a running time asymptotic bound of $O(n + v + e)$, i.e., linear in all quantities, if one assumes a practically constant-time indexing scheme from a variable to its node. (***Proof sketch***: Non-empty nodes are fewer than variables and only linear cost is incurred when combining non-empty nodes pair-wise, since each node has a distinct set of variables, used to index into any node that may intersect that set in the other alias graph. Empty nodes arise in the result and are only examined in the input if there is an edge into or out of them, therefore their number is below e. The number of edges in the output is at most that in the input—taken as the union of both input graphs).

Empty nodes with no in-edges are only one instance of nodes that no longer encode useful access path aliasing. Such nodes should be garbage-collected for maximum efficiency, producing a normalized input. The node collection algorithm is as follows:

Algorithm: `gc(g)`

- Any node in `g` containing a single variable and with no incoming or outgoing edges is eliminated.
- Any node in `g` containing no variables and with either zero in-edges or one in-edge and zero out-edges is eliminated.

In all cases covered by the above algorithm, the node does not encode alias pairs that would disappear with the node's removal: either there are no two paths (or variable names) that reach the node, or the node does not extend other paths beyond the implicit extension with the same field names that is already captured by the data structure.

6.2.2 Use in Practice

Having described the individual steps of an analysis using a must-alias data structure, we turn our attention to how it is used in the context of a realistic analysis. Consider the must-alias analysis of the Doop framework, discussed in Chapter 5. This computes a `MustAlias(i, ctx, ap1, ap2)` relation, i.e., aliased access paths at each program point and calling context. That is, each instruction-and-context combination is associated with an alias graph. Initially, conceptually every possible variable has its own node. (Implementation-wise, this is represented as an empty graph, requiring no initialization.) Every program instruction is visited and its alias graph is updated based on the instruction semantics and the operations that we described earlier. Specifically, every variable-aliasing instruction merges nodes (creating them if they only existed implicitly, i.e., they were single-variable nodes), every load and store instruction creates nodes and/or edges, every instruction also integrates the alias graphs of its predecessors, intersecting them if the instruction is a control-flow merge point. The visit order is not important for correctness, though it might affect performance (i.e., the number of steps needed before convergence is achieved).

At a call instruction, the analysis creates a new instruction-and-context pair (unless it exists already) for the first instruction of the callee method and the given context. Maximum context depth is a parameter of the analysis and if reached the call instruction will not be further analyzed. This is a sound approach for a must-alias analysis, since the aim is to compute an *under*-approximation of the alias relationships that are guaranteed to hold. With our data structure, the alias graph at the call site is copied to the first instruction in the callee method and then the usual operations are applied.

The above are repeated until a fixpoint is met. At any given alias graph, the number of non-empty nodes is bounded by the number of local variables in the program text. Empty nodes can arise but `gc` ensures that the intersection operation, where merging of states takes

place, can never produce more modes than the union of its inputs graphs: if an empty node is kept, it is because an incident edge existed in the input, hence a corresponding node appeared in the input.

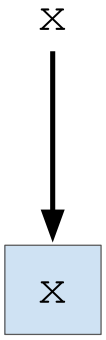
6.2.3 Declarative Implementation

As discussed earlier, the alias graph data structure can be employed in a must-alias analysis by maintaining alias graphs per-program-point (i.e., per context-qualified instruction), and updating them (to incorporate information from their predecessor instructions) until fixpoint.

The data structure description we have seen so far considers this update to be a destructive operation. For instance, after a `Move` instruction, we saw the nodes of two variables getting merged. Similar merging can be induced by information that the analysis discovers while it is executing (i.e., not directly induced by the semantics of the current instruction)—e.g., propagated from predecessors.

We have also designed and implemented a declarative/purely functional version of the data structure. The main reason for the declarative implementation has been fairness in experimental comparisons. We will compare our optimized implementation against the must-alias analysis of the Doop framework, implemented in Datalog. Thus, it is desirable to also implement a, perhaps not as optimal, declarative version of the data structure in Datalog. This will allow us to isolate the effect of destructive updates from the inherent properties of the data structure.

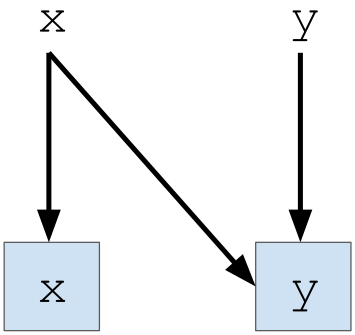
In the declarative version of the structure, aliased abstract objects are not merged, but instead associated with each other. Schematically, we can consider that each variable has its own node and points to at least one abstract object—at first the abstract object signifying "whichever object the variable got assigned to" (at its single-assignment site):



(For the declarative implementation, we will represent abstract objects as squares, to avoid confusion.)

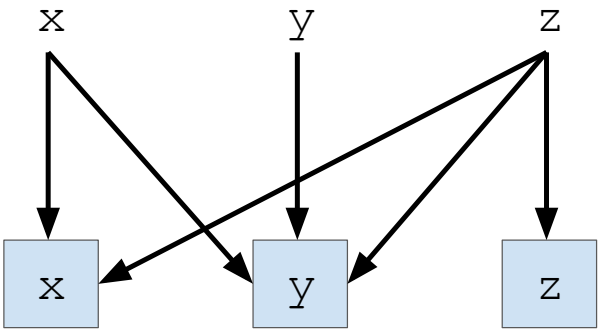
Since we no longer have destructive updates, variables can point to multiple abstract objects. For instance, after a `Move` instruction, `x = y`, we have:

As before, however, a variable can really point to a single concrete object. Therefore, the



two abstract objects that `x` points to have to be different abstractions of the same concrete object—their equivalence is encoded, but implicitly. It is easy to check that variables `x` and `y` are aliased, in the usual sense: they both point to the same abstract object.

If the next operation is another `MOVE`, `z = x`, variable `z` is made to point to both abstract objects that `x` points to:



Thus, the declarative data structure is less efficient than its imperative counterpart: a quadratic representation of an alias set cannot be precluded, and depends on the order of alias-inducing instructions. In our example, we could then assign `z`, the variable with the most out-edges, to a new variable, `w`, then assign `w` (which now has the most out-edges) to a new variable, and so on. In practice we expect that this effect will be mitigated. If the next instruction assigns `y` to a new variable, `u`, then `u` receives only a single extra edge, maintaining a more compact representation of the alias set. The cost, much as in the imperative structure, is that the alias set is not fully explicit and requires a transitive closure computation to be materialized.

6.3 Evaluation

We implemented an analysis that functionally matches the one present in the DOOP framework that is implemented in Datalog (whose core ideas are aptly captured in the model of the previous chapter). In contrast, our implementation is in Java, since our optimized

alias graph data structure has imperative features in its full form. Finally, to also eliminate language-level factors of the Datalog-vs-Java implementations, we also produced an optimized Datalog implementation, based on our purely functional data structure.

The three implementations are functionally equivalent, with very minor variations, due to clear engineering differences: The original Datalog analysis has to bound the access path length for aliases to a finite number, while the optimized data structure implicitly stores aliases for longer access paths. This may allow inferring more aliases even for shorter access paths. Also, for engineering simplicity, the Java implementation incorporates just a naive fixpoint computation. In each fixpoint iteration, it keeps track of a set of instructions whose alias graphs might be affected by the current changes, and those are the instructions that will be inspected in the next iteration. This process is repeated until there are no further instructions for consideration. The fixpoint computation is only crudely tuned for performance and the algorithm is eager to mark an instruction as a candidate for the next iteration.

Since the analyses are virtually equivalent, we next compare them only on speed. (For context, however: the analysis is quite effective and computes far from trivial results. It produces enough must-alias inferences to determine unique points-to targets for 20-40% of local variables in all benchmarks. Its current main uses inside Doop are to enable strong updates in a may-point-to analysis, as well as to produce results for inspection by humans, to aid program comprehension.)

We use a 64-bit machine with an Intel(R) Xeon(R) E5-2687W v4 (24-cores) CPU at 3.00GHz. The machine has 512GB of RAM. All measurements are single-threaded (though, as is common, Java runs its garbage collector in extra threads) and all executions occupy only a small fraction of the available RAM. We experiment with the DaCapo benchmark programs [11] v.2006-10-MR2 and v.9.12-bach under JDK 1.7.0_45. We use the LogicBlox Datalog engine, v.3.10.14.

Speed across benchmarks. Figure 6.4 shows the performance effect of our optimized data structure on analyzing the benchmark programs. We bounded the access path length (for the original Datalog analysis) to 3 and the analysis context depth to 2.

Note that the figure is log-scale. Across all benchmarks, the difference between the optimized implementations and the original is typically at least an order of magnitude and often close to two. The speedup of the two optimized implementations (vs. the original) is also shown more explicitly in Figure 6.5: over half the benchmarks enjoy speedups of over **20x** for both the Java and the Datalog optimized implementation. The Java version of the data structure achieves a median speedup of **25.7x** (min **8.4x**, max **68.9x**), while the Datalog version has a median speedup of **24.6x** (min **5.4x**, max **47.3x**). The analysis time typically drops from over ten minutes to under half a minute.

It is not hard to see why the explicit representation is not competitive. Figure 6.6 correlates the number of aliased access-path pairs (computed by the original analysis) and execution time. (This applies to context-qualified access paths, in the application and libraries alike, as long as the library code is reachable from application code with the given context depth.)

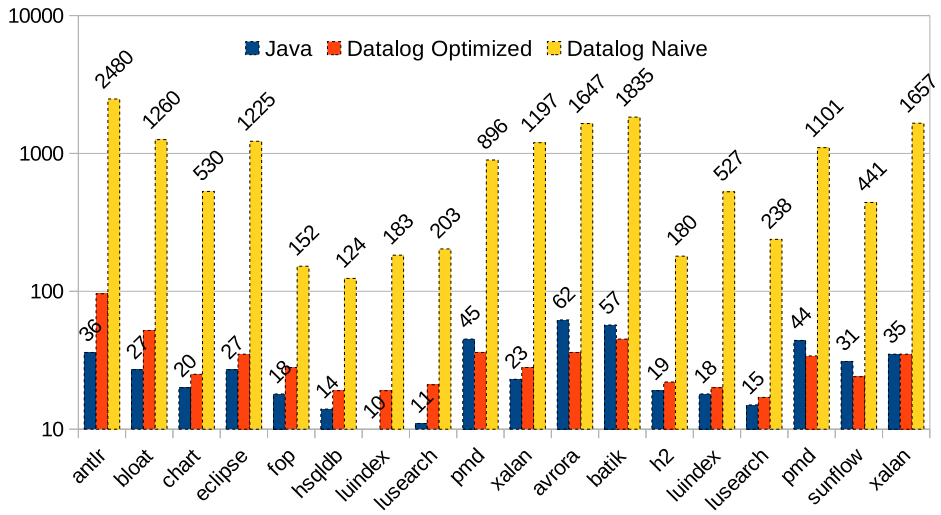


Figure 6.4: Execution time (in seconds) of the analysis. We only show the numbers for the Java and Datalog naive versions, to avoid crowding the chart.

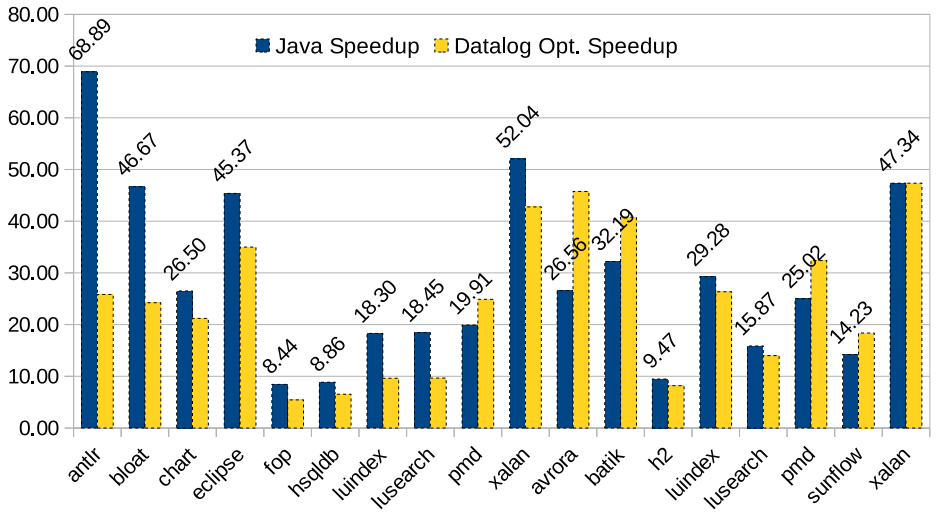


Figure 6.5: Speedups of employing the optimized data structure.

This metric reflects the size (in tuples) of the corresponding relation in the Datalog database. It clearly suggest that maintaining access path relationships explicitly can prove quite costly.

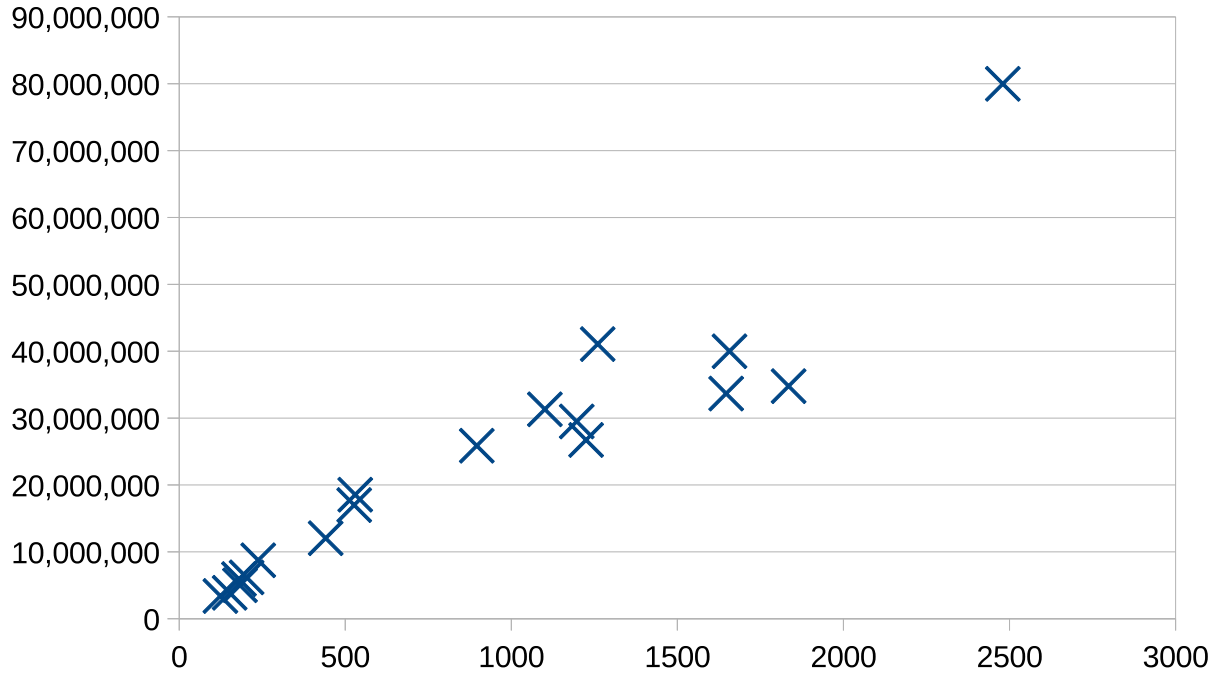


Figure 6.6: Number of pairs of (instruction-and-context qualified) access paths that must alias vs. analysis time.

Furthermore, the setup explicitly downplays the benefits of our optimized data structure: First, the "optimized" running time also includes import time and computing must-point-to results for checking the equivalence of analyses. The latter is quite costly, since it requires re-expanding the compactly-preserved access paths. The true computation times of the optimized analysis are about one-third of the times listed in Figure 6.4. Second, the configuration parameters (context depth of 1 and access paths of at most length 2) are very modest, to present the explicit representation in the best possible (while still realistic) light. Changing these parameters can incur dramatic slowdowns for the explicit representation, as we examine next.

Varying access-path length. To further see the performance advantage of the optimized representation of must-alias information, one can vary the maximum access path length allowed for computations of the original, explicit (Datalog) implementation. Figure 6.7 shows how running time varies for maximum access path lengths of 2, 3 (same as in Figure 6.4), 4 and 5. The numbers are for the xalan benchmark. The speedup readily grows to over **75x** for an allowed access path length of 5. The optimized Datalog implementation is shown as a baseline although it should be (and is) largely unaffected by the change of maximum access path length.

Varying context depth.

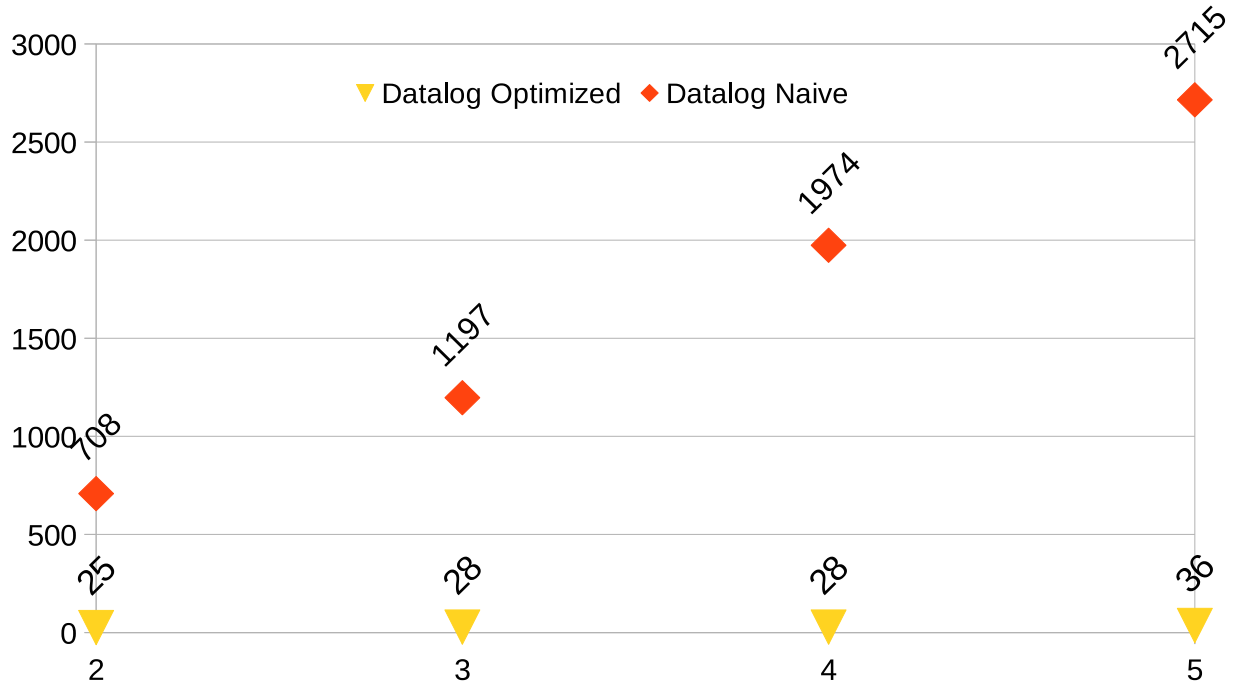


Figure 6.7: Execution time when varying maximum access-path length. Optimized Datalog running time given as a baseline.

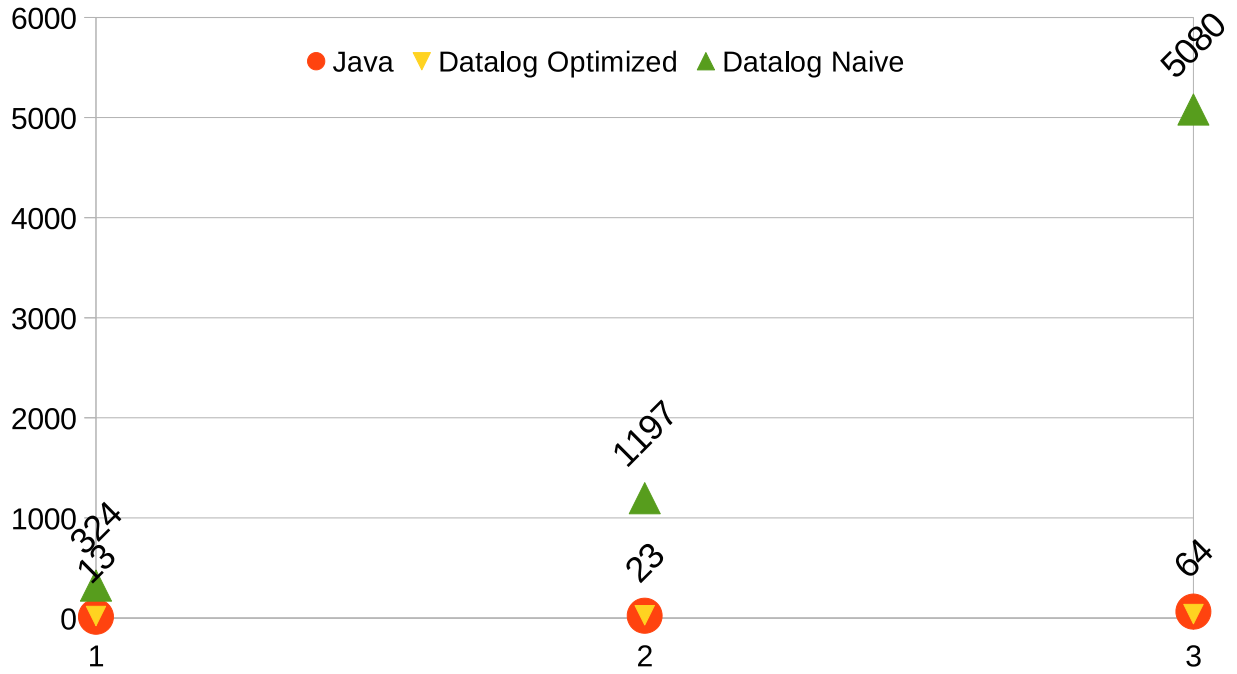


Figure 6.8: Execution time when varying maximum context depth.

Similar observations can be made by varying the context depth of the analysis. As seen in Figure 6.8, although the running time of the optimized implementation grows slowly, the running time of the explicit representation of alias relationships gets dramatically higher. For a context depth of 4, the explicit representation did not terminate after one-and-a-half hour.

Recall the two claimed benefits of the optimized representation: long access paths are represented implicitly, and equivalence classes are represented with linear space and time complexity, instead of quadratic. It is the latter factor that comes into play when context depth increases: alias sets grow in size, by exploiting inter-procedural inference (e.g., aliasing established at the caller and propagated through formal arguments) in addition to local instructions.

6.4 Summary

We presented a data structure for the its optimized implementation of must-alias analysis over access paths. In all, the optimized representation fulfills its promise of a much more economical representation of must-alias (equivalence) relations. The algorithmic improvements afforded by the specialized data structure yield a large performance advantage, often approaching two orders of magnitude.

7. DEFENSIVE POINTS-TO ANALYSIS

> You don't just give up. You don't just let things happen. You make a stand!
>
> *Rose Tyler* - Doctor Who

Soundness is a coveted property of static analyses, to the extent that the term is often colloquially used as a synonym for "correctness". For a may-analysis, soundness means that the analysis abstraction over-approximates all concrete executions. A sound value-flow or points-to analysis is one that computes, per program point or per variable, value sets that represent (at least) all values that could possibly arise at the respective point during any possible execution.

Full soundness is hard to achieve in practice due to code that cannot be analyzed (e.g., dynamically generated/loaded code, binary/native code) or dynamic language features (e.g., reflection, `eval`, `invokedynamic`). We collectively refer to such features as *opaque code*. For instance, the Java code below invokes an unknown method, identified by string `methodName`, over an object, `obj`.

```
Method m = obj.getClass().getMethod(methodName);
m.invoke(obj);
```

String `methodName` could be a true run-time value—e.g., read from a file or external resource. Object `obj` could itself be of a type not available during analysis—e.g., `obj` could be obtained through the network and statically typed using a vague interface or root-of-hierarchy type.

Faced with such complications, all past analyses that claim soundness have done so under *a priori* qualifications. Prominently, abstract-interpretation-based [29] approaches, such as Astrée [35], have long emphasized soundness. The conceptual form of such a soundness result is as follows:[1]

> An *Analysis* of programs in language *Lang* is sound relative to language subset *Lang*′ and executions set *Exec*′ iff:
>
> $$\forall \text{ program } P \in Lang\text{: } P \in Lang' \wedge e \in Exec' \implies e \in \gamma(Analysis(P))$$
>
> (where γ is the concretization function that maps abstractions in the output domain of *Analysis* to concrete executions in a universe *Exec*, superset of *Exec*′).

The problem with this formulation of soundness is that, although it yields provable theorems, the *a priori* qualification excludes virtually all realistic programs. The *Lang*′ or *Exec*′ of published proofs disqualify the vast majority of modern programs "in the wild". Language subset *Lang*′ will typically exclude all dynamic features (e.g., reflection) and/or

[1] This formulation is due to Xavier Rival of the Astrée project (e.g., [129]).

executions subset $Exec'$ will disqualify all behaviors that are deemed too-dynamic (e.g., invoking dynamically-loaded code). Reflection alone disqualifies **~80%** of Java programs in the 461-program corpus of the recent Landman et al. study [79].

The above issues have led several members of the static analysis community to proclaim that "*all published whole-program analyses are unsound*" [94], i.e., their soundness guarantee does not apply to realistic programs, and similarly that "*[there is not] a single realistic whole-program analysis tool [...] that does not purposely make unsound choices*". The problem is, therefore, both theoretical and practical. Soundness theorems do not give guarantees for realistic programs. Implementations of analyses in tools happily perpetuate the illusion: they handle soundly the language features one can prove theorems about, while cutting corners in the sound handling of all *other* features, in order to demonstrate greater scalability or precision. For instance, in our earlier Java code fragment, even if the type of `obj` is known, many implemented static analyses will not consider all its methods (which now form a small finite set) as possible values of `m`, but will instead ignore the code altogether. This phenomenon has led to the introduction of the term *soundy* [94] to characterize such analyses (recall Section 2.9). Despite the derogatory tone, "soundy" analyses are the current *good* case of static analyses! They are realistic analyses that handle all "normal" language features soundly.

In this work, we propose *defensive analysis*: a static analysis architecture that addresses the above soundness shortcomings. The basis of defensive analysis can be seen as a different conceptual formulation of soundness.

> An *Analysis* of program P in language *Lang* computes results, $Analysis(P)$, together with soundness marker sets, $Claim(P)$. The *Analysis* is sound iff:
>
> $$\forall \text{ program } P \in Lang, \text{execution } e\text{: } e[Claim(P)] \in \gamma(Analysis(P))[Claim(P)]$$
>
> (where γ is as before, and $e[Claim(p)]$ is the restriction of an execution e to program points with soundness claims, and the definition is similarly lifted to sets of executions).

In other words, the analysis imposes no (or very liberal) *a priori* restrictions to its soundness claims, but instead *computes* the claimed domain of its soundness: the program parts for which the analysis result is sound. The soundness theorem applies to all (or most) programs, under all execution conditions—instead of eagerly disqualifying the vast majority of real-world programs. The extent of soundness is now defined over program points and becomes an experimentally measurable quantity: the size of $Claim(P)$ (which we term the *coverage* of the analysis) can be measured to quantify for which percentage of a program's points the analysis is guaranteed to produce sound results.

The challenge of defensive analysis is, thus, to distinguish parts of the program that are certain to not be affected by opaque code. Delineating "safe" from "unsafe" parts of the program is an ambitious goal, since opaque code can do virtually anything: it can add dynamically-generated subclasses with never-seen methods that get called (via dynamic dispatch or first-class functions) at unsuspecting program points; it can call any existing method

or alter any field via reflection; it can interpose its own implementations at every place where the program uses a reflective lookup; worst of all, it can wreak havoc on all parts of the heap reachable from any reference that escapes into opaque code.

To achieve the goal of distinguishing "safe" inferences, defensive analysis has a different logical form from past analyses. Instead of adding sophisticated handling of opaque code, defensive analysis redefines the analysis logic for regular, common language features, to defensively protect against the possibility that the analysis information potentially depends on opaque code. In essence, a defensive analysis produces inferences only when these are guaranteed to hold because of existing code and language features, and cannot possibly be violated by other, unknown code. Although this seems like a straightforward mode of operation, applying it to yield a realistic static analysis with useful coverage is a challenge.

We designed and implemented a defensive may-point-to (henceforth just "points-to") analysis for Java. The analysis follows the above form, explicitly designating points-to sets that are sound, i.e., that contain at least all the values that may ever arise in actual executions. Soundness guarantees carry over to the implementation: the soundness proof explicitly models all other language features as `<unknown>` instructions and makes only weak, semantically-justified assumptions (e.g., a type-safe heap) about them. Soundness reasoning is defensive in that it establishes when the analysis can be certain to know the full contents of a points-to set, no matter what opaque code can do (within the stated weak assumptions).

In our effort to implement defensive analysis in a realistic package, we found *laziness* to be an essential feature—the analysis cannot scale without it for real-world programs. Laziness means that the analysis does not compute points-to sets unless it can also claim their soundness. That is, program points outside of the $Claim(P)$ set do not get populated at any point—they remain empty throughout the analysis. Consequently, all points-to sets with a potentially unbounded number of objects (e.g., sets that depend on reflection or dynamic loading) are represented as the *empty set*: the analysis never computes any contents for them. An empty analysis result merely means "I don't know", which could signify that the points-to set is affected by opaque code, or simply that the analysis cannot establish that it is *not* affected by opaque code. Laziness yields high efficiency: the analysis can fall-back to an empty set (i.e., implicitly unbounded) without performing any computation or occupying space.

The defensive nature of the analysis combined with laziness result in a very simple specification. The analysis does not need to integrate complex escape or alias reasoning (i.e., "can this object *ever* escape into opaque code?"), but only best-effort logic (i.e., "here are simple, safe cases, when the object cannot possibly be affected by opaque code"). Failure to establish non-escaping merely means that the points-to set remains empty, to denote "I don't know" or "potentially unbounded".

In terms of applications, soundness is a requirement for ambitious clients, such as automatic optimization or semantics-preserving refactoring. Such clients are simply not feasible with past approaches. In particular, inter-procedural automatic optimization has long been hindered by the absence of sound points-to analysis information. Thus, our work has significant

potential for wide practical application.

Concretely, in this chapter we describe the following:

- We offer a general static may-point-to analysis that yields sound results for realistic programs in the presence of opaque code. (Arguably other approaches [61, 62, 80] achieve soundness, but not for the full problem. Additionally, an analysis can be nominally sound by rejecting wholesale any program that employs features that the analysis does not handle. This is also not a solution. We compare more thoroughly in Section 8.4 of related work.)

- The analysis is efficient, leveraging its lazy representation of points-to sets. As a result, it can be made precise, beyond the limits of standard whole-program points-to analyses—e.g., for a 5-call-site-sensitive and flow-sensitive analysis. The analysis is also modular: it can be applied to any subset of the program, and will merely leave more points-to sets empty if other parts are unknown.

- We show that the analysis, though quite defensive, yields useful coverage. In measurements over large Java benchmarks, our analysis computes guaranteed over-approximate points-to sets ($Claim(P)$) for **34-74%** of the local variables of a conventional unsound analysis. (This number is much higher than that of a conventional sound but intra-procedural analysis.) Similar effectiveness is achieved for other metrics (e.g., number of calls de-virtualized), again with actionable, guaranteed-sound outcomes.

- The work brings some clarity to the domain of static analysis of opaque code. The approach allows, for the first time, to quantitatively weigh the benefits of sound inter-procedural analysis against its costs.

7.1 Analysis Illustration

We next describe the setting of defensive analysis and illustrate its principles and behavior.

7.1.1 Soundness and Design Decisions

Defensive analysis is a may-point-to analysis based on access paths, i.e., expressions of the form "`var(.fld)*`"). That is, the analysis computes *the abstract objects* (i.e., allocation sites in the program text) *that an access path may point to.* The analysis is flow-sensitive, hence we will be computing separate points-to information per program point. Both of these design decisions are integral elements of the analysis, as we will justify in Section 7.1.2.

Soundness in this setting means that the analysis computes an over-approximation of any points-to set—i.e., the analysis computes (abstractions of) all objects that may occur in an actual execution. However, since not all allocation sites are statically known (due to dynamically loaded code), such an over-approximation cannot be explicit: not all possible

values in a points-to set can be listed. Thus, there needs to be a special value, $\top$, to denote "unknown", i.e., that the analysis cannot bound the contents of a points-to set.

Defensive analysis takes the above observation one step further, by employing a *lazy* approach: it never populates a points-to set if it cannot guarantee that it is bounded. Thus, an empty points-to set for an access path signifies that (as far as the analysis knows) the access path can point to anything.[2]

In other words, an empty set can be thought to represent a bottom ($\bot$) value during the analysis computation: it just marks a set as having no known values—yet. A set stops being empty only when all the possible ways (in known or unknown code) to contribute values to it have been examined and are found to have bounded contents. At the end of the analysis, all sets that have remained empty signify that the analysis could not bound their contents, i.e., they do not belong in the set $Claim(P)$ of program points with soundness claims. Therefore an empty set after termination of the analysis is conceptually equivalent to a top ($\top$) value: the set could contain anything. This is consistent with the defensive nature of the analysis: not knowing all the values of a set is considered just as bad as knowing it can point to anything.

With this representation choice, the analysis does not need to expend effort in order to be sound. All points-to sets (for any valid access path, of any length) start off empty, i.e., if the analysis were to stop at that point it would report them as having $\top$ values, meaning "the set can contain anything". This is a sound answer, and is only subsequently refined.

This lazy evaluation means that defensive analysis does not need to employ sophisticated mechanisms to simply be sound. For instance, instead of a precise over-approximate escape analysis, defensive analysis can use a simple analysis (including none at all) to compute straightforward cases when an object is guaranteed to never escape into opaque code.

7.1.2 Background and Illustrating Design Decisions

We can see the rationale behind our design decisions through simple examples.

Baseline intra-procedural resoning. It is easy for an analysis to be sound locally, in an *intra-procedural* setting. For instance, when a variable is freshly assigned with a newly allocated object, we are guaranteed to soundly know its points-to set:

```
x = new A();  // abstract object a1, x points-to set is {a1}
```

We can also propagate such information transitively through local assignments, as long as no opaque code can interfere. In the case of local variables, standard concurrency models (for Java, C++, etc.) do not allow interference from other threads, hence points-to sets remain sound, as long as the code itself does not call out to opaque code:

[2] We use an explicit abstract value for `null`, therefore a points-to set that only contains `null` is not empty. This is standard in flow-sensitive analyses, anyway. (In flow-*insensitive analyses*, `null` is typically a member of every points-to set, so it is profitable to not represent it, and hence have an empty set mean a `null`-only reference. No such benefit would arise in our flow-sensitive setting.)

```
x = new A();  // abstract object a1, x points-to set is {a1}
y = x;        // y points-to set is {a1}
z = y;        // z points-to set is {a1}
```

This approach is one often taken by traditional compilers (ahead-of-time or just-in-time alike) in order to perform intra-procedural optimizations, such as those based on traditional data-flow analysis. (Later, in our experimental evaluation, we compare against such a baseline "intra-procedural sound" analysis.)

However, the challenge is to also reason soundly about *inter-procedural* behavior. This includes reasoning about the heap (i.e., reading fields of objects) and about method calls and returns, whose resolution may be dynamic. This will be the focus of the defensive analysis specification.

Inter-procedural elements. The large potential for opaque code to affect inter-procedural analysis results has prevented past analyses from being sound. For instance, consider a simple heap load instruction:

```
x = y.fld;
```

Imagine that the analysis has (somehow) soundly computed all the objects that `y` may point to. It may also know all the places in the code where field `fld` is assigned and what is assigned to it. However, the analysis still cannot compute soundly the points-to set of `x` unless it also knows that all objects referenced by `y` can never escape to opaque code. This is hard to establish: not only do all sites of opaque code (reflection, unknown instructions, potential dynamic code generation sites, and more) need to be marked, but the analysis needs to know an over-approximation of which objects these sites can reach. This requires to have pre-computed an over-approximate (i.e., sound) points-to analysis, which is the problem we are trying to solve in the first place. Past work has dealt with this problem with unrealistic assumptions. For instance, Sreedhar et al. [149] present a call-specialization analysis that can handle dynamic class loading, but only if given the results of a sound may-point-to analysis as input.

Instead, defensive analysis pessimistically computes that a points-to set is $\top$ (i.e., can contain anything) unless it is certain that its contents are bounded. When can the analysis know this, however? Such a guarantee of bounded contents typically comes from having precisely tracked the contents of a variable or field all the way from its last assignment, and having established that no other code could have interfered. For instance, let us expand our earlier example:

```
y.fld = new A(); // abstract object a1, y.fld points-to set is {a1}
...              // analyzable, non-interfering code
x = y.fld;
```

The analysis can now know that the points-to set of `x` is `{a1}`, i.e., the singleton containing the allocation site for `A` objects on line 1. For this to be true, the analysis has to establish that all code between the store instruction to `y.fld` and the subsequent load does not interfere with the value of `y.fld`. For example, we can be certain of such non-interference if the

code does not contain a store to field `fld` of *any* object, does not call any methods, and no other thread can change the heap at that segment of the program. These are simple, local conditions that the analysis may well be able to establish.

In practice, our defensive analysis will do a lot more: it will track method calls, up to a maximum context depth, to ascertain when they can interfere with points-to sets. (If any interference is detected, the points-to set propagated forward is empty.) For instance, in the example code below, the analysis can know with certainty the points-to set of `x` on line 6, whenever method `foo` is called from line 3 of the program fragment.

```
1  y.fld = new A(); // abstract object a1, y.fld points-to set is {a1}
2  z.otherFld = new B();
3  foo(y);
4
5  void foo(W w) {
6      x = w.fld;   // x-for-call-site-3 points-to set is {a1}
7  }
```

Note the elements that contribute to such reasoning: The result holds soundly only when `foo` is called from the specific call site. This result is established only by tracking the value of `y.fld` (renamed to `w.fld` inside method `foo`) instruction-by-instruction all the way to line 6. The heap store instruction on line 2 is guaranteed to not affect `y.fld` (regardless of whether `z` and `y` alias or not), since Java guarantees object isolation and the reference is to a different field. (More on language model assumptions in Section 7.1.3.)

The above example helps illustrate the design choices of defensive analysis: it is a flow-sensitive, context-sensitive analysis because it needs to track all points-to information that is guaranteed to hold, per-instruction, following closely all possible control-flow of the program, even across calls. It is also an analysis computing points-to information on access paths because this gives significantly more ability to reason about the heap locally. For instance, in the above program fragment, we may not know which objects `y` may point to.[3] However, we do know that `y.fld` certainly points to abstract object `a1` after line 1!

Laziness. Finally, consider the design choice of representing unbounded points-to sets as empty, i.e., to lazily compute the contents of points-to sets only if they can be proven to be finite. Defensive analysis requires laziness for scalability. (Experimentally, a non-lazy analysis does not scale for any non-zero context depth, i.e., cannot be effective inter-procedurally.)

Laziness means skipping an explicit representation of $\top$, in favor of keeping points-to sets empy ($\bot$ in the usual lattice of sets) as long as possible. (As mentioned earlier, at the end of the analysis, all sets that stayed $\bot$ become implicitly $\top$.) This has the minor benefit of avoiding storage of $\top$ values, since empty sets are represented without consuming memory. More majorly, however, it enables the analysis to give a convenient meaning to any finite points-to sets that arise. Instead of "*this set currently has bounded contents, but may become*

[3] In fact, even if we did know, these would be abstract objects. Static analysis would almost never be able to establish soundly what their `fld` field points to, because this information needs to capture the `fld` values of all *concrete* objects (not just the latest one) represented by the same abstract object.

$\top$ *during the course of the analysis*", a non-empty set of values implies "*this set has bounded contents and is guaranteed to always have bounded contents*". By making this distinction, the analysis never wastes effort computing points-to sets with explicit (non-$\top$) contents only to later discover that the points-to set is $\top$. For an example of how much wasted effort can be saved by being lazy, consider an example program involving a heap load and a virtual call:

```
1  y.fld = new A();  // abstract object a1
2  while (...) {
3      x = y.fld;
4      x.foo(y);
5  }
```

An analysis may have computed all the abstract objects that `y.fld` may point to at line 3. One of these computed objects may induce a different resolution of the call instruction (line 4), which can suddenly lead to the discovery that an object aliased with `y.fld` can enter opaque code (while this was not true based on what the analysis had computed earlier). Since the object referenced by `y.fld` can change in code that is not analyzed, the points-to set of `x` at the load instruction will need to be augmented with the implicit over-approximation special value, $\top$. This means that all previously computed values for the points-to sets of `x` and `y.fld` are subsumed by the single $\top$ value. Computing these values and all others that depend on them constitutes wasted effort. To make matters worse, this is more likely to happen for *large* points-to sets, i.e., the more work the analysis has performed on computing an explicit points-to set, the larger (and less precise) the set will be, and the more likely it is that the work will be wasted because the set will revert to $\top$.

The design principle of "laziness in order to avoid wasted effort" is responsible for the scalability of defensive analysis. As we show in our experiments, defensive analysis scales to be flow-sensitive, 5-call-site sensitive over large Java benchmarks and the full JDK. In standard past literature for all-program-points analyses, even a flow-insensitive, 2-call-site-sensitive analysis has been infeasible over these benchmarks [144].[4]

The alternative to using empty sets to represent "anything" would be to use a special $\top$ value. However, this would necessitate (non-monotonic) negative judgments of the form "if the set does not include the value $\top$ then ...". Instead, with the empty set representation, the logic becomes monotonic—"if the set includes some value then ..."—allowing for its efficient implementation with generic fixpoint machinery, such as a Datalog engine.

[4]It is worth emphasizing that, although defensive analysis is lazy, this is a very different form of laziness than that of *on-demand* points-to analysis (e.g., [10, 148]). An on-demand analysis only computes points-to information for program points that may affect a particular site of interest, instead of the entire program. The defensive analysis we describe is an all-program-points analysis: it computes points-to information for the entire program, i.e., for all possible points-to queries, including ones potentially devised in the future. Yet the analysis is lazy in that it only computes values for points-to sets that it can prove to have bounded contents.

7.1.3 Soundness Assumptions

The soundness claims of defensive analysis are predicated on assumptions about the environment. These assumptions reflect well the setting of safe languages, such as Java:

- **Object isolation.** Objects can only be accessed via high-level references. This means that objects and fields are isolated: an object can be referenced outside the dynamic scope of a method or by a different thread only if a reference to the object has escaped the method or current thread. (This restriction also implies that objects are not contained in one another, though they can contain references to each other.) A field can only be accessed via a base object pointer and a unique field signature.
- **Stack frame isolation.** Local variables are isolated from each other, thread-private and private to their allocating method. No external code can access the local variables of a method, even if the code is executed (i.e., is a callee) under the dynamic scope of the method.
- **Concurrency model.** In the simplified model of this chapter, soundness is predicated on the assumption that standard mutexes (or operations on volatile variables) are used to protect all shared memory data. We later discuss how our implementation removes this assumption.The reason for the simplified concurrency model is that it allows presenting the analysis in its purest form, dealing with core language features such as heap loads/stores and calls, but unencumbered by auxiliary considerations (e.g., computing objects that do not escape into other threads).

Thus, our setting is clearly that of a safe language with near-unlimited potential for dynamic behavior. Notably, we can have unknown instructions; calls to native code with arbitrary behavior (over a well-typed, isolated heap); generation and loading of unknown code (which may also be called, via dynamic dispatch, by unsuspecting *known* code); arbitrary access to existing or unknown objects (both field read/writes and method calls) via reflection, i.e., without such access being identifiable in the program text; and more.

7.2 Defensive Analysis, Informally

The discussion of analysis principles in the previous sections gives the main tenets of defensive analysis. However, these need to be concretely applied over all complex language features affecting points-to information: control-flow merging, heap manipulation, and method calls. We give informal examples next. Following these examples should significantly facilitate understanding the formal specification of the analysis, in later sections.

Control-flow merging. Consider a branching example:

```
1  if (complexCondition())
2      x = new A();            // abstract object a1, x points-to set is {a1}
3  else
4      x = notFullyAnalyzed(); // x points-to set is {}
5  // x points-to set is {}
```

The first branch of the above `if` expression establishes that the points-to set of variable `x` is `{a1}`. For a conventional analysis, this would result in adding `a1` to the points-to set of `x` at the merge point (at line 5). The defensive analysis, however, has to be conservative and not compute values that may later become $\top$. Therefore, it will add `a1` to the final points-to set of `x` only if it can also prove that the points-to set of `x` in the second branch is bounded, i.e., non-empty. If the analysis is not certain of this, the points-to sets of `x`, both in the second branch and at the merge point, stay empty. Inability to bound the points to set of `x` in the second branch can be due a variety of reasons: e.g., there can be opaque code inside `notFullyAnalyzed`, or the analysis may reach its maximum context depth, so that the return value of the method is not tracked precisely.

Heap manipulation. Similar treatment applies to all cases of points-to sets (e.g., for complex access paths) when information is merged: the analysis yields a non-empty result only if it is certain that the result could not have been invalidated by any other code, available or not. For instance, consider the following example of heap store instructions:

```
x.fld = new A();             // abstract object a1, x.fld points-to set is {a1}
y.fld = notFullyAnalyzed(); // x.fld points-to set is {}
```

After the first instruction, the points-to set of access path `x.fld` is computed to be `{a1}`. However, in most cases, the analysis will not be able to ascertain that `x` and `y` are not aliased. Therefore, after the second instruction, the points-to set of `x.fld` will be empty, i.e., unknown. This reflects well the defensive nature of the analysis: whenever uncertain, points-to sets will default to empty, i.e., undetermined.

Generally, since the analysis is flow-sensitive and access-path based, store instructions certain to operate on the same object perform *strong updates*, while store instructions that *possibly* operate on the same object perform *weak updates*:

```
x.fld = new A();  // abstract object a1, x.fld points-to set is {a1}
x.fld = new B();  // abstract object b1, x.fld points-to set is {b1}
y.fld = new B();  // abstract object b2, x.fld points-to set is {b1, b2}
```

In this case, the points-to information of access path `x.fld` is set to `{b1}` after the second store instruction, ignoring the previous contents. (The example assumes that types `A` and `B` are both compatible with the static type of `x.fld`.) After the third store instruction, however, a new element is added to the points-to set—again, under the assumption that the analysis cannot determine whether `x` and `y` are aliased.

The different element in defensive analysis is that if any of the involved points-to sets is empty, both strong and weak updates yield an empty points-to set. For instance, replacing either of the last two allocations (`new B()`) above with a call to opaque (or not fully analyzed) code would make all subsequent points-to sets of `x.fld` be empty.

Method calls. Defensive analysis computes sound may-point-to information simultaneously with sound call-graph information. The analysis employs the same principles for the call-graph representation as for points-to: a finite set of method call targets means that the set is guaranteed bounded, while an empty set of method call targets means that the analysis

cannot (yet) establish that all target methods are known.

To compute a sound over-approximation of method call targets, one needs a bounded may-point-to set for the receiver. Otherwise, the receiver object could be unknown—e.g., an instance of a dynamically loaded class—resulting in an unsound call-graph.

When the set of method call targets is not bounded, dynamic calls cannot be resolved and the analysis has to be conservative. For instance, in the example below, a conventional unsound analysis would resolve the virtual call `x.foo()` to, at least, the method `A::foo`, i.e., `foo` in class `A`.

```
if (complexCondition())
    x = new A();  // abstract object a1
else
    x = notFullyAnalyzed();
x.foo();
```

In contrast, recall that for a defensive analysis the points-to set of `x` at the point of the call to `foo` is empty. Accordingly, the defensive analysis does *not* resolve the virtual call at all: per the lazy evaluation principle, there is no point of computing what *one* target of the call will do, when other targets are unknown and full soundness (i.e., guaranteed over-approximation) is required. This means that all heap information (i.e., all access-path points-to information, except for *trivial* access paths consisting of a single local variable and no fields) that held before the method call ceases to hold after it! (There are notable exceptions—e.g., for access paths with final fields, or for cases when an escape analysis can establish that some part of the heap does not escape into the called method. Section 7.4 discusses such intricacies.)

When method calls *can* be resolved, the target methods have to be analyzed under a context uniquely identifying the callee. A defensive analysis may know all methods that can get called at a certain point, but *it cannot know all callers of a method*. Consider the following example:

```
1  void caller() {
2      A x = new A(); // abstract object a1
3      callee(x);
4  }
5
6  void callee(A y) {
7      ...
8  }
```

Assume that there is no other discernible call to `callee` anywhere in the program. An unsound analysis would establish that variable `y` in `callee` (i.e., immediately after line 6) points to abstract object `a1`. A defensive analysis, however, cannot do the same unconditionally. The points-to set of `y` without context information has to be the empty set. The reason is that there may be completely unknown callers of `callee`—e.g., in existing code, via reflection, or in dynamically loaded code. Such callers could pass different objects as arguments to `callee` and the analysis cannot upper-bound the set of such arguments. Thus, the only safe answer for a defensive analysis is "undetermined"—i.e., an empty set.

Thus, in order to propagate analysis results inter-procedurally, a defensive analysis has to leverage context information. In the above example, what the analysis will establish is that `y` points to `a1` *conditionally*, under context 3, signifying the call-site instruction (line 3 in the code snippet).

The above implies that the use of context in a defensive analysis is rather different than in a traditional unsound points-to analysis. Contexts in standard points-to analysis can be *summarizing*: a single context can merge arbitrary concrete (dynamic) executions, as long as any single concrete execution maps uniquely to a context. For instance, a 1-object-sensitive analysis [105] merges all calls to a method as long as they have the same abstract receiver object, independently of call sites.

Context in a conventional analysis only adds *precision*, relative to a context-insensitive analysis. In contrast, context in a defensive analysis is necessary for *correctness*: since information is collected per-program-point, propagating points-to sets from a call site to a callee can only be done under a context that identifies the call-site program point. Contexts cannot freely summarize multiple invocation instructions, because there may be others, yet unknown, invocations that would result in the same context.

Therefore, a context-sensitive defensive analysis has to be, at a minimum, *call-site sensitive* [137, 140]: the call site of an analyzed method has to be part of the context (as will, for deeper context, the call site of the caller, the call site of the caller's caller, etc.). Other kinds of context (e.g., object-sensitive context [105, 142]) can be added for extra precision.

7.3 A Model of Defensive Analysis

We next present a rigorous model of our defensive analysis. The model is based upon the input domain and the minimal intermediate language presented in previous chapters—also presented here for clarity. The language can be straightforwardly enhanced with features such as arrays, static members and calls, exceptions, etc. The input program is assumed to be in a single-return-per-method form. We will use the following formalism, auxiliary functions and predicates throughout:

V is a set of variables	v, u		
T is a set of types	T, S	i: `v = new T()`	object allocation
F is a set of fields	f	i: `v = u`	move (or assignment)
M is a set of methods	meth	i: `v = u.f`	field load
I is a set of instruction labels	i, j, k	i: `v.f = u`	field store
C is a set of contexts	c, d	i: `v.meth(*)`	virtual call
O is a set of abstract objects	$\hat{o}$	i: `return`	method return
P is a set of access paths	ap	i: `<unknown>`	anything else
$\mathbb{N}$ is the set of natural numbers	n		

Figure 7.1: Input domains and common meta-variables used, as well as the instruction set of the intermediate language.

- Instructions are linked into a control-flow graph, via relation $i \xrightarrow{NEXT} j$.
- Objects can potentially identify their allocation instruction, e.g., $\widehat{o_i}$.
- $\texttt{meth}_{\texttt{T}}$ is the result of looking up method signature `meth` in type `T`.
- $\texttt{meth}[n]$ is the n-th instruction of method `meth`.
- We overload the "$\in$" operator to more than set membership, in unambiguous contexts, namely: $i \in \texttt{meth}$ (instruction is in method), $\texttt{f} \in \texttt{ap}$ (field is in access path), $\widehat{o} \in \texttt{T}$ (abstract object is of type), $\texttt{v} \in \texttt{T}$ (variable is of type).
- $arg_n^{\texttt{meth}}$ and arg_n^i denote the n-th formal or actual arg of a method and invocation instruction, respectively. (By convention, the **`this`**/base variable of a method invocation is assumed to be the 0-th argument.)
- $\texttt{ap}[\texttt{v}/\texttt{u}]$ is the access path `ap` after substituting the base `v` to `u` (if applicable).

7.3.1 Analysis Structure

Figure 7.1 shows the analysis specification, in terms of constraints. Any solution satisfying these constraints has the desired soundness property and in Section 7.3.2 we discuss extra considerations so that the constraints can also be used to *compute* a solution. We recommend following the figure together with our text explaining the rules: although the rules are precise (transcribed from a mechanized logical specification) some are hard to follow without explanation of their intent beforehand. The analysis constraints define the following relations:

- The AccessPathPointsTo relation, in two varieties, before and after an instruction: $i : \texttt{ap} \xrightarrow{IN}_c \widehat{o}$ and $i : \texttt{p} \xrightarrow{OUT}_c \widehat{o}$ (`ap` may point to $\widehat{o}$ before/after instruction i executed under context c). This is our sound may-point-to relation: if, at the end of the analysis, the set of $\widehat{o}$ for given i, p, c is not empty, it will be a superset of the abstract objects $\widehat{o}$ pointed by `ap` at the given program point and context during any dynamic execution.[5]
- The MayCall relation, i.e., our sound call-graph representation: $i \xrightarrow[c']{CALLS}{}^{c} \texttt{meth}$ (instruction i executed under context c may call method `meth` and the resulting context will be c').
- The Reachable relation, $\overline{\texttt{meth}}^c$, denoting that method `meth` is reachable under context c, and should, thus, be analyzed. This relation is partially populated when the analysis starts: it holds an initial set of methods, under the empty context Init, that should be analyzed.

[5]To be precise, concrete objects arise during execution but we are considering their standard mapping to abstract objects, per allocation site.

$$(\textsc{Alloc})\ \frac{i:\ \mathtt{v}\ =\ \mathbf{new}\ \mathtt{T}()\qquad i \in \mathtt{meth}\qquad \overline{\mathtt{meth}}^{c}}{i:\ \mathtt{v} \xrightarrow{OUT}_{c} \widehat{o}_i}$$

$$(\textsc{Move})\ \frac{i:\ \mathtt{v}\ =\ \mathtt{u}\qquad i:\ \mathtt{ap} \xrightarrow{IN}_{c} \widehat{o}}{i:\ \mathtt{ap}[\mathtt{u}/\mathtt{v}] \xrightarrow{OUT}_{c} \widehat{o}}$$

$$(\textsc{Load})\ \frac{i:\ \mathtt{u}\ =\ \mathtt{v}.\mathtt{f}\qquad i:\ \mathtt{v}.\mathtt{f} \xrightarrow{IN}_{c} \widehat{o}}{i:\ \mathtt{u} \xrightarrow{OUT}_{c} \widehat{o}}$$

$$(\textsc{Store-1})\ \frac{i:\ \mathtt{u}.\mathtt{f}\ =\ \mathtt{v}\qquad i:\ \mathtt{v} \xrightarrow{IN}_{c} \widehat{o}}{i:\ \mathtt{u}.\mathtt{f} \xrightarrow{OUT}_{c} \widehat{o}}$$

$$(\textsc{Store-2})\ \frac{i:\ \mathtt{u}.\mathtt{f}\ =\ \mathtt{v}\qquad i:\ \mathtt{v} \xrightarrow{IN}_{c} \widehat{o}\qquad i:\ \mathtt{w}.\mathtt{f} \xrightarrow{IN}_{c} \widehat{o}'\qquad \mathtt{w} \neq \mathtt{u}}{i:\ \mathtt{w}.\mathtt{f} \xrightarrow{OUT}_{c} \widehat{o}\qquad i:\ \mathtt{w}.\mathtt{f} \xrightarrow{OUT}_{c} \widehat{o}'}$$

$$(\textsc{CFG-Join})\ \frac{j \xrightarrow{NEXT} i\qquad j:\ \mathtt{ap} \xrightarrow{OUT}_{c} \widehat{o}\qquad \forall k : (k \xrightarrow{NEXT} i) \implies (k:\ \mathtt{ap} \xrightarrow{OUT}_{c} *)}{i:\ \mathtt{ap} \xrightarrow{IN}_{c} \widehat{o}}$$

$$(\textsc{Frame-1})\ \frac{i:\ \mathtt{v} \xrightarrow{IN}_{c} \widehat{o}\qquad \neg(i:\ \mathtt{v}\ =\ *)\qquad \neg(i:\ \texttt{<unknown>})}{i:\ \mathtt{v} \xrightarrow{OUT}_{c} \widehat{o}}$$

$$(\textsc{Frame-2})\ \frac{\begin{array}{c} i:\ \mathtt{ap} \xrightarrow{IN}_{c} \widehat{o}\qquad \mathtt{ap} = \mathtt{v}.* \\ \neg(i:\ *.\mathtt{meth}(*))\quad \neg(i:\ *.\mathtt{f}\ =\ *)\quad \neg(i:\ \mathtt{v}\ =\ *)\quad \neg(i:\ \texttt{<unknown>}) \end{array}}{i:\ \mathtt{ap} \xrightarrow{OUT}_{c} \widehat{o}}$$

$$(\textsc{Frame-3})\ \frac{i:\ *.\mathtt{f}\ =\ *\qquad i:\ \mathtt{ap} \xrightarrow{IN}_{c} \widehat{o}\qquad \mathtt{f} \notin \mathtt{ap}}{i:\ \mathtt{ap} \xrightarrow{OUT}_{c} \widehat{o}}$$

$$(\textsc{Call})\ \frac{i:\ \mathtt{v}.\mathtt{meth}(*)\qquad i:\ \mathtt{v} \xrightarrow{IN}_{c} \widehat{o}\qquad \widehat{o} \in \mathtt{T}\qquad c' = \mathcal{NC}(i,c,\widehat{o})}{\overline{\mathtt{meth}_{\mathtt{T}}}^{c'}\qquad i \xrightarrow{CALLS}{}^{c}_{c'}\ \mathtt{meth}_{\mathtt{T}}}$$

$$(\textsc{Args})\ \frac{i \xrightarrow{CALLS}{}^{c}_{c'}\ \mathtt{meth}\qquad i:\ \mathtt{ap} \xrightarrow{IN}_{c} \widehat{o}\qquad j = \mathtt{meth}[0]}{j:\ \mathtt{ap}[arg^{i}_{n}/arg^{\mathtt{meth}}_{n}] \xrightarrow{IN}_{c'} \widehat{o}}$$

$$j : \texttt{return} \quad j \in \texttt{meth} \quad i \xrightarrow{CALLS}{}^{c}_{d} \texttt{meth} \quad j : \texttt{ap} \xrightarrow{IN}_{d} \widehat{o} \quad \texttt{ap} = \texttt{v}.*$$

$$\left\{\forall\, \texttt{meth}', c' : (i \xrightarrow{CALLS}{}^{c}_{c'} \texttt{meth}') \implies \left(\exists\, j', \texttt{ap}' : \right.\right.$$

$$\text{(Ret)} \quad \frac{\left.\left.(j' : \texttt{return}) \wedge (j' \in \texttt{meth}') \wedge (\texttt{ap} = \texttt{ap}'[arg_n^{\texttt{meth}'}/arg_n^i]) \wedge (j' : \texttt{ap}' \xrightarrow{IN}_{c'} *)\right)\right\}}{i : \texttt{ap}[arg_n^{\texttt{meth}}/arg_n^i] \xrightarrow{OUT}_{c} \widehat{o}}$$

Figure 7.1: Inference Rules for Defensive Points-to Analysis.

Alloc, Move, Load, Store-1. The first four rules of the analysis are rather straightforward. The Alloc rule is the only one with some minimal subtlety: if an object is freshly allocated, we know that the variable it is directly assigned to points to it. This inference is valid in any reachable context, even the initial, making-no-assumptions, Init context. Therefore this rule is responsible for kickstarting the analysis, producing the first points-to inferences (valid locally) that will then propagate.

Store-2. The Store-2 rule is the first one exhibiting the defensive and lazy features of the analysis. The rule performs a "weak update" on points-to sets of possibly affected access paths, as long as they are guaranteed to be bounded, i.e., they are non-empty. At a store instruction, `u.f = v`, if an access path `w.f` has a base explicitly different from `u` (with `f` being the same), then its points-to set is augmented with any element ($\widehat{o}$) of the points-to set of `v`, while maintaining its original elements ($\widehat{o'}$). This rule defensively adds more information to guarantee an over-approximation in the case of access paths that may be aliases for the same object. The subtlety of the rule lies in its handling of empty points-to sets. If *either* of the points-to sets (of `v` or of `w.f`) is empty before the instruction, the rule does not match, hence the points-to set of `w.f` after the instruction does not acquire any contents. This is consistent with our sound handling: if the earlier contents or the update cannot be upper-bounded, then the resulting points-to set cannot be, either.

Note the contrast between rules Store-1 and Store-2. We do not need to determine precisely the aliasing relationship between base variables `u` and `w`. If there is a chance that the variables are aliased, it is safe to conservatively add more possible values to the points-to set of `w.f`. In the case of Store-1, however, we could do better than the conservative treatment and perform a strong update.

CFG-Join. The next rule deals with merging information from an instruction's predecessors (or merely propagating it, in the case of a single predecessor).

Informally, the rule states that if *some* predecessor instruction, j, has established that ap can point to $\widehat{o}$, *and* if all other predecessors, k, establish that ap points to *something* (so that its points-to set is non-empty, i.e., bounded) then the information is propagated to the points-to relation of the successor instruction. (We use * to mean "any value", throughout the rules.) Note the defensive handling: if even a single predecessor has an unbounded (i.e., empty) points-to set for ap, then the rule is not triggered and the resulting points-to set

remains empty. (This conservative handling can be relaxed, to ignore predecessors that are guaranteed to not affect a certain access path, as will be discussed in Section 7.4.)

Frame-1, Frame-2, Frame-3. The next three rules are *frame rules*, responsible for the propagation of unchanged information.

Informally, the first rule merely says that points-to information for local variables (i.e., an access path consisting of just "`v`") is maintained after an instruction, if it existed before it, as long as the instruction does not directly assign the local variable (as is the case for a load, move, or allocation directly into this local variable). The soundness of this rule is predicated on our earlier assumption of *stack frame isolation*: local variables are isolated from each other, thread-private, and private to their allocating method. Therefore their points-to set cannot change, except with instruction such as the above.

This is the first time we see a treatment of `<unknown>` instructions, which can encode any richer instruction set than our basic intermediate language. The analysis conservatively avoids propagating any points-to information over an unknown instruction. This is also used to handle concurrency, under our simplified model: both `monitorenter/monitorexit` instructions and all accesses to `volatile` variables in the input program are represented simply as `<unknown>` instructions in our intermediate language. (The treatment of `<unknown>` collectively by the analysis rules ensures that all heap information is dropped at that program point, i.e., points-to sets are empty after the instruction.)

The next two rules apply in the case of complex access paths, i.e., of length 2 or more. (Actually rule FRAME-3 also applies to variable-only access paths, but not meaningfully: that case is subsumed by FRAME-1.) First, similarly to the earlier rule, points-to information for the access path is maintained after an instruction (assuming it held before it) unless the instruction assigns the same base variable (again via a load, move, or allocation), or is a call, store, or unknown. Second, points-to information for complex access paths is propagated over all store instructions that affect fields not participating in the access path.

The soundness of these rules is predicated on the *object isolation* and *concurrency model* assumptions of Section 7.1.3. Under these assumptions, the only way to change the points-to set of an access path is via store instructions (on the same field), changing the base of the access path, invoking (potentially opaque) methods, and executing unknown instructions (including `monitorenter/monitorexit`). The rules have strong preconditions to preclude these cases. At the level of the model, we only care about soundness under the given assumptions, no matter how strict. In Section 7.4 we will discuss practical enhancements—e.g., when method calls are fine because the analysis has computed the full potential of their effects on the heap.

Call. The next rule uses points-to information to establish a sound call-graph. The $i \xrightarrow{CALLS}{}^{c}_{c'}$ `meth` relation over-approximates information using the same approach as points-to sets: for a given invocation site, i, and context, c, the relation holds either an empty set (i.e., no matching values exist for (i, ctx)—denoting an unbounded set of destinations—or an over-approximation (i.e., a superset) of all possible targets of the invocation at i under c.

The rule is mostly a straightforward lookup of the target method, based on the receiver

object's type. There are a couple of subtleties, however. The receiver object needs to have an upper-bounded (i.e., non-empty) points-to set, a new context is constructed using function $\mathcal{NC}$, and the target method is considered reachable under the new context. The exact definition of $\mathcal{NC}$ will determine the context sensitivity of the analysis. (We will return to this point promptly in Section 7.4.)

Args. The ARGS rule handles points-to information propagation over calls, from caller to callee. Points-to information for rebased access paths is established for the first instruction ($j = \texttt{meth}[0]$) of a called method, under the callee's established context. The rule examines all access paths whose base variable is an actual argument of the call, as long as they have some points-to information (before the invocation).

Recall our discussion of Section 7.2 regarding method calls and the use of context. The points-to information established at a callee cannot be conflating different callers—there may be unknown callers for the same method, either in existing code (e.g., via reflection) or in dynamically loaded code. Therefore, if we might mix callers, the only sound inference for local points-to sets is $\top$: we cannot bound the values that all callers may pass. Instead, we need to have contexts that uniquely identify the caller, so that we can safely propagate bounded points-to sets.

A straightforward way to ensure that the pair (`meth`, c') uniquely identifies invocation instruction i and context c is to use *call-site sensitivity*: c' is formed by combining i and c—that is, $\mathcal{NC}(i, c, \widehat{o}) = cons(i, c)$. (Contexts can typically grow only up to a pre-determined depth, at which point the $\mathcal{NC}$ function will not return anything, the CALL rule will fail to make an inference, hence the current rule will not fire, leaving points-to sets at the callee empty, i.e., undetermined.)

Ret. The final rule performs a similar propagation of values, this time from callee to caller. The rule is significantly complicated by its last condition (the forall-exists implication), which is key for soundness. The rule states that if some callee has points-to information for complex access path `ap` at a return point, then this information is propagated to the caller, provided that *all other callees* for the same instruction, i, and caller context, c, also have *some* (i.e., non-empty) points-to information for the same access path `p` at their return point. A further complication is that access path `ap` will appear rebased differently for each one of the callees—e.g., access path `actual.field` may appear as `formalA.fld` and `formalB.fld` in two callees `A` and `B`. The rule has to also account for such rebasing.

Note also the earlier condition that access path `ap` be complex, i.e., to have length greater than 1. This reflects call-by-value semantics for references: for a call `meth(actual)` to a method with signature `meth(F formal)`, the points-to information of access path `formal` is not reflected back to the caller, yet the points-to information of longer access paths, e.g., `formal.fld`, is.

The handling of a method return is the only point where a context can become stronger. Facts that were inferred to hold under the more specific context, c', are now established, modulo rebasing, under c. Since c' has to uniquely identify c, typically c will be shorter by one context element.

7.3.2 Reasoning

We prove the soundness of the analysis under an informal language model. We do not attempt to formalize the full effects of opaque code (e.g., what reflection or native code can or cannot do). Such a formalization would be tedious and partial, as new capabilities are added to reflection or dynamic loading APIs with every JDK version. Instead, we establish that the analysis rules always compute over-approximate finite points-to sets (or empty sets), and that this property cannot be affected by opaque code under the common informal understanding of the assumptions of Section 7.1.3. For instance, it is clear from the "stack frame isolation" assumption that local variables cannot change values except by action of the current instruction, i.e., that rule FRAME-1 is alone responsible for soundly transferring such points-to information from the program point before an instruction to after.

A detailed model that formally captures "stack frame isolation" is perhaps desirable assurance (in the vein of verified compilers) but adds nothing to the effort to *invent* a realistic, sound points-to analysis. By analogy, sound compiler optimizations (i.e., ones that do not break the program) exist in virtually all mainstream compilers, but a minuscule fraction of those have been formally verified.

There are two main properties of the defensive analysis:

- Soundness: the analysis computes an over-approximation of points-to sets that may arise during any program execution. Any non-empty set contains a superset of its dynamic contents under any possible execution. Any empty set is considered trivially "over-approximate", to avoid special-casing all our statements. In effect, the analysis produces a set of soundness markers, $Claim(P)$, which coincide with the non-empty points-to sets. No claims are made about empty points-to sets.
- Laziness: the analysis does not waste work; elements that enter a points-to set are never removed (by reverting the set to the $\top$ value—i.e., an empty set).

Theorem 1. *There exists an evaluation order of the rules, such that the defensive analysis model is sound: all points-to sets computed are over-approximate, i.e., are either empty or contain all possible values arising during program execution, under the assumptions of Section 7.1.3.*

Proof. The proof is inductive. Initially, all points-to/call-target sets encoded in relations $\xrightarrow{IN}$, $\xrightarrow{OUT}$, $\xrightarrow{CALLS}$ are empty. (We treat relation i : `ap` $\xrightarrow{IN}_{c}$ $\widehat{o}$ as encoding a set of $\widehat{o}$s for given $i, \texttt{ap}, c$; relation $i \xrightarrow{CALLS}^{c}_{c'}$ `meth` as encoding a set of `meth`s for given i, c, c', etc.)

Therefore, we start from a trivially over-approximate state.

Importantly, the inductive step does *not* hold for a single application of a rule. Intermediate states of evaluation may not be over-approximate: an element may enter a set before the rest of its contents. (For instance, consider a statement `v = u` and prior points-to set $\{\widehat{o_1}, \widehat{o_2}\}$ for `u`. A single application of the MOVE rule for $\widehat{o_1}$ will leave the points-to set of `v` in a non-over-approximate state: the set will be missing the $\widehat{o_2}$ value.)

Thus, the inductive step applies to states after past rules have been evaluated fully. Consider a rule R as a monotonic update to a set of values s. That is, $R(s) \supseteq s$. A rule has been fully evaluated at fixpoint, i.e., when $R(s) = s$. The next inductive step considers the state after a full evaluation of any rule.

The inductive step of the proof is captured in a lemma:

Lemma 1. *The analysis rules preserve soundness under full single-rule evaluation. That is, if relations* $\xrightarrow{IN}$, $\xrightarrow{OUT}$, *and* $\xrightarrow{CALLS}$ *encode over-approximate points-to/call-target sets before a full evaluation of a rule, they will encode over-approximate sets after a full evaluation.*

Proof sketch of Lemma 1. The lemma is established by exhaustive examination of the rules. We mentioned key parts of the reasoning in our earlier presentation of the rules. All rules over complex access paths (i.e., of length ≥ 2) affect the heap and require the "concurrency model" and "object isolation" assumptions of Section 7.1.3. Rules on plain-variable access paths use the "stack-frame isolation" assumption. Every rule is careful to produce values for points-to/call-target sets only if all input sets are non-empty (i.e., guaranteed over-approximate and bounded), and to consider all possible such values. For rules CALL, ARGS, and RET the lemma holds only under the previously-stated assumption on the $\mathcal{NC}$ constructor: the pair $(\texttt{meth}, c')$ needs to uniquely identify invocation instruction i and context c. Consider, for example, rule ARGS. We need to establish that the points-to set j : $\texttt{ap}[arg^i_n/arg^{\texttt{meth}}_n] \xrightarrow{IN}_{c'}$ is over-approximate given that i : $\texttt{ap} \xrightarrow{IN}_{c}$ is. (The rule form makes the former be a superset of the latter, we need to reason that they are actually the same set.) Instruction j uniquely identifies method `meth` and actual-to-formal access-path rebasing can never merge access paths (since different formal variables cannot have the same names). If c' and `meth` arise for only a single call-site and caller-context pair, (i, c), then the property holds. ■

The lemma establishes the inductive step of our proof. The sets computed by the analysis are initially over-approximate and remain over-approximate after every full evaluation of a single rule. At fixpoint, when full evaluation of any rule no longer changes the output sets, the property holds, concluding the theorem's proof. ■

An interesting question is whether *any* evaluation order of the rules is guaranteed to yield sound points-to sets at fixpoint. The answer is "almost yes". All but one of the analysis rules are monotonic (in the usual domain of sets, i.e., with the empty set at the bottom), therefore yield a confluent evaluation: any order will yield the same result at fixpoint. (We have a machine-checked proof of the latter property, by encoding the rules in the Datalog language, which allows only recursion through monotonic inferences.) The single exception is the RET rule. There is hidden non-monotonicity in the $\forall$ iteration over call-graph edges, which contains an implication. If the CALL rule is not fully evaluated when the RET rule applies, it is possible to produce points-to sets that will later be invalidated, because more callees will be discovered (for whom the points-to relationship does not hold for the given access path). Therefore, for soundness to hold, the analysis rules have to always apply in such a fashion that the CALL rule is fully evaluated (not globally but on its own, per the

earlier definition) before the RET rule is considered. This evaluation order should be enforced by any sound implementation of the rules of Figure 7.1.

Based on the above observation on the rules' monotonicity, we also establish our laziness result.

Theorem 2. *A points-to set encoded in our analysis relations grows monotonically, as long as the* RET *rule is applied only during local fixpoints (i.e., after full evaluation) of the* CALL *rule.*

7.4 Implementation and Discussion

We have implemented defensive analysis in the Datalog language and integrated it with DOOP. The full implementation consists of over 400 logical rules, yet the minimal model of Section 7.3 captures well its essential features. We also completed a second, largely equivalent, implementation on the Soufflé Datalog engine [133]. Both implementations are publicly available in the DOOP repository.

The defensive analysis model admits several enhancements and refinements, as well as gives rise to observations. We discuss such topics next, especially noting those that pertain to our full-fledged implementation of the analysis.

Observations. A defensive analysis is naturally modular, yet the question is whether it can produce useful results. The analysis can be applied to any subset of the code of an application or library and it will produce sound inferences. Omitting code merely means that more points-to sets will end up being empty: the analysis only infers points-to sets when an upper-bound of their contents is known based on the current code under analysis. This defensive approach, however, may end up computing too many empty points-to sets. Therefore, the key quality metric is that of the analysis's *coverage*: for how many program elements (e.g., local variables) can the analysis produce non-empty points-to information? Coverage has similarly been used as a key metric in other work that infers specifications modularly [19].

Additionally, a defensive analysis is not in competition with a conventional, unsound analysis, but instead complements it. The defensive analysis computes which of the points-to sets have known upper bounds and which are potentially undetermined. If, instead of an empty set, a client desires to receive the (incomplete) subset of known contents for non-bounded points-to sets, the results of the two analyses can be trivially combined.

Pragmatics. With minor adaptation, the analysis logic can work on static single assignment (SSA) input. Our implementation is indeed based on an SSA intermediate language. The benefit is that for trivial access paths (just a single variable) points-to information does not need to be kept per-instruction: the points-to set remains unchanged, since the variable is not re-assigned.

A full-fledged analysis should cover more language features than the model of Section 7.3. Our implementation handles, in a manner similar to the earlier rules, features such as static

and special method invocations, static fields, final fields, constructors (also implicitly initializing fields to `null`), and more. Of particular note are `final` instance and static fields, which allow propagating information in a lot more settings (e.g., even when the analysis context depth has been reached and points-to sets would normally default to empty after a call).

Expanding the Analysis Reach. Defensive analysis is naturally pessimistic. Its key feature is that it will populate points-to sets only when it can establish that they are bounded. However, the analysis uses simplistic techniques to establish such boundedness, i.e., it recognizes guaranteed-safe cases.

There are several sound inferences that the analysis could make but the model of Section 7.3 does not. However, the principle remains: when the analysis errs in modeling something precisely (as all static analyses will do, for different cases), it will err on the side of being conservative, i.e., compute nothing. Although defensive analysis will never reach the inferences of an unsound analysis (even without any opaque code), it can be enhanced to approach it. Arbitrarily complex mechanisms can be added to increase the coverage of the analysis (i.e., the true properties it can infer precisely):

- The rule shown earlier for control-flow merge points is conservative. Information propagates at control-flow merge points if all of the predecessors have some points-to information for the access path in question. This condition is too strict: several predecessors will not have points-to information for an access path simply because the access path is not even assigned in the predecessor branch (e.g., it is based on a local variable that is set on a different branch only). Consider a program fragment:

  ```
  x.f = new A();
  while (...) {
      y = x.f;
  }
  ```

  The head of the loop has two control-flow predecessors: one due to linear control flow and one due to the loop back-edge. However, the loop itself does not change the points-to set of `x.f`. It is too conservative to demand that the back-edge also have a bounded points-to set for `x.f` before considering the linear control-flow edge.

  In our implementation we have special support for detecting that a program path does not affect an access path. We use this to limit the $\forall$ quantification of the rule to range over "relevant" predecessors. We note that this scenario only applies to complex access paths in practice, due to the SSA form of our input.

- When an unknown method call is encountered, the analysis assumes worst-case behavior with respect to its heap information. This can be relaxed arbitrarily by modeling system methods and annotating them appropriately. Possible information about calls includes "this library call does not affect user-level objects", "this method only affects its arguments", "this method does not affect static variables", etc. Additional manual modeling includes library collections (including arrays) which can be represented as abstract objects.

Our current implementation does some minimal modeling of library collections and annotates only a handful of methods, as a proof-of-concept. A representative example is that of method `Float.floatToRawIntBits`. This native method is called by the implementation of the `put` operation in Java `HashMaps` and, since it is opaque, would prevent all propagation of points-to information beyond a `put` call.

- The analysis coverage can be expanded by employing it jointly with a *must-alias* analysis [22, 67, 170], an *escape* analysis [12, 37], and a *thread-escape* analysis. A must-alias analysis will increase the applicability of the rule for heap loads, and can be combined with the rule for heap stores to enable more strong updates. An escape analysis will result in less conservativeness in the propagation of information to further instructions (i.e., in frame rules). A thread-escape analysis can help relax our concurrency model. We currently support simple, conservative versions of all three analyses in our implementation, but do not enable them by default.

Why access paths? Our defensive analysis is access-path based, as opposed to instance-field based. That is, instead of inferences of the form "abstract object $\widehat{o_1}$ points to $\widehat{o_2}$ via field `f`" our analysis makes inferences of the form "program expression `v.f` points to $\widehat{o_2}$". We conjecture that this is a necessity for a sound analysis. The issue is one of logical quantification. What is the meaning of the sentence "abstract object $\widehat{o_1}$ points to $\widehat{o_2}$ via field `f`"? A natural definition is "*some* concrete object mapping to abstract object $\widehat{o_1}$ points via field `f` to some concrete object mapping to $\widehat{o_2}$." However, this definition is too weak to allow sound inferences at the point of a load instruction. An alternative definition is "*all* concrete objects mapping to abstract object $\widehat{o_1}$ point via field `f` to some concrete object mapping to $\widehat{o_2}$." However, this inference is impossible to establish soundly at any program point: even if all concrete instances of the abstract object satisfy this property somewhere (e.g., at the end of a constructor), there is virtually never a single point at which all past concrete objects (mapping to the same abstract one) simultaneously satisfy the inference. It is possible that, in future work, an analysis can distinguish the most recent instance of an object [7]. At present, however, the access path abstraction seems like a particularly good fit for our defensive analysis logic.

The analysis formulation of Section 7.3 assumes that all access paths exist in advance. In the implementation, we instead create access paths lazily, as needed (an idea already explored in previous chapters). The formulation is suitable for integration with other analyses that use access paths. A particularly good fit is the must-alias analysis on access paths of Chapter 5. Such an analysis improves our may-point-to analysis in several ways: it helps analyze more load instructions; it helps perform strong updates at store instructions (when the base variable of the store is a must-alias with the base of an access path). The may-point-to and must-alias analyses can benefit from each other when run in iteration: a sound must-alias analysis requires a sound call-graph over-approximation and vice versa. We have integrated such a must-alias analysis in the implementation, but do not enable it by default due to currently high cost.

Context depth. As seen earlier, a defensive analysis may compute empty (undetermined) points-to sets because it has reached its maximum context depth. It is worth pointing out,

however, that method calls *further away* than the maximum context depth can influence the points-to inferences of a method. For an easy example, consider the case of a large number, N, of methods that form a call chain and unconditionally return to their callers what their callee returns to them. If the final (N-th) method returns a new object, then that object will propagate all the way back to the first method of the call chain, regardless of the maximum context depth, D. The limitation of context depth only concerns properties that *depend on* conditions established more than D calls back in the call-stack.

7.5 Evaluation

There are five research questions that our evaluation seeks to answer:

- **RQ1:** Does defensive analysis produce coverage for large parts of realistic programs? Or do points-to sets overwhelmingly stay empty?
- **RQ2:** Does the coverage of defensive analysis benefit from its advanced features (i.e., inter-procedural handling, as well as handling of control-flow merging)?
- **RQ3:** Does defensive analysis have an acceptable running time, given that it is flow-sensitive and context-sensitive?
- **RQ4:** Does defensive analysis yield results that can benefit a client that requires soundness, such as an optimization?
- **RQ5:** Can benefits be obtained for a fully relaxed concurrency model, as opposed to the model of Section 7.1.3?

Setup. Since defensive analysis is a unique beast, it is indeed an interesting question to ask what it can be compared against. As closest comparable (though still a very dissimilar analysis) we chose to compare to a highly-precise conventional analysis with state-of-the-art best-effort soundness: a 2-object-sensitive/heap-sensitive analysis (2objH) with reflection support. This is the most precise analysis in the Doop framework that still manages to scale to the majority of the DaCapo benchmarks. We use static best-effort reflection handling (`--enable-reflection-classic` flag), i.e., the analysis tries to statically resolve all reflection calls based on string matching.

We analyze, under JDK 1.7.0_75, the DaCapo benchmark programs [11] v.2006-10-MR2 as well as v.9.12-Bach. The 9.12-Bach version contains several different programs, as well as more recent versions of some of the same programs. (We show results for all of the v.2006-10-MR2 benchmarks and for those of the v.9.12-Bach benchmarks that could be analyzed by the Doop framework in under 3 hours.) We also use the two non-Android benchmarks (NTI, jFlex) from the Julia set by Nikolić and Spoto [110]. We use the LogicBlox Datalog engine, v.3.10.14, on a Xeon E5-2667 v.2 3.3GHz machine with only one thread running at a time and 256GB of RAM.

Defensive analysis is run with a 5-call-site-sensitive context (*5def* for short). **3** instances (of **44** total) did not finish with the default precision in 3 hours: the 2objH baseline did not finish for jython and h2; xalan did not finish for the 5def analysis. In these cases we used lower precision: context-insensitive for the unsound analysis and 4-call-site-sensitive (*4def*) for defensive. We use diacritical marks (* and ˆ) in the figures to remind the reader of the different analysis setting for these benchmarks.

Coverage. Figure 7.2 shows the coverage of defensive analysis, i.e., the number of non-empty points-to sets (for local variables) computed for all benchmarks. The input program is in SSA form, therefore the points-to sets for variables are a normalized representation of all points-to information in the program: they reflect the analysis-computed values for all program expressions, separately for each control-flow point.

The analysis yields non-empty points-to sets for a significant portion of each program—the median benchmark has **45.6%** of variables with points-to information for *some* context, while **35.5%** have points-to information for a context INIT (i.e., unconditionally).[6] It is worth emphasizing that conditional points-to guarantees (under *some* context) are valuable in a defensive analysis: they are often the best any analysis can ever do! Recall our earlier discussion of Section 7.2: many of the useful inferences of a defensive analysis will be under *some* context even when the inference holds under *all known* contexts in existing code. No analysis can preclude the existence of other callers in opaque, and possibly not yet existing, code. Such callers can arise in dynamically generated code and can invoke existing methods, e.g., using reflection.

Thus, the defensive analysis achieves a large proportion of the benefits of an unsound analysis, while guaranteeing these results against uses of opaque code. We can answer **RQ1** affirmatively: defensive analysis covers a large part of realistic programs (over one-third unconditionally; close to one half under specific calling conditions), despite its conservative nature.

Comparison with intra-procedural. We have earlier referred to the "easy", intra-procedural parts of the analysis reasoning: what a compiler or VM would likely do to perform sound local data-flow analysis. This is the subject of **RQ2**, also answered by Figure 7.2. The figure includes results for an intra-procedural baseline analysis that captures the low-hanging fruit of sound reasoning: local variables that directly or transitively (via "move" instructions) get assigned an allocated object. That is, the "Intra-proc Sound" analysis is otherwise the same as the full "defensive" logic, with the exception of the new "interesting" cases (control-flow merging, heap manipulation, and inter-procedural propagation).

The result answers **RQ2** affirmatively: defensive analysis has significantly higher coverage than the baseline intra-procedural analysis. (And the difference only grows when considering an actual client, in later experiments.) Although the benefit is not broken down further in the figure, the handling of method calls alone (i.e., rules CALL, ARGS and RET) is responsible for the lion's share of the difference between the full defensive analysis and the intra-procedural

[6] If a variable has a points-to value for context INIT, then it also has that value under every specific context that arises for the variable. Therefore, points-to sizes for INIT are always lower than conditional, context-specific sizes.

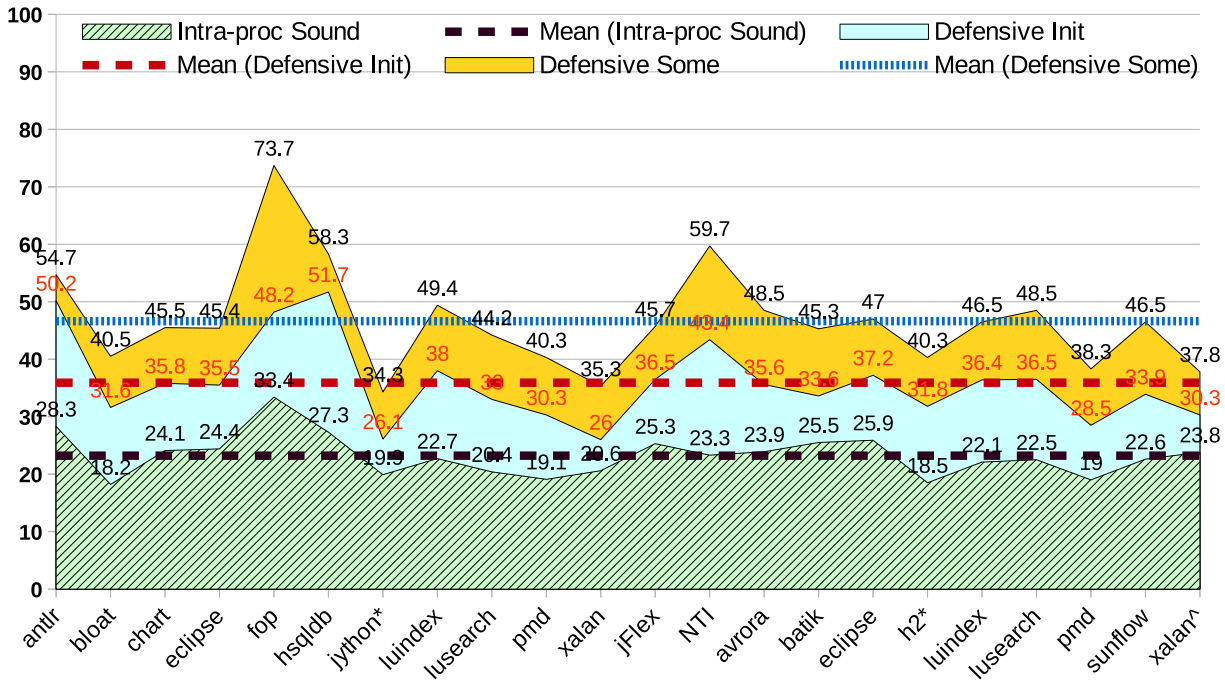


Figure 7.2: Percentage of application variables (deemed reachable by baseline 2objH analysis) that have non-empty points-to sets for defensive analysis under some context and INIT context (no assumptions). Intra-procedural sound points-to analysis (defensive minus the complex cases) shown as baseline. Arithmetic means are plotted as lines.

sound analysis.

Running time. Figure 7.3 shows the running times of the analysis, plotted next to that of 2objH, for reference. Although the two analyses are dissimilar, 2objH is qualitatively the closest one can get to defensive analysis with the current state of the art: it is an analysis with high precision, run with best-effort soundness support. Therefore, 2objH can serve as a realistic point of reference. As can be seen, the running times of defensive analysis are realistically low, although its flow-sensitive and 5-call-site-sensitive nature suggests it would be a prohibitively heavy analysis. This answers **RQ3** and confirms the benefits of laziness: a defensive analysis that only populates points-to sets once they are definitely bounded, achieves scalability for deep context.

Client analysis: devirtualization. Our baseline analysis, 2objH, is highly precise and effective in challenges such as devirtualizing calls (resolving virtual calls to a single target method). On average, it can devirtualize **89.3%** of the calls in the benchmarks studied (min **78.5%**, max **95.2%**). However, these results are unsound and a compiler cannot act upon them. For optimization clients, such as devirtualization, soundness is essential. Using sound results, a JIT compiler can skip dynamic tests (of the inline caching optimization) for all calls that the analysis soundly covers.

Figure 7.4 shows the virtual calls that defensive analysis devirtualizes, as a percentage of

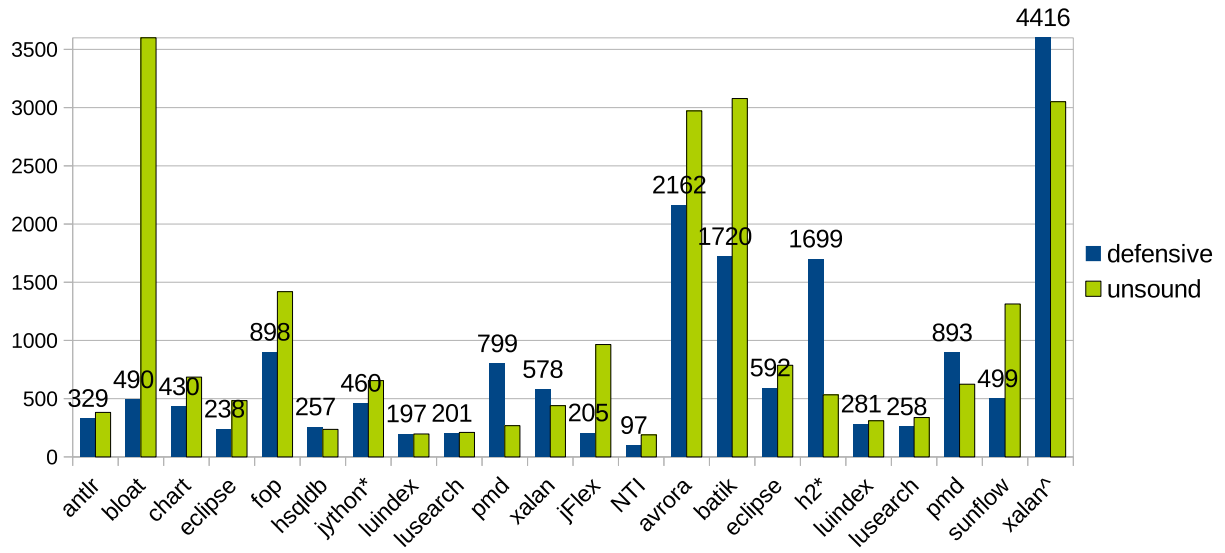


Figure 7.3: Execution time (in seconds) of defensive analysis, with running time of 2objH (with unsound reflection handling) shown as a baseline. Labels are shown for defensive analysis only to avoid crowding the plot.

those devirtualized by the unsound analysis.

As can be seen, defensive analysis manages to recover a large part of the benefit of an unsound analysis (median **44.8%** for optimization under a context guard, **38.7%** for unconditional, INIT context, optimization), performing much better than the baseline intra-procedural must-analysis (at **14.6%**). This answers **RQ4** affirmatively: the coverage of defensive analysis translates into real benefit for realistic clients.

Concurrency model. A compiler (JIT or AOT) author may (rightly) remark that the concurrency model of Section 7.1.3 is not appropriate for automatic optimizations. The Java concurrency model permits a lot more relaxed behaviors, so the analysis is not sound for full Java as stated. However, the benefit of defensive analysis is that it starts from a sound basis and can add to it conservatively, only when it is certain that soundness cannot possibly be violated. Accordingly, we can remove the assumption that all shared data are accessed while holding mutexes, by applying the load/store rules only when objects trivially do not escape their allocating thread. We show the updated numbers for the devirtualization client (now fully sound for Java!) in Figure 7.5. The difference in impact is minimal: **43%** of virtual call sites can be devirtualized conditionally, under some context, while **36%** can be devirtualized unconditionally. This helps answer **RQ5**: defensive analysis can yield actionable results for a well-known optimization, under the Java memory model, for a large portion of realistic programs.

Points-to set sizes. Finally, it is interesting to quantify the precision of the defensive analysis, for the points-to sets it covers. This precision is expected to be high, since defensive analysis is flow- and context-sensitive, but exact figures help put it in perspective.

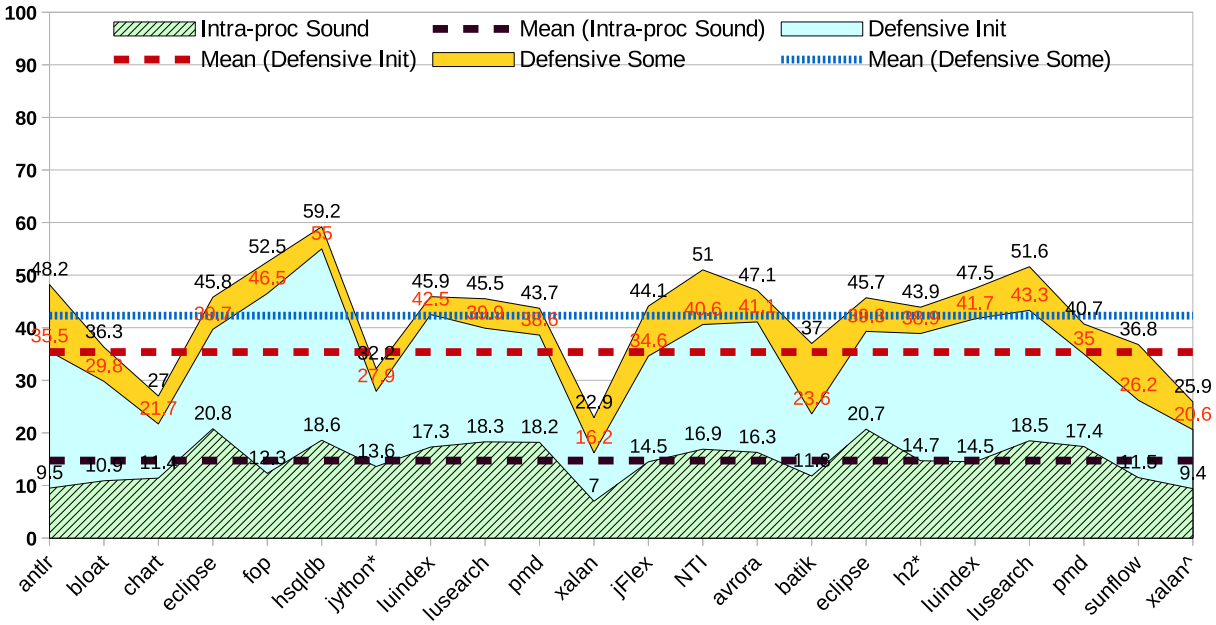


Figure 7.4: Virtual call sites that are found to have receiver objects of a single type. These call sites can be soundly devirtualized. Numbers are shown as percentages of devirtualization achieved by unsound 2objH analysis.

		defensive	2objH	2objH	defensive		
DaCapo 2006-10-MR2	antlr	1.01	1.10	3.04	1.05	avrora	DaCapo 9.12-Bach
	bloat	1.02	2.12	1.05	1.04	batik	
	chart	1.09	1.09	1.53	1.07	eclipse	
	eclipse	1.06	1.31	2.07	1.04	h2*	
	fop	1.00	1.03	1.04	1.01	luindex	
	hsqld	1.01	1.04	1.08	1.03	lusearch	
	jython*	1.01	6.05	1.04	1.01	pmd	
	luindex	1.02	1.02	1.08	1.05	sunflow	
	lusearch	1.04	1.06	1.19	1.04	xalan^	
	pmd	1.01	1.05	1.02	1.01	jFlex	
	xalan	1.05	1.12	1.03	1.03	NTI	
			"defensive" mean	**1.03**			
			"2objH" mean	1.51			

Table 7.1: Average number of abstract objects pointed-by per variable, for variables for which both analyses compute results.

Table 7.1 shows average points-to set sizes for the defensive analysis vs. the 2objH analysis. The sets (excluding null values) are computed over variables covered by both analyses, for non-empty defensive analysis sets and under context INIT of the defensive analysis, i.e., unconditionally. (The numbers are for the simplistic concurrency model, but remain

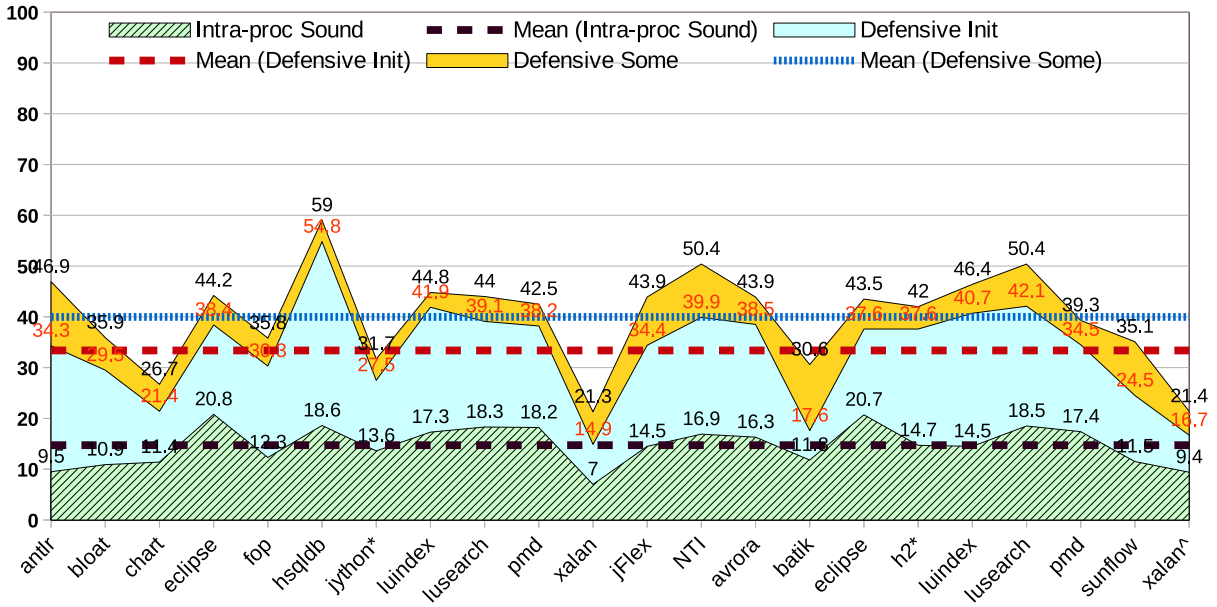


Figure 7.5: Virtual call sites (percentage of 2objH) that are found to have receiver objects of a single type. Updates Figure 7.4, this time with soundness under a relaxed memory model.

unchanged to two significant digits for the relaxed concurrency model.)

As can be seen, the defensive analysis is highly precise when it produces non-empty points-to sets, typically yielding points-to set sizes very close to 1. 2objH is also a very precise analysis (for variables with bounded points-to sets), so it remains competitive, yet clearly less precise. Notably, points-to set sizes close to 1 are the Holy Grail of points-to analysis: such precision is actionable for nearly all conceivable clients of a points-to analysis.

7.6 Summary

Static analysis has long suffered from unsoundness for perfectly realistic language features, such as reflection, native code, or dynamic loading. We presented a new analysis architecture that achieves soundness by being *defensive*. Despite its conservative nature, the analysis manages to yield useful results for a large subset of the code in realistic Java programs, while being efficient and scalable. Additionally, the analysis is modular, as it can be applied to any subset of a program and will yield sound results.

We expect this approach to open significant avenues for further work. The analysis architecture can serve as the basis of other sound analysis designs. The defensive analysis itself can be combined with several other analyses (may-escape, must-alias) that have so far been hindered by the lack of a sound substrate.

Part III

Epilogue

8. RELATED WORK

> Hermits United. We meet up every 10 years, swap stories about caves. It's good fun... for a hermit.
>
> *The 10th Doctor* - Doctor Who

8.1 Hybrid-Context Sensitivity

We have discussed directly related work throughout Chapter 3. Here we selectively mention a few techniques that, although not directly related to ours, offer alternative approaches to sweet spots in the precision/performance tradeoff.

Special-purpose combinations of context sensitivity have been used in the past, but have required manual identification of classes to be treated separately (e.g., Java collection classes, or library factory methods). An excellent representative is the TAJ work for taint analysis of Java web applications [157]. In contrast, we have sought to map the space and identify interesting hybrids for general application of context sensitivity, over the entire program.

The analyses we examined are context-sensitive but flow-insensitive. We can achieve several of the benefits of flow sensitivity by applying the analysis on the static single assignment (SSA) intermediate form of the program. This is easy to do with a mere flag setting on the Doop framework. However, the impact of the SSA transformation on the input is minimal. The default intermediate language used as input in Doop (the Jimple representation of the Soot framework [158, 159]) is already close to SSA form, although it does not guarantee that every variable is strictly single-assignment without requesting it explicitly. Published work by Lhoták and Chung [82] has shown that much of the benefit of flow sensitivity derives from the ability to do strong updates of the points-to information. Lhoták and Chung then exploited this insight to derive analyses with similar benefit to a full flow-sensitive analysis at lower cost.

A demand-driven evaluation strategy reduces the cost of an analysis by computing only those results that are necessary for a client program analysis [58, 150, 151, 170]. This is a useful approach for client analyses that focus on specific locations in a program, but if the client needs results from the entire program, then demand-driven analysis is typically slower than an exhaustive analysis.

Reps [123] showed how to use the standard magic-sets optimization to automatically derive a demand-driven analysis from an exhaustive analysis (like ours). This optimization combines the benefits of top-down and bottom-up evaluation of logic programs by adding side-conditions to rules that limit the computation to just the required data.

An interesting recent approach to demand-driven analyses was introduced by Liang and Naik [91]. Their "pruning" approach consists of first computing a coarse over-approximation of

the points-to information, while keeping the provenance of this derivation, i.e., recording which input facts have affected each part of the output. The input program is then pruned so that parts that did not affect the interesting points of the output are eliminated. Then a precise analysis is run, in order to establish the desired property.

8.2 Introspective Analysis

The effort to tune the context sensitivity of an analysis is pervasive in the literature. Nevertheless, most approaches fundamentally differ from ours of Chapter 4, either by trying to vary context sensitivity based on syntactic properties or by trying to focus on only a part of the program that matters for answering a given query. In contrast, we attack the context-sensitive scalability problem head-on, in the all-points points-to analysis setting, with context used all over the program and library.

Typical scalable points-to analysis frameworks such as Wala [43] and Doop [16] employ a multitude of low-level heuristics for tuning the precision and scalability of an analysis. These include using extra context for collection classes, using a heap context for arrays in an analysis without a context-sensitive heap, allocating strings or exceptions context-insensitively, treating library factory methods with deeper context, etc. Such heuristics are typically user-selected and prominent in the documentation of the respective frameworks, and have also appeared in the literature (e.g., [69, 157]). However, all such approaches are mere hard-wired heuristics and do not address the major scalability problem that our approach aims to solve. The scalability issues identified in earlier literature and discussed throughout this paper are present after all such heuristics have been employed.

A more general approach is the hybrid-context sensitivity of Chapter 3. Such a hybrid analysis attempts to emulate call-site sensitivity for static method calls and object sensitivity for dynamic calls. The approach becomes interesting when context is deep (e.g., how are context elements merged when a dynamic call is made inside a static call?). Nevertheless, the hybrid-context sensitivity approach does not change the essence of the problem we are trying to solve. For hard-to-analyze applications, hybrid context-sensitive algorithms are equally unscalable as their component algorithms. For the purposes of our experimental study, which only tests the scalability of heavyweight benchmarks, hybrid-context sensitivity is virtually indistinguishable from object sensitivity.

In recent years there have been many more instantiations of introspective analysis, with very different metrics of cost and benefit. These modern instantiations outperform our original Heuristic-A and Heuristic-B but keep the same flavor: Zipper [87] aims to achieve mostly-guaranteed precision with heuristically better scalability, whereas Scaler [88] achieves guaranteed scalability and typically significantly better precision than a context-insensitive analysis.

More interesting applications of selective context sensitivity have been explored in the context of *demand-driven* pointer analysis. A demand-driven evaluation strategy reduces the cost of an analysis by computing only those results that are necessary for a client program analysis [58, 150, 151, 170]. This is a useful approach for client analyses that focus on spe-

cific locations in a program, but if the client needs results from the entire program, then demand-driven analysis is typically slower than an exhaustive analysis.

In the demand-driven space, refinement-based analyses have been used primarily in the work of Sridharan and Bodík [150] and of Liang and Naik [91]. Sridharan and Bodík introduce refinement-based analysis as a way to adaptively increase the precision characteristics of an existing analysis algorithm when a client analysis is not satisfied with the result. The approach allows turning on field sensitivity, as well as higher call-site sensitivity for an analysis algorithm. Yet, unlike ours, it is not a general approach that can apply to any kind of context and a large number of different algorithms. Liang and Naik's "pruning" approach consists of first computing a coarse over-approximation of the points-to information, while keeping the provenance of this derivation, i.e., recording which input facts have affected each part of the output. The input program is then pruned so that parts that did not affect the interesting points of the output are eliminated. Then a highly context-sensitive precise analysis is run, in order to establish the desired property. This approach is similar to introspective context sensitivity in that the analysis is run twice and a separate query over the first-run result determines the second run's characteristics. Nevertheless, our approach requires no provenance computation (which is unlikely to scale for an all-points analysis) and works even when we want answers for the entire program—i.e., when pruning is not possible.

Both of the above demand-driven approaches can be viewed as complements of our introspective context sensitivity. In the demand-driven world, it is possible to estimate the *benefit* that a more precise analysis may yield: either the client is happy with the current level of precision (which implies there is no further benefit to be obtained) or it is not, in which case more precision should be added. In our all-points pointer analysis problem we have no such information. This motivates our *cost*-based heuristics, which attempt to estimate "what can go wrong" when more precision gets added, as opposed to "what can be gained", as in demand-driven techniques.

8.3 Must-Alias Analysis

Logical Model

There are several approaches in the literature that present must-analyses in the pointer analysis setting or employ them in a may-analysis. Our approach is a must-alias analysis applied to Java bytecode, but conceptually it is distinguished by its minimizing the distance between the implementation and the declarative specification.

Ma et al. [98] present an algorithm for null-pointer dereference detection using a context-insensitive may-alias and a must-alias analysis; the latter is used to increase the precision of the former, by enabling strong updates when possible.

Nikolić and Spoto [110] present a must-alias analysis that tracks aliases between program expressions and local variables (or stack locations, since they analyze Java bytecode, which is a stack-based representation). The analysis is related to ours both because of its application

to Java bytecode and because it is constraint-based: the analysis is a generator of constraints, which are subsequently solved to produce the analysis results. Abstractly, this is a relative of our Datalog-based approach, but it is unclear how the two may compare in terms of engineering tradeoffs.

Hind et al. [60] present a collection of pointer analysis algorithms. Among them, the most relevant to this work is a flow-sensitive interprocedural pointer alias analysis. The authors optimistically produce *must* information for pointers to single non-summary objects.

Emami et al. [39] present an approach that simultaneously calculates both must- and may-point-to information for a C analysis. Their empirical results "show the existence of a substantial number of definite points-to relationships, which forms very valuable information"—much in line with our own experience.

The analysis of [33] is essentially a flow-sensitive may-point-to analysis that performs strong updates, as it maps *access paths* to *heap objects* (abstracted by their allocation sites). The approach uses a flow-insensitive may-point-to analysis to bootstrap the main analysis. However, it provides no *definite* knowledge of any sort, since the aim is to increase the precision of the may-analysis. For instance, even if an access path points to a single heap object, according to the De and D'Souza analysis, there is no *must* point-to information derived, since this object could be a summary object (i.e., one that abstracts many objects allocated at the same allocation site). To reason about such cases, other approaches, such as the more expensive shape analysis algorithms [132], additionally maintain summary information per heap object. In this way, they allow must point-to edges to exist only if the target is definitely not a summary node.

Must- information is often computed in conjunction with a client analysis. One of the best examples is the typestate verification of Fink et al. [45], which demonstrates the value of a must-analysis and the techniques that enable it.

An approach for integrating *must* point-to reasoning in an analysis is to propagate such information only at instructions where we know that the given heap allocation target still refers to the last object allocated at that site [2]. Thus, an execution path that may create another object at the same site (such as when reaching the end of the loop) would invalidate any previous must-point-to facts (i.e., it will stop them from propagating any further).

Generally, must-analyses can vary greatly in sophistication and can be employed in an array of different combinations with may-analyses. The analysis of Balakrishnan and Reps [7], which introduces the *recency abstraction*, distinguishes between the most recently allocated object at an allocation site (a concrete object, allowing strong updates) and earlier-allocated objects (represented as a summary node). The analysis additionally keeps information on the size of the set of objects represented by a summary node. At the extreme, one can find full-blown shape analysis approaches, such as that of Sagiv et al. [132], which explicitly maintains must- and may- information simultaneously, by means of three-valued truth values, in full detail up to predicate abstraction: a relationship can definitely hold ("must"), definitely not hold ("must not", i.e., negation of "may"), or possibly hold ("may"). Summary and concrete nodes are again used to represent knowledge, albeit in full detail, as captured by arbitrary predicates whose value is maintained across program statements, at the cost of a

super-exponential, worst-case complexity.

Jagannathan et al. [67] present an algorithm for must-alias analysis of functional languages. The algorithm adapts must-alias insights to the setting of captured variables in closures. For instance, must-alias information for non-summary objects permits strong updates, which the authors find to improve analysis precision. We employ must-alias analysis results quite similarly in applications of our model analysis.

Data Structures

Our optimized data structure is (partly) based on the observation that must-alias sets are equivalence classes. This is not the first time that a data structure that efficiently implements equivalence classes has been used to speed up pointer analysis. Most notably, a Steensgaard-style (or *unification-based*) [153] analysis computes may-point-to sets that are equivalence classes. This means that points-to sets are disjoint—if two points-to sets are found to possibly overlap, they get unified. This loses precision (relative to a standard subset-based points-to analysis) but enables the algorithm to use union-find trees for a very efficient representation.

Another optimized data structure often used in pointer analysis is the *constraint graph*: a graph with nodes denoting pointer variables and an edge between nodes `p` and `q` denoting flow (e.g., a direct assignment) from variable `p` to variable `q`. Online cycle elimination by Fändrich et al. [41] detects cycles in the constraint graph and collapses all nodes in a cycle into a representative node, since such nodes will have identical points-to information. The technique of Nasre [109] extends such constraint graph reasoning based on the observation that if two nodes have the same dominator in the constraint graph, then they are clones: the values flowing to them are (only) those of the dominator node. Several other constraint graph optimizations are applied off-line (i.e., before the points-to analysis runs). Prime examples of such techniques are Rountev and Chandra's [131] and Hardekopf and Lin's [55]. (Hardekopf and Lin have also applied similar ideas in a hybrid online/offline setting [57].) Both of these techniques perform an off-line detection of equivalent points-to sets and use this knowledge to eliminate redundant work in subsequent points-to computations. Our data structure can be seen as somewhat analogous to constraint-graph techniques, in the sense that we do not compute the flow of objects or the fully expanded set of all possible alias pairs. Instead, we compute the "wiring" (i.e., the alias relationships, locally, that the program induces) and keep the alias information in condensed form, until it needs to be queried by a client analysis.

Another conceptual relative of our data structure is the model presented by Madhavan et al. [99] for modular may-analyses. That model is similar in that it invents abstract nodes for heap objects that resemble ours (without the equivalence-class nature). The Madhavan et al. approach aims to achieve modular reasoning, i.e., to model the heap effects of a method without knowing its calling environment. To do so, the approach creates abstract nodes that represent concepts such as "whichever object variable `x` may point to". Our data structure has nodes with a similar meaning, however we also take advantage of the "must" nature of the analysis to merge nodes, every time the same access path can reach both.

8.4 Defensive Analysis

There is certainly past work that attempt to ensure a sound whole-program analysis, but none matches the generality and applicability of our approach. We selectively discuss representative approaches.

The standard past approach to soundness for a careful static analysis has been to "bail out": the analysis detects whether there are program features that it does not handle soundly, and issues warnings, or refuses to produce answers. This is a common pattern in abstract-interpretation [29] analyses, such as Astrée [35], which have traditionally emphasized sound handling of conventional language features. However, this is far from a solution to the problem of being sound for opaque code: refusing to handle the vast majority of realistic programs can be argued to be sound, but is not usefully so. In contrast, our work handles *all* realistic programs, but returns partial (but sound) results, i.e., produces non-empty points-to sets for a subset of the variables. It is an experimental question to determine whether this subset is usefully large, as we do in our evaluation.

Hirzel et al. [61, 62] use an online pointer analysis to deal with reflection and dynamic loading by monitoring their run-time occurrence, recording their results, and running the analysis again, incrementally. However, this is hardly a *static* analysis and its cost is prohibitive for precise (context-sensitive) analyses, if applied to all reflection actions.

Lattner et al. [80] offer an algorithm that can apply to incomplete programs, but it assumes that the linker can know all callers (i.e., there is no reflection—the analysis is for C/C++) and the approach is closely tied to a specific flow-insensitive, unification-based analysis logic [153], necessary for simultaneously computing inter-related points-to, may-alias, and may-escape information.

Sreedhar et al. [149] present the only past approach to explicitly target dynamic class loading, although only for a specific client analysis (call specialization). Still, that work ends up making many statically unsound assumptions (requiring, at the very least, programmer intervention), illustrating well the difficulty of the problem, if not addressed defensively. The approach assumes that only the public API of a "closed world" is callable, thus ignoring many uses of reflection. (With reflection, any method is callable from unknown code, and any field is accessible.) It "[does] not address the Java features of reloading and the Java Native Interface". It "optimistically assumes" that "[the extant state of statically known objects] remains unchanged when they become reachable from static reference variables". It is not clear whether the technique is conservative relative to adversarial native code (in system libraries, since the JNI is ignored). Finally, the approach assumes the existence of a sound may-point-to analysis, even though none exists in practice!

Traditional conservative call-graph construction (*Class Hierarchy Analysis (CHA)* [34] or *Rapid Type Analysis (RTA)* [6]) is unsound. Such algorithms explore the entire class hierarchy for matching (overriding) methods and consider all of them to be potential virtual call targets. However, even this is not sufficient for a sound static analysis of opaque code: classes can be generated and loaded dynamically during program execution. CHA cannot find target methods that do not even exist statically, yet modeling them is precisely what

is needed for soundness in real-world conditions. For instance, Java applications, especially in the enterprise (server-side) space, employ dynamic loading heavily, and patterns such as *dynamic proxies* have been standardized and used widely since the early Java days.

Furthermore, such heuristic "best-effort" over-approximation is detrimental to analysis precision and performance. CHA is an example of a loose over-approximation in an effort to capture most dynamic behaviors. (Similar loose over-approximations have been proposed, for instance, for reflection analysis [145].) Loose over-approximations compute many more possible targets than those that realistically arise. This yields vast points-to sets that render the analysis heavyweight and useless due to imprecision. (Avoiding such costs is exactly why past analyses have often opted for glaringly unsound handling of opaque code features.) Our lazy representation of "don't know"/"cannot bound" values as empty sets addresses the problem, by keeping all points-to sets compact.

The conventional handling of reflection in may-point-to analysis algorithms for Java [44, 86, 89, 92, 93, 145] is unsound, instead relying on a "best-effort" approach. Such past analyses attempt to statically model the result of reflection operations, e.g., by computing a superset of the strings that can be used as arguments to a `Class.forName` operation (which accepts a name string and returns a reflection object representing the class with that name). The analyses are unsound when faced with a completely unknown string: instead of assuming that *any* class object can be returned, the analysis assumes that *none* can. The reason is that over-approximation (assuming any object is returned) would be detrimental to the analysis performance and precision. Even with an unsound approach, current algorithms are heavily burdened by the use of reflection analysis. For instance, the documentation of the Wala library directly blames reflection analysis for scalability shortcomings [44],[1] and enabling reflection on the Doop framework slows it down by an order of magnitude on standard benchmarks [145]. Furthermore, none of these approaches attempt to model dynamic loading—a ubiquitous feature in Java enterprise applications.

8.5 General Directions in Program Analysis

Finally, in this section, we extend our focus on the broader area of (static) program analysis and automatic program understanding. We touch upon various methodologies that have been introduced in past literature aiming to tackle more or less similar issues to the ones previously discussed.

8.5.1 Control-Flow Analysis (k-CFA)

The term *control-flow analysis* (CFA) commonly describes an algorithm incorporating both data-flow and control-flow reasoning, and more specifically, in which data-flow depends on

[1] The Wala documentation is explicit: "*Reflection usage and the size of modern libraries/frameworks make it very difficult to scale flow-insensitive points-to analysis to modern Java programs. For example, with default settings,* Wala*'s pointer analyses cannot handle any program linked against the Java 6 standard libraries, due to extensive reflection in the libraries.*" [44]

control-flow, and at the same time, control-flow depends on data-flow. In the setup of our work, points-to reasoning plays the part of data-flow and call-graph reasoning plays the part of control-flow. But, control-flow analysis is not strictly confined to the setting of pointer analysis in object-oriented languages. It can also apply to functional languages in a similar problem known as *flow analysis*—in which variables may a value flow to? Both settings introduce a degree of complexity due to higher-order features; functional languages have first-class functions and object-oriented languages have dynamic dispatch.

The origin of k-CFA is found in Shivers's dissertation [140], and was developed to solve the higher-order control-flow problem in λ-calculus-based languages. Consequently, it applies both in functional languages that are explicitly vulnerable to the issue because closures are passed around as first-class values, and in object-oriented languages where the issue is more implicit in method invocations being resolved dynamically.

k-CFA is a well-known family of control-flow analysis algorithms, widely recognized in both communities that popularized the idea of context-sensitive flow analysis. Although, in the setting of object-oriented languages, k-CFA is often made synonymous to "context-sensitive-to-depth-k" it is in fact an algorithm that incorporates context sensitivity along with several other design decisions. Nevertheless, informally, k-CFA commonly refers to a k-call-site-sensitive analysis with a k-context-sensitive heap.

8.5.2 CFL Reachability Formulation

In previous chapters we either formulated pointer analysis algorithms directly as Datalog rules, or in the case of defensive analysis (Chapter 7) as inference rules that can be translated to Datalog rules in a straightforward fashion. This minimalistic approach allows one to better reason about the way that language features interact with each other when designing an analysis algorithm, without the burden of taking implementation details into account. Additionally, such formulation can easily offer time and space complexity bounds for the analysis under development as well as termination guarantees for the resulting algorithm. Furthermore, as an added bonus, the Datalog specification is quite close to the actual implementation of the algorithm in a Datalog engine.

Several past pointer analysis algorithms (as well as other related analyses) have been formulated as a *context-free language (CFL) reachability* problem [119] that can later be translated into Datalog rules. The core idea is that one encodes an input program as a labeled graph, and a specific analysis as a context-free grammar, G. The relation being computed by the analysis (e.g., aliasing information) holds for two nodes of the graph iff there exists a path from one node to the other, such that concatenating the labels on the edges along that path gives a string that belongs to the language $L(G)$ defined by the aforementioned grammar.

In more detail, nodes in the input graph represent program elements such as variables, methods, types, statements, and so on. Edges represent relations between such nodes. For instance, an edge $e(s, t)$ may represent a local assignment statement for the variables encoded in graph nodes s and t. Other common edge encodings include field accesses (load/stores), method invocations, pointer dereferences, etc. The exact choice of domains depends on the

actual analysis that is being designed each time. In order to encode many different input relations simultaneously on the graph, different kinds of edges can be employed. For a given analysis, a context-free grammar G encodes the desired computed attributes as non-terminal symbols, and supplies production rules that express how they relate to the simpler relations represented by graph edges—and terminals in the grammar. The CFL reachability solution in then commonly computed using a dynamic programming algorithm.

The first application of CFL reachability in program analysis intended to solve various interprocedural dataflow-analysis problems [104, 124], but since then is has been used in a wide range of problems, such as: (1) program slicing [125] (more on that later), (2) shape analysis [121], (3) the computation of points-to relations [95, 120, 136, 150, 151, 166], (4) the demand-driven computation of may-alias pairs in a C-like language [170], (5) Andersen-style pointer analysis for Java [151].

As previously mentioned, any CFL reachability problem can be converted to a Datalog program [120], but not the other way around. In that sense, CFL reachability implicitly corresponds to a restricted subset of Datalog programs, often called *chain programs*. As a consequence, the core advantage of CFL reachability is that it describes Datalog programs that allow for more efficient implementations. In this setting, Datalog relations represent labeled edges in the graph. For instance, the fact `P(X, Y)` encodes that node `X` and node `Y` are connected via an edge labeled `P`. A chain program consists of rules of the following form:

$$\texttt{P(X, Y)} \leftarrow \texttt{Q}_0\texttt{(X, Z}_1\texttt{)}, \texttt{Q}_1\texttt{(Z}_1\texttt{, Z}_2\texttt{)}, \ldots, \texttt{Q}_k\texttt{(Z}_k\texttt{, Y)}.$$

The corresponding grammar G provides production rules as the following:

$$P \rightarrow Q_0\ Q_1\ \ldots\ Q_k$$

A more concrete example, related to pointer analysis, is given in the following production rule that describes how information (i.e., abstract objects) flow from the point of allocation, through various assignments, to reach a program expression (e.g., variable).

$$\mathit{FlowsTo} \rightarrow \mathit{Alloc}\ (\ \mathit{Assign}\ )\ *$$

The graph in Figure 8.1 encodes a small code snippet identified by the previous production rule.

Initially, an abstract object `obj` is allocated to variable `x`. Then, `x` is assigned to `y` and `y` is assigned to `z`. Subsequently, an indirect flow is inferred from object `obj` to variable `z`.

Dyck-CFL Reachability. A more restrictive variant is that of *Dyck-CFL reachability*. Restrictions on the underlying context-free grammar result in a *Dyck* language, i.e., one that generates balanced-parentheses expressions. This restrictive approach still suffices for certain simple pointer analysis algorithms and at the same time it enables very aggressive performance optimizations [168, 169].

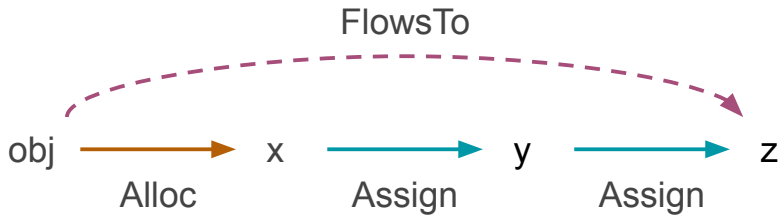


Figure 8.1: Example of a CFL Reachability Graph.

Figure 8.2 gives a simple example of a graph corresponding to a Dyck grammar. An opening parenthesis encodes a field store, and a closing one encodes a field load. In both cases, the field being accessed is used as subscript to differentiate different fields.

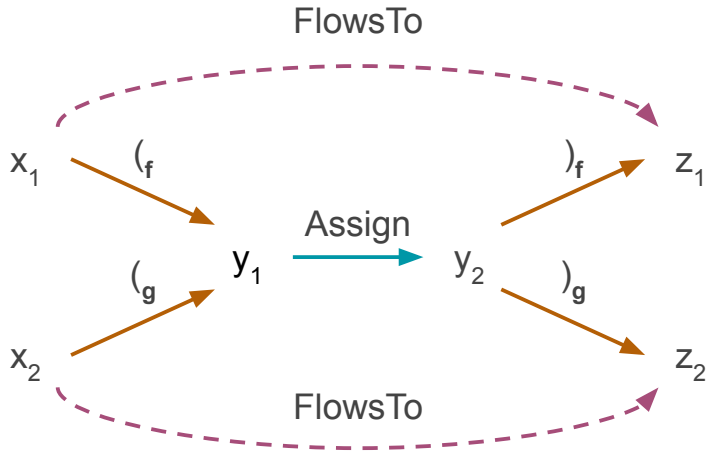


Figure 8.2: Example of a Dyck-CFL Reachability Graph.

The field differentiation of parentheses avoids erroneous inferences, such as the value of x_2 flowing into z_1. The corresponding Dyck grammar includes productions rules as the following:

$$\mathit{FlowsTo} \rightarrow \mathit{Assign} \mid (_f\ \mathit{FlowsTo}\ )_f \mid (_g\ \mathit{FlowsTo}\ )_g \mid \ldots$$

8.5.3 Probabilistic Pointer Analysis

As previously mentioned, pointer analysis has evolved into a critical tool for compiler analysis and optimizations. Many sophisticated optimizations enforced by a compiler need some kind of proof that certain properties hold in order to guarantee that the compilation process does not introduce errors into the program.

To that end, one approach is to either use some variation of a *must* analysis (as in Chapters 5 and 6), or a *sound-may* analysis (as in Chapter 7) and look at the complement of the result

(i.e, if a sound-may analysis claims a set of behaviors may apply to a certain point in the program, then the compiler is certain that no other behavior may arise).

On the other hand, compilers in recent years have another potential direction to follow; that of *speculative optimizations*. A speculative optimization typically involves a code transformation that allows ambiguous memory references to be scheduled in a potentially unsafe order, and requires a recovery mechanism to ensure program correctness. Nowadays, speculative optimizations are widely used in the compilation pipeline, which allows compilers to employ the results of a *may* analysis as well. The very nature of those optimizations means that they are accompanied with safeguards for when applying them was not the correct choice. Thus, a compiler can aggressively exploit information from an analysis that is not always guaranteed to hold.

More aggressive, hardware-supported techniques, such as thread-level speculations [54, 76, 130] and transactional programming [53, 54] apply speculative parallelization of sequential programs by utilizing specialized hardware. The downside is that many of these optimizations rely on extensive data dependence profiling in order to decide when to speculate. Such information is expensive to acquire or potentially unavailable.

This has led to the family of *probabilistic pointer analysis* algorithms [21, 30, 116] that aim to compute pointer information accompanied with some amount of likelihood. These algorithms provide an alternative approach to profiling. Since they are an instance of a static analysis, they can be employed at compile-time to alleviate the aforementioned lack of profile information.

A probabilistic pointer analysis algorithms statically predict the probability of each points-to relation at each program point. Such probabilities are especially useful to a compiler when applying some kind of speculative optimization. Beside more traditional techniques (for instance, static or dynamic profiling), various statistics concepts and stochastic models (e.g., sparse transformation matrices, Discrete-Time Markov chains, etc.) from other fields have been imported into the domain of static analysis for this purpose.

8.5.4 Recency Abstraction

Any static analysis algorithm will construct an abstraction model of the program's memory, with a single *abstract* heap object potentially encoding multiple *concrete* (runtime) objects. The most common approach is (as already discussed in previous chapters) to use each allocation instruction to encode a single abstract object in memory.

A different approach is offered by Balakrishnan and Reps [7] in their *recency-abstraction* technique. In that approach, each allocation site encodes *two* abstract memory objects: (1) one that represents the *most-recently-allocated* object (for that allocation site), and (2) one that summarizes all other, previously allocated objects. An analysis can exploit this most-recently-allocated object, since it represents a single concrete runtime object, and apply "strong updates" reasoning. This proves essential in improving precision and scalability on a flow-sensitive analysis.

8.5.5 Separation Logic (& Hoare Logic)

Pointer analysis ultimately constructs a model of the heap, by computing all the heap objects that each program expression may point to during execution. But, this is by no means the only approach that analyzes the heap. Other approaches, that stem from the field of *separation logic* [18, 66, 111, 112, 127, 128], have been used to reason about the heap and produce proofs regarding pointer safety. Separation logic, in turn, extends the theory of *Hoare logic* [5].

Hoare logic provides a formal system to reason about program correctness, by encoding a language's semantics (and therefore the program's as well) in *Hoare triples*. A Hoare triple has the form $\{P\}\ C\ \{Q\}$, and encodes that whenever an assertion P holds, before executing command C, then assertion Q is quaranteed to hold afterwards—if C terminates. The P and Q assertions express conditions on local variables—written in standard mathematical notation alongside some form of calculus (e.g., *first-order logic*).

Hoare logic provides two variants of operation: (1) a forward approach in which one starts from a precondition and generates formulas in order to prove a postcondition, (2) and a backwards approach in which the opposite direction is followed (i.e., start from a postcondition and prove a precondition). In the general case, regardless of which variant is employed, the process cannot result in a fully automated reasoning and building a general proof may require human guidance to some extent.

Separation logic builds upon Hoare logic by introducing additional operators in the syntax of assertions that focus on local reasoning. For instance, the *separating conjunction* operator, $P * Q$, asserts that conditions P and Q hold for separate parts of memory, and thus can be used on program proofs to enable modular reasoning. Another interesting operator is that of *separating implication*, $P \mathrel{-\!\!*} Q$, which asserts that if the current heap is extended with a part where P holds, then Q holds in the extended heap. It is noteworthy that, despite all of its extensions, separation logic is not more "powerful" than Hoare logic—all that is provable in separation logic is also provable in Hoare logic. The extensions serve to simplify the specifications and proofs.

For example, the condition $(x \mapsto y\ *\ y \mapsto x)$ asserts that x points to y and separately y points to x. This formula describes *precisely* two allocated memory parts—denoted by x and y—i.e., it is guaranteed that the pointers do not alias.

A simple example of a separation logic rule, given some resource r, is the following:

$$\{\ isOpen(r)\ \}\ \texttt{closeRes(r)}\ \{\ isClosed(r)\ \}$$

This rule describes a *specification* for a method `closeRes` that given a resource handler r, closes that resource. Now, given a precondition $\{isOpen(r_1)\ *\ isOpen(r_2)\}$ one can infer the following formula:

$$\{\ isOpen(r_1)\ *\ isOpen(r_2)\ \}\ \texttt{closeRes(r}_1\texttt{)}\ \{\ isClosed(r_1)\ *\ isOpen(r_2)\ \}$$

This highlights how separating conjunction allows one to reason about mutations in memory, mimicking the actual updates happening in RAM during execution. Such reasoning leads to logical proofs about imperative programs that match computational intuition.

The previous formula expansion is based on a general pattern in separation logic; a *frame rule* that allows one to go from smaller to bigger specifications. It is named after the classic frame problem found in artificial intelligence. Here $frame$ describes a part of the program state that remains unchanged after the semantics of command C have been applied.

$$\frac{\{P\}\ C\ \{Q\}}{\{P\ *\ frame\}\ C\ \{Q\ *\ frame\}}$$

The frame rule is key to local reasoning in separation logic; reasoning and specifications should concentrate on the resources that are affected by a given command, without mentioning what remains unchanged.

Bi-abduction. In classical logic, entailment statements, such as $A\ \vdash\ G$, denote that A implies G. Subsequently, the notion of *abduction* extends such statements in order to infer some "missing" assumption $?M$, given an assumption A and a goal G:

$$A \wedge ?M\ \vdash\ G$$

This is similarly expressed in separation logic via the separating conjunction operator, which also partitions the premises:

$$A * ?M\ \vdash\ G$$

Finally, this leads to the more general problem of *bi-abduction*, in which a theorem prover tries to infer "missing" information in both parts of the entailment statement:

$$A * ?Antiframe\ \vdash\ G * ?Frame$$

The notion of bi-abduction has allowed analyses of large programs to circumvent the fact that, normally, reasoning is an untractable problem. Bi-abduction enables one to break a large analysis of a whole program in small *independent* analyses of its parts (e.g., methods). This allows a theorem prover to scale independently of the size of the analyzed code. This approach has the added benefit of making the analysis incremental; if some code changes in the future, the analysis doesn't need to re-analyze the unchanged part of the code, but can instead reuse what was previously inferred.

As an illustrating example, assume that a method has the following generic specification:

$$\{P\}\ \texttt{meth()}\ \{Q\}$$

Additionally, assume that $CallingState$ represents what was computed to hold before the method invocation. In order to utilize the method specification, the following implication has to hold as well:

$$CallingState\ \vdash P$$

Bi-abduction is used at method call sites for two reasons: to discover missing state that is needed for the above implication to hold (the antiframe), as well as state that the call leaves

unchanged (the frame). For instance, assume that the following two calling statements are under examination:

$$\texttt{closeRes(r}_1\texttt{)} \ ; \ \texttt{closeRes(r}_2\texttt{)}$$

Considering the first call, one could guess (or have prior domain-specific knowledge) that a precondition including $\{ \ isOpen(r_1) \ \}$ could be a reasonable starting point. Thus, bi-abduction has to answer the following query (ϵ represents the empty state, i.e., presume nothing):

$$\epsilon * ?Antiframe \ \vdash \ isOpen(r_1) * ?Frame$$

This is trivially solved by picking $(?Antiframe = isOpen(r_1))$ and $(?Frame = \epsilon)$. After applying common logical rules, the formula is converted to the following trivial implication:

$$isOpen(r_1) \ \vdash \ isOpen(r_1)$$

The formula satisfies the requirement to correctly make the method call, which leads to:

$$\{ \ isOpen(r_1) \ \} \ \texttt{closeRes(r}_1\texttt{)} \ \{ \ isClosed(r_1) \ \} \ \texttt{closeRes(r}_2\texttt{)}$$

The condition $isClosed(r_1)$ doesn't have enough information to satisfy the second call to `closeRes`, so following similar reasoning as before, the next bi-abduction query is the following:

$$isClosed(r_1) * ?Antiframe \ \vdash \ isOpen(r_2) * ?Frame$$

It is simple to find a solution satisfying the previous query by picking $(?Antiframe = isOpen(r_2))$ and $(?Frame = isClosed(r_1))$. This leads to the updated formula:

$$\{ \ isOpen(r_1) \ * \ isOpen(r_2) \ \} \ \texttt{closeRes(r}_1\texttt{)} \ \{ \ isClosed(r_1) \ * \ isOpen(r_2) \ \} \ \texttt{closeRes(r}_2\texttt{)}$$

which leads to the final answer, by updating the postcondition of the second call as well:

$$\{ \ isOpen(r_1) \ * \ isOpen(r_2) \ \} \ \texttt{closeRes(r}_1\texttt{)} \ \{ \ isClosed(r_1) \ * \ isOpen(r_2) \ \}$$

$$\texttt{closeRes(r}_2\texttt{)} \ \{ \ isClosed(r_1) \ * \ isClosed(r_2) \ \}$$

Monoidics & Facebook Infer. It is noteworthy to mention here a quite successful tool related to separation logic. *Infer*, or also known as *Facebook Infer* [20], is as static analysis tool initially developed by Monoidics in 2009 (by Calcagno, Distefano and O'Hearn), and later aquired by Facebook in 2013. In 2015 the code was open-sourced and has since been widely used by various groups.

Infer has its roots in the theory of separation logic and builds upon previous successful tools in academic work on automatic program verification (e.g., Smallfoot and SpaceInvader). Written in OCaml, it offers support for multiple programming languages (Java, C, C++, and Objective-C). It is deployed in Facebook to analyze its Android and iOS applications. Facebook claims that Infer has helped developers discover hundreds of bugs per month.

8.5.6 Program Verification

The ability to formally prove programs correct is a desirable element of any programming language, since this leads to more reliable programs [14, 42]. The downside is that, in many cases proving a program correct is a tedious and impractical endeavor, but nevertheless, it can be quite valuable in reasoning about the semantics of a given program. Throughout the years, many theories and tools have been introduced in order to tackle this issue.

Hoare logic, as well as separation logic, are such formal system for reasoning about a program's correctness. Alongside a set of axioms or rules (such as the frame rule), one provides semantics for every program element (e.g., assignments, function calls, etc.) and then proceeds to prove the desired properties. Other interesting formal systems include *Propositional Calculus*, *First-Order Logic* and solvers for the *Boolean Satisfiability Problem* (SAT Solvers).

Proposed systems and tools fall in one of two big subcategories. Either they aim to provide fully automated theorem proving (also known as *ATP* or *automatic deduction*), or they allow for interaction with a human during the process of formulating a formal proof (these systems are referred to as *proof assistants*). Since, proofs generated by automated theorem provers are typically large, the problem of compressing them is crucial and various techniques have been proposed in order to make a prover's output smaller and consequently more easily understandable and verifiable.

SAT Solvers (& SMT Solvers). The Boolean satisfiability problem (or abbreviated as SAT) revolves around determining if there exists an interpretation that satisfies a given Boolean formula (i.e., specific Boolean values to each variable in the formula, such that the formula is evaluate to `true`). For instance, the formula $(a \wedge \neg b)$ is satisfiable (with the solution being $a =$ `true`, $b =$ `false`), whereas the formula $(a \wedge \neg a)$ is not.

SAT was the first problem to be proven to be *NP-complete*, and thus, there is no known algorithm that efficiently solves any SAT problem instance. Nevertheless, as of 2007, heuristic SAT-algorithms are able to solve problem instances involving tens of thousands of variables and formulas consisting of millions of symbols, which is sufficient for many practical SAT problems from, e.g., artificial intelligence, circuit design, and automatic theorem proving.

Specifically, in the domain of automatic theorem proving, it has been shown that problems can often be reduced to Boolean satisfiability formulas, and hence a SAT solver can be applied to search for a solution. Recent advances have made this approach quite feasible in practice [48, 59, 161].

The general approach to modeling a problem for a SAT solver is as follows: (1) define a finite set of possibilities, called *states*, (2) model states using propositional variables, (3) use propositional formulas to describe legan and illegal states, and (4) construct a propositional formula describing the desired state. The process of SAT solving takes place, and if the formula is satisfiable, then the satisfying assignment also gives the desired state, or if the formula is unsatisfiable, then the desired state does not exist.

Finally, a closely related notion is that of *satisfiability modulo theories* solvers [85, 126] (or

abbreviated as SMT solvers). Satisfiability modulo theories generalizes Boolean satisfiability (SAT) by adding a combinations of background theories such as: equality reasoning, arithmetic, fixed-size bit-vectors, arrays, uninterpreted functions, and other useful first-order theories. SMT solvers are not any more powerful than SAT solvers. They will still run in exponential time or be incomplete for the same problems in SAT. The advantage of SMT is that many things that are obvious in an SMT solver can take a long time for an equivalent SAT solver to rediscover. Any problem that is provable by a SAT solver is also provable by an SMT solver. A highly successful instance of an SMT solver is Z3 [32], which is developed by Microsoft Research.

Coq Proof Assistant. Another quite notable mention to a successful tool has to include the *Coq* proof assistant [115] (named after its principal developer, Thierry Coquand). Coq is an interactive proof assistant that was initially released in 1989. It provides a formal language (called Gallina) to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. It enables one to express mathematical assertions, mechanically check proofs of those assertions, assists in finding formal proofs, and finally, extract a certified program from the constructive proof process of its formal specification.

Coq is based on the theory of the calculus of *inductive constructions* (a λ-calculus with a rich type system), a derivative of the calculus of constructions. It is not an automated theorem prover, but includes automatic theorem proving procedures.

Typical applications include the certification of properties of programs (e.g., the CompCert compiler certification project, the Verified Software Toolchain for verification of C programs, or the Iris framework for concurrent separation logic), the formalization of mathematics (e.g., the full formalization of the Feit-Thompson theorem, or the Four color theorem), and teaching.

8.5.7 Program Synthesis

Program synthesis is the task of automatically constructing a program in the underlying programming language that satisfies the user intent expressed in the form of some high-level specification. It differs from program verification, in that the program is to be constructed rather than already existing. However, both fields make use of formal proof techniques, and both show varied degree of automation. Specification in program synthesis are usually expressed in a logical calculus.

In 1957, Alonzo Church, during the Summer Institute of Symbolic Logic at Cornell University tried to synthesize circuits from mathematical requirements. Eventually, researchers of Artificial Intelligence in the 1960s elaborated on the concept of program synthesis to apply it to symbolic AI research.

Since its inception, program synthesis has been considered the holy grail of Computer Science. Pnueli considered it to be one of the most central problems in the theory of programming [117]. Despite inherent challenges in the problem such as ambiguity of user intent and

a typically enormous search space of programs, the field of program synthesis has developed many different techniques that enable program synthesis in different real-life application domains.

After the development of the first automated theorem provers, there was a lot of pioneering work on deductive synthesis approaches [49, 102, 160]. The principle was to use a theorem prover to first construct a proof of a user-provided specification, and then use the proof to extract the corresponding logical program. A different popular direction was that of transformation-based synthesis [101], in which a high-level complete specification was transformed repeatedly until the desired low-level program is acquired.

A common problem with approaches that assumed a complete formal specification turned out to be that providing such a specification could be as complicated as writing the program itself. This leads to new techniques based on inductive specifications such as *input-output* examples, *demonstrations*, genetic programming, and more [75, 139, 146, 154]. The more recent approaches allow a user to additionally provide a skeleton (grammar) of the space of possible programs, in addition to a specification [3].

Program synthesis is now successfully applied in software engineering, biological discovery, computer-aided education, end-user programming, and data cleaning. In the last decade, several applications of synthesis in the field of programming by examples have been deployed in mass-market industrial products. Popular synthesis frameworks include the SKETCH system [147], the PROSE framework for FlashFill-like programming by examples [51], and the ROSETTE virtual machine for solver-aided programming [156]. Potentially, one of the biggest applications where programming synthesis is used nowadays is making computer programming more accessible. Applications such as AutoProf, FlashFill, and Storyboard Programming Tool allow students to write programs in more intuitive ways by manipulating certain concepts directly without having to touch code.

8.5.8 Program Slicing

Program slicing [64, 96, 155, 162] is a technique for simplifying programs by focusing on selected aspects of semantics. It refers to the computation of the set of program statements—the program *slice*—that may affect the values at some point of interest—the *slicing criterion* (e.g., the value of variable `x` at program point n). The slice is constructed by deleting the parts of the program that are irrelevant to those values.

Program slicing can be used in program debugging to locate the source of errors more easily, since it produces a minimal example of code that exhibits the same erroneous behavior as the original program. Other applications include software maintenance, optimization, and program analysis.

When discussing program slicing there are two dimensions of interest: a semantics one and a syntactic one. The dimension of semantics describes what is to be preserved, and three main paradigms are observed: (1) *static* slicing which preserves a program's static behavior, (2) *dynamic* slicing which preserves a program's dynamic behavior, and finally, (3) *conditioned slicing* which attempts to bridge the gap between the previous two.

The dimension of syntax presents only two alternatives: either (1) *syntax-observing* slicing (the norm in existing work) which preserves a program's original syntax, merely removing irrelevant parts, or (2) *amorphous* slicing which is free to perform any syntactic transformation as long as the semantics constraints are preserved.

Finally, given a slicing criterion, there are two possible forms of slice that can be produced: either a *backward* one or a *forward* one. A backward slice contains the program's statements that can have some effect on the slicing criterion, whereas a forward one contains those statements that are affected by the slicing criterion.

Slices constructed in static slicing tend to be rather large. This is particularly true for well-constructed programs, where the computation of the value of each variable is highly dependent upon the values of many other variables. Dynamic slicing exploits the fact that some useful information might have been observed during some program execution. For instance, it is reasonable to construct a slice that is based on certain values that the input had when an error was observed. Dynamic slices are quite attractive, far simpler than the ones produced by static slicing, and can more easily highlight bugs in the original program. Nevertheless, static slices are still relevant in applications where the slice has to be sound for every potential execution.

Therefore, it is apparent that static and dynamic slicing represent two extremes—either the input is not taken into account (static approach) or the input is crucial on the process of deciding the relevant program statements (dynamic approach). Conditioned slicing lies in between; it takes into account information regarding the input without being so specific as to have the precise values. For instance, a potential condition could be "variable `x` is greater than `y`, and `z` is 42". This is in contrast to a static approach that would entirely disregard specific input values, or to a dynamic approach that would require precise values for all variables of interest.

8.5.9 Dynamic Symbolic Execution

The notion of *symbolic execution* is quite old in computer science [31]. It provides a methodology of analyzing a program to determine which inputs cause the execution of various program parts. An interpreter follows the control flow of a program, assuming *symbolic* values for inputs rather than concrete, actual ones as a normal program execution would.

For instance, at an instruction that reads some input and assigns it to a variable `x`, symbolic execution would assign some symbolic value such as λ. If later, variable `y` is assigned the result of (`x * 2`), it would consequently have the symbolic value (λ `* 2`), and so on and so forth. When dealing with branching and conditions, symbolic execution would proceed along both branches, assuming each time the appropriate symbolic values. In our running example, if a branch instruction had the condition (`y == 10`), symbolic execution would follow both paths independently, each time accumulating the corresponding constraint—i.e., (λ `* 2 == 10`) or (λ `* 2 != 10`), respectively. Subsequently, a constraint solver could deduce that (λ `== 5`), in the case that the "if" branch had been followed.

There are various limitations that burden symbolic execution [36, 77, 97, 152]. These include,

but are not limited to, the following:

- *Path explosion* is due to executing all feasible program paths; a number that grows exponentially with an increase in program size and can even be infinite in cases of unbounded loop iterations.
- The handling of *memory aliasing*, since such a property cannot always be accurately computed in a static manner, and thus, symbolic execution cannot recognize that a change to the value of one variable might affect the values of other variables as well.
- The treatment of *arrays*, since references such as `A[i]` can only be specified dynamically, when the value for `i` has a concrete value. A compromise is to treat the entire array as a single value (a technique also known as *array smashing* [13]).
- *Environment interactions*, given that programs interact with their environment in various ways (e.g., performing system calls, reading from a file or database, dealing with threads) and modeling such interactions is challenging.

Dynamic symbolic execution [17, 47, 134, 135, 165] (also known as *concolic execution*—a portmanteau of concrete and symbolic) is a powerful automated testing technique that is based on a hybrid approach; it keeps track of the program state both concretely, as a dynamic analysis would, and symbolically, as a static analysis would. In layman's terms, it boils down to running a symbolic execution alongside a concrete one. This way, the symbolic execution can benefit from the concrete path to avoid the explosion on the number of possible paths. Only one path is considered at a given time, but the constraints on input are accumulated along the path and then inverted to produce new inputs, aiming to reach new code parts. The goal of dynamic symbolic execution is to systematically generate non-equivalent inputs (i.e., inputs that lead the program's execution along different paths).

Essentially, a dynamic symbolic execution algorithm has the following steps:

1. Choose a particular set of variables to be input variables. These will be treated as symbolic variables, whereas all others will be treated as having concrete values.
2. Log any operation that may affect a symbolic variable's value or a path condition.
3. Generate an arbitrary input to jump-start the process.
4. Concretely execute the program and generate a set of symbolic constraints and path conditions.
5. Negate the last path condition that is not already negated, in order to visit a new execution path.
6. Invoke an automated satisfiability solver (SAT or SMT) on the new set of path conditions to generate a new input.
7. Re-execute the program (step 4) and repeat the process.

The steps above reveal a few complications in the whole process. Namely, first that the algorithm performs a depth-first search over the implicit tree of possible execution paths. In practice this path tree will be very large, or maybe even infinite. To prevent spending too much time on one small part of the program, the search may be limited by depth. Second,

both symbolic execution and automated theorem provers have limitations on the classes of constraints they can represent and solve. Any time a constraint that is outside the reach of the given solver is reached, the symbolic execution may substitute the current concrete value of one of the variable in order to simplify the problem.

The area of dynamic symbolic execution has led to the development of many successful tools, such as SAGE and Pex by Microsoft, the KLEE [134, 135] and S2E (open-source) tools which are widely used in many companies including NVIDIA and IBM, Cloud9 by EPFL, Symbolic PathFinder (SPF) by NASA [114], and more.

9. CONCLUSIONS AND FUTURE WORK

> Everything's got to end sometime. Otherwise nothing would ever get started.
>
> *The 11th Doctor* - Doctor Who

In this final chapter, we assess our initial thesis and conclude, while also considering interesting directions for future work.

The first part of our dissertation thesis states that it is possible to obtain *precise* yet *scalable* static pointer analysis algorithms by carefully employing different policies for different parts of the program.

In Chapter 3, we presented an analysis that combines call-site sensitivity and object sensitivity in a non-trivial manner. Instead of keeping both context flavors at all times, we alter an object-sensitive analysis so that it uses call-site sensitive elements in places where it would be more beneficial (e.g., in handling static call invocations). We show that this approach not only bears the precision benefits of combining the two flavors, but also avoids incurring the accumulated cost. For instance, in our experiments we observed an average speedup of **1.53x** alongside a more precise analysis. Additionally, we provide a concise way to formulate variations of both a uniform and a selective approach, in order to experiment with different flavor combinations and context depths.

In Chapter 4, *introspective analysis* examines another approach to a precise and scalable analysis. Our two-phase algorithm allows for a cheap, yet imprecise, analysis to run as a first step in order to gather crucial metrics that gauge the potential effect that a more precise context might have on various program elements (e.g., object allocations or method invocations). Subsequently, a more precise analysis is applied only on those elements that were deemed to benefit from the additional precision, without at the same time imposing a significant penalty on performance.

We employ various heuristics for deciding which program elements are worth the extra effort of a more precise handling, and show experimentally that although it is not a "first line of defense" kind of analysis, introspective analysis can be a valuable "if all else fails" alternative. Users can "dial-in" scalability (by parametrizing the heuristics that decide which elements are to be handled more accurately) to the exact level required, without having to sacrifice a significant fraction of precision. Previously hopeless analyses now become feasible. For instance, a variation of our analysis scales to all but one benchmark in under **20 minutes**, while keeping about **2/3** of the precision gains that a more precise, yet "heavy" analysis would achieve.

The second part of our thesis stipulates that analyses can be designed to offer novel, strong guarantees on the soundness of results, but only for specific parts of the program.

In Chapters 5 and 6, we model an instance of a *must*-alias analysis, a conservative analysis that *under*-approximates results but can guarantee that what is reported is actually correct.

The nature of the analysis makes it valuable for compiler optimizations, program understanding, the improvement of bug detectors, and even as an internal component in more sophisticated analyses.

The analysis we present is minimal, yet it models core features in a handful of declarative rules. This makes reasoning about the analysis semantics less arduous. For instance, this partially led to the insights that called for the introduction of a specialized data structure in Chapter 6. Additionally, the analysis highlights a non-conventional use of context; instead of a beneficial add-on, it is a crucial part of the analysis that guarantees that interprocedural propagation of information remains valid. Another benefit stemming from our analysis is its *incrementality*. Soundness is not compromised if only a portion of the program-under-analysis or its libraries are available; only completeness. The availability of more code simply implies more inferences for our analysis. Precomputed facts are guaranteed to always hold, independently of what new parts of the code are analyzed in the future.

Following our insights from Chapter 5, we introduce a specialized data structure that exploits the fact that must-alias sets are equivalence classes, and as such there is no need to explicitly compute each alias pair. This "laziness" in computation is further exploited to implicitly encode the extension of alias information to longer access paths. Our data structure is in the form of a alias graph that abstractly represents local variables and the heap. Nodes (abstract objects) are alias classes, edges are field-points-to relationships.

In a complementary fashion, we describe all the algorithms on our alias graph required by a must-alias analysis. We implemented our data structure both imperatively, in Java, with destructive updates, and purely functionally, in Datalog. Both implementations yield large performance improvements compared to an explicit representation of all alias pairs. The imperative version achieves a speedup of up to **two orders** of magnitude, with the declarative implementation nearly matching it in most cases. As a result, the running time of a realistic must-alias analysis becomes small—**a few tens of seconds** for large benchmarks and the full Java library.

Finally, in Chapter 7, we conclude our contributions with another conservative and *fully sound* may-analysis. Soundness in our *defensive analysis* means that it has to (correctly) over-approximate all concrete executions. This proves quite challenging in practice due to code that cannot be analyzed (e.g., dynamically generated code, or native code) or dynamic language features (e.g., reflection).

The analysis employs a different logical approach from past analyses, in order to successfully distinguish "safe" inferences (i.e., certain to not be affected by unknown code, and hence correct). In essence, our analysis produces inferences only when these are guaranteed to hold because of existing code, and cannot possibly be violated by other, unknown code. In our effort to implement defensive analysis in a realistic package, we found that *laziness* is an essential feature—the analysis cannot scale without it for real-world programs.

Experiments show that the analysis is efficient, leveraging its lazy representation of points-to sets. As a result, it can be made precise, beyond the limits of standard whole-program points-to analyses (e.g., achieving a 5-call-site-sensitive and flow-sensitive analysis). The analysis is also modular since it can be applied to any subset of the program. Although

quite defensive, the analysis yields useful coverage over large Java benchmarks. In our experimental setup, the analysis computes guaranteed over-approximate points-to set for **34-74%** of the local variables of a conventional unsound analysis. Similar effectiveness is achieved for other metrics, again with actionable, guaranteed-sound outcomes.

To summarize, we advocate that modern, sophisticated, static pointer analyses need not make a sacrifice over precision or scalability, to achieve the other. Both properties are achievable with appropriate tuning and design choices, for different parts of the program. Complementary, it is possible for analyses to compute results alongside with strong soundness guarantees, again focusing at specific parts of the program. To conclude, a static pointer analysis algorithm doesn't have to use a one-size-fits-all handling of every language feature and program point, but instead it is favorable to methodically differentiate its policies for different parts of the code, towards different desired outcomes.

9.1 Future Work

Finally, we will discuss some interesting future directions to tackle existing limitations of our approaches.

Hybrid-Context Sensitivity. Our approach in hybrid analysis showed that it is beneficial to combine different kind of context flavors when dealing with different program elements. Our work focused on the handling of static call invocations, and also only examined two alternatives of contexts (in a given analysis). Future work can explore the existence of other language features that would benefit from a different context, as well as the potential variation of context depth itself. An example of an interesting program element that might benefit from a different context is the handling of collections (e.g., lists, maps, sets, etc.) that the Java library offers. Maybe an analysis can keep its context depth limited in "normal" code, and push for higher precision (via allowing for more context depth) when analyzing the code of a collection class.

Introspective Analysis. Future work in the context of our introspective analysis, first includes the examination of more sophisticated and highly tuned heuristics. Our heuristics, were good enough to gauge potential program elements that would benefit from a more precise handling, but there is always room for improvement. Especially, in picking more fined-tuned values for the parameters of each heuristic, one might examine an approach in which the actual values depend on other metrics specific to the program at hand. For instance, whether the program is heavy on reflection or the use of static call, etc. Another potential direction to explore is whether a more sophisticated strategy in the refinement steps is favorable. In our current implementation, the first step is a cheap and crude analysis, and the second step is the fully-fledged precise analysis. Maybe a multi-staged approach that slowly increases precision, while in interleaved steps re-evaluates which program elements need more accurate handling, bears significant precision and scalability gains.

Must-Alias Analysis. The main venue of exploration in our must-alias analysis is that of expanding our minimal model. The presented model captures the core elements that an analysis of this nature has to handle, but still misses many more language features. Improvements towards that end will server in inferring more alias pairs, thus increasing the analysis coverage. An additional direction, is that of modeling specific library code that is too hard to manually analyze but could potentially be crucial in improving the analysis reasoning. For example, such modeling might include important "low-level" methods such as `equals` or `clone` that have clear and "safe" semantics.

Defensive Points-To Analysis. Finally, regarding our last contribution, future work includes directions similar to those of our must-alias analysis. A potential modeling of crucial methods, such as access to collections and core native functions could significantly increase the coverage of the analysis. Those methods are hard to automatically analyze in a sound manner, but have clear semantics offered by the language specification. Furthermore, another future approach includes the modeling of a more refined concurrency model. The simplified concurrency model we presented allowed us to describe the analysis in its purest form, starting from a sound basis, with the potential of adding to it conservatively.

REFERENCES

[1] Karim Ali and Ondrej Lhoták. "Application-Only Call Graph Construction". In: *European Conference on Object-Oriented Programming (ECOOP)*. ECOOP '12. Beijing, China: Springer, 2012, pp. 688–712. ISBN: 978-3-642-31056-0.

[2] Rita Z. Altucher and William Landi. "An Extended Form of Must Alias Analysis for Dynamic Allocation". In: *Principles of Programming Languages (POPL)*. POPL '95. San Francisco, California, USA: ACM, 1995, pp. 74–84. ISBN: 0-89791-692-1.

[3] R. Alur et al. "Syntax-guided synthesis". In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 1–8.

[4] Lars O. Andersen. "Program Analysis and Specialization for the C Programming Language". PhD thesis. DIKU, University of Copenhagen, 1994.

[5] Krzysztof R. Apt. "Ten Years of Hoare's Logic: A Survey-Part I". In: *ACM Trans. Programming Languages and Systems* 3 (1981), pp. 431–483. DOI: 10.1145/357146.357150.

[6] David F. Bacon and Peter F. Sweeney. "Fast static analysis of C++ virtual function calls". In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. OOPSLA '96. San Jose, California, USA: ACM, 1996, pp. 324–341. ISBN: 0-89791-788-X.

[7] Gogul Balakrishnan and Thomas W. Reps. "Recency-Abstraction for Heap-Allocated Storage". In: *International Symposium on Static Analysis (SAS)*. SAS '06. Seoul, Korea: Springer, 2006, pp. 221–239.

[8] George Balatsouras et al. "A Datalog Model of Must-Alias Analysis". In: *International Workshop on State Of the Art in Program Analysis (SOAP)*. SOAP '17. Barcelona, Spain: ACM, 2017, pp. 7–12. ISBN: 978-1-4503-5072-3.

[9] Marc Berndl et al. "Points-to analysis using BDDs". In: *Programming Language Design and Implementation (PLDI)*. PLDI '03. San Diego, California, USA: ACM, 2003, pp. 103–114. ISBN: 1-58113-662-5.

[10] Sandip K. Biswas. "A demand-driven set-based analysis". In: *Principles of Programming Languages (POPL)*. POPL '97. Paris, France: ACM, 1997, pp. 372–385. ISBN: 0-89791-853-3.

[11] Stephen M. Blackburn et al. "The DaCapo benchmarks: Java benchmarking development and analysis". In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 169–190. ISBN: 1-59593-348-4.

[12] Bruno Blanchet. "Escape analysis: Correctness proof, implementation and experimental results". In: *Principles of Programming Languages (POPL)*. POPL '98. San Diego, California, USA: ACM, 1998, pp. 25–37. ISBN: 0-89791-979-3.

[13] Bruno Blanchet et al. “Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software”. In: *The Essence of Compuation: Complexity, Analysis, Transformation* (2002).

[14] Ahmed Bouajjani et al. “Programs with Lists Are Counter Automata”. In: *International Conference on Computer Aided Verification (CAV)*. CAV ’06. Seattle, WA, USA: Springer, 2006, pp. 517–531. ISBN: 3-540-37406-X. DOI: 10.1007/11817963.

[15] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. “SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution”. In: *SIGPLAN Notices* 10.6 (1975), pp. 234–245. DOI: 10.1145/390016.808445.

[16] Martin Bravenboer and Yannis Smaragdakis. “Strictly Declarative Specification of Sophisticated Points-to Analyses”. In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. OOPSLA ’09. Orlando, Florida, USA: ACM, 2009. ISBN: 978-1-60558-766-0.

[17] Cristian Cadar et al. “EXE: Automatically Generating Inputs of Death”. In: *ACM Trans. Inf. Syst. Secur.* (2008).

[18] Cristiano Calcagno et al. “Compositional Shape Analysis by Means of Bi-Abduction”. In: *J. ACM* 58.6 (2011). ISSN: 0004-5411. DOI: 10.1145/2049697.2049700.

[19] Cristiano Calcagno et al. “Compositional Shape Analysis by Means of Bi-abduction”. In: *Principles of Programming Languages (POPL)*. POPL ’09. Savannah, GA, USA: ACM, 2009, pp. 289–300. ISBN: 978-1-60558-379-2.

[20] Cristiano Calcagno et al. “Moving Fast with Software Verification”. In: *NASA Formal Methods (NFM)*. NFM ’15. Pasadena, CA, USA: Springer, 2015, pp. 3–11.

[21] Peng-Sheng Chen et al. “Interprocedural probabilistic pointer analysis”. In: *IEEE Transactions on Parallel and Distributed Systems* 15.10 (2004), pp. 893–907.

[22] Jong D. Choi, Michael Burke, and Paul Carini. “Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects”. In: *Principles of Programming Languages (POPL)*. POPL ’93. Charleston, South Carolina, USA: ACM, 1993, pp. 232–245. ISBN: 0-89791-560-7.

[23] Stephen Chong. *Personal communication*. 2013.

[24] Cristina Cifuentes. *Personal communication*. 2013.

[25] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic”. In: *Logics of Programs, Workshop*. Vol. 131. LOP ’81. Yorktown Heights, New York, USA: Springer, 1981, pp. 52–71. ISBN: 3-540-11212-X.

[26] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. In: *ACM Trans. on Programming Languages and Systems* 8.2 (1986), pp. 244–263. DOI: 10.1145/5397.5399.

[27] Patrick Cousot and Radhia Cousot. "Abstract Interpretation and Application to Logic Programs". In: *Logic Programming* 13.2&3 (1992), pp. 103–179. DOI: 10.1016/0743-1066(92)90030-7.

[28] Patrick Cousot and Radhia Cousot. "Abstract Interpretation Frameworks". In: *Logic and Computation* 2.4 (1992), pp. 511–547. DOI: 10.1093/logcom/2.4.511.

[29] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Principles of Programming Languages (POPL)*. POPL '77. Los Angeles, California, USA: ACM, 1977, pp. 238–252.

[30] Jeff Da Silva and J. Gregory Steffan. "A Probabilistic Pointer Analysis for Speculative Optimizations". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ASPLOS XII. San Jose, California, USA: ACM, 2006, pp. 416–425.

[31] Roger B. Dannenberg and George W. Ernst. "Formal program verification using symbolic execution". In: *IEEE Transactions on Software Engineering* (1982), pp. 43–52.

[32] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. TACAS '08. Budapest, Hungary: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78799-0.

[33] Arnab De and Deepak D'Souza. "Scalable Flow-sensitive Pointer Analysis for Java with Strong Updates". In: *European Conference on Object-Oriented Programming (ECOOP)*. ECOOP '12. Beijing, China: Springer, 2012, pp. 665–687. ISBN: 978-3-642-31056-0.

[34] Jeffrey Dean, David Grove, and Craig Chambers. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis". In: *European Conference on Object-Oriented Programming (ECOOP)*. ECOOP '95. Århus, Denmark: Springer, 1995, pp. 77–101. ISBN: 3-540-60160-0.

[35] David Delmas and Jean Souyris. "Astrée: From Research to Industry". In: *International Symposium on Static Analysis (SAS)*. SAS '07. Kongens Lyngby, Denmark: Springer, 2007, pp. 437–451. ISBN: 978-3-540-74060-5.

[36] R. A. DeMilli and A. J. Offutt. "Constraint-Based Automatic Test Data Generation". In: *IEEE Transactions on Software Engineering* (1991), pp. 900–910.

[37] Alain Deutsch. "On the Complexity of Escape Analysis". In: *Principles of Programming Languages (POPL)*. POPL '96. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 358–371. ISBN: 0-89791-769-3.

[38] Michael Eichberg et al. "Defining and Continuous Checking of Structural Program Dependencies". In: *International Conference on Software Engineering (ICSE)*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 391–400. ISBN: 978-1-60558-079-1.

[39] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. "Context-sensitive interprocedural points-to analysis in the presence of function pointers". In: *Programming Language Design and Implementation (PLDI)*. PLDI '94. Orlando, Florida, USA: ACM, 1994, pp. 242–256. ISBN: 0-89791-662-X.

[40] E. Allen Emerson and Edmund M. Clarke. "Characterizing Correctness Properties of Parallel Programs Using Fixpoints". In: *Proc. of the 7th Internatioal Colloquium on Automata, Languages and Programming*. Vol. 85. ICALP '80. Noordwijkerhout, Netherlands: Springer, 1980, pp. 169–181. ISBN: 3-540-10003-2.

[41] Manuel Fähndrich et al. "Partial Online Cycle Elimination in Inclusion Constraint Graphs". In: *Programming Language Design and Implementation (PLDI)*. PLDI '98. Montreal, Quebec, Canada: ACM, 1998, pp. 85–96. ISBN: 0-89791-987-4.

[42] James Fetzer. "Program Verification: The Very Idea". In: *Commun. ACM* 31 (1988), pp. 1048–1063. DOI: 10.1145/48529.48530.

[43] Stephen J. Fink. *T.J. Watson Libraries for Analysis (WALA)*. http://wala.sourceforge.net.

[44] Stephen J. Fink et al. *WALA UserGuide: PointerAnalysis*. http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis. 2013.

[45] Stephen Fink et al. "Effective typestate verification in the presence of aliasing". In: *International Symposium on Software Testing and Analysis (ISSTA)*. ISSTA '06. Portland, Maine, USA: ACM, 2006, pp. 133–144. ISBN: 1-59593-263-1.

[46] Robert W Floyd. "Assigning Meanings to Programs". In: *Proc. of Symp. in Applied Mathematics. Mathematical Aspects of Computer Science*. Vol. 19. Providence, Rhode Island: American Mathematical Society, 1967, pp. 19–32.

[47] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA, 2005, pp. 213–223.

[48] Carla P. Gomes et al. *Satisfiability solvers*. 2007.

[49] Cordell Green. "Application of Theorem Proving to Problem Solving". In: IJCAI '69. Washington, DC, USA: Morgan Kaufmann Publishers Inc., 1969, pp. 219–239.

[50] Salvatore Guarnieri and Benjamin Livshits. "GateKeeper: mostly static enforcement of security and reliability policies for Javascript code". In: *Proc. of the 18th USENIX Security Symposium*. SSYM' 09. Montreal, Canada: USENIX Association, 2009, pp. 151–168. URL: http://dl.acm.org/citation.cfm?id=1855768.1855778.

[51] Sumit Gulwani. "Automatic String Processing in Spreadsheets Using Input-Output Examples". In: *Principles of Programming Languages (POPL)*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 317–330. ISBN: 978-1-4503-0490-0.

[52] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. "CodeQuest: Scalable Source Code Queries with Datalog". In: *European Conference on Object-Oriented Programming (ECOOP)*. ECOOP '06. Nantes, France: Springer, 2006, pp. 2–27. ISBN: 3-540-35726-2.

[53] L. Hammond et al. "Transactional coherence and consistency: simplifying parallel hardware and software". In: *IEEE Micro* 24.6 (2004), pp. 92–103.

[54] Lance Hammond, Mark Willey, and Kunle Olukotun. "Data Speculation Support for a Chip Multiprocessor". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ASPLOS VIII. San Jose, California, USA: ACM, 1998, pp. 58–69.

[55] Ben Hardekopf and Calvin Lin. "Exploiting pointer and location equivalence to optimize pointer analysis". In: *International Symposium on Static Analysis (SAS)*. SAS '07. Kongens Lyngby, Denmark: Springer, 2007, pp. 265–280. ISBN: 978-3-540-74060-5.

[56] Ben Hardekopf and Calvin Lin. "Semi-sparse flow-sensitive pointer analysis". In: *Principles of Programming Languages (POPL)*. POPL '09. Savannah, GA, USA: ACM, 2009, pp. 226–238. ISBN: 978-1-60558-379-2.

[57] Ben Hardekopf and Calvin Lin. "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code". In: *Programming Language Design and Implementation (PLDI)*. PLDI '07. San Diego, California, USA: ACM, 2007, pp. 290–299. ISBN: 978-1-59593-633-2.

[58] Nevin Heintze and Olivier Tardieu. "Demand-Driven Pointer Analysis". In: *Programming Language Design and Implementation (PLDI)*. PLDI '01. Snowbird, Utah, USA: ACM, 2001, pp. 24–34. ISBN: 1-58113-414-2.

[59] Marijn JH Heule and Armin Biere. "Proofs for satisfiability problems". In: *All about Proofs, Proofs for all* 55.1 (2015), pp. 1–22.

[60] Michael Hind et al. "Interprocedural Pointer Alias Analysis". In: *ACM Trans. Program. Lang. Syst.* 21.4 (1999), pp. 848–894. ISSN: 0164-0925. DOI: 10.1145/325478.325519.

[61] Martin Hirzel, Amer Diwan, and Michael Hind. "Pointer Analysis in the Presence of Dynamic Class Loading". In: *European Conference on Object-Oriented Programming (ECOOP)*. ECOOP '04. Oslo, Norway: Springer, 2004, pp. 96–122. ISBN: 3-540-22159-X.

[62] Martin Hirzel et al. "Fast online pointer analysis". In: *ACM Trans. Program. Lang. Syst.* 29.2 (2007). DOI: 10.1145/1216374.1216379.

[63] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[64] Susan Horwitz, Thomas Reps, and David Binkley. "Interprocedural Slicing Using Dependence Graphs". In: 12.1 (1990), pp. 26–60.

[65] William E. Howden. "Symbolic Testing and the DISSECT Symbolic Evaluation System". In: *IEEE Trans. Software Engineering* 3.4 (1977), pp. 266–278. DOI: 10.1109/TSE.1977.231144.

[66] Samin S. Ishtiaq and Peter W. O'Hearn. "BI as an Assertion Language for Mutable Data Structures". In: *Principles of Programming Languages (POPL)*. POPL '01. London, UK: ACM, 2001, pp. 14–26. ISBN: 1-58113-336-7.

[67] Suresh Jagannathan et al. "Single and Loving It: Must-alias Analysis for Higher-order Languages". In: *Principles of Programming Languages (POPL)*. POPL '98. San Diego, California, USA: ACM, 1998, pp. 329–341. ISBN: 0-89791-979-3.

[68] John B. Kam and Jeffrey D. Ullman. “Monotone Data Flow Analysis Frameworks”. In: *Acta Informatica* 7 (1977), pp. 305–317.

[69] George Kastrinis and Yannis Smaragdakis. “Efficient and Effective Handling of Exceptions in Java Points-To Analysis”. In: *International Conference on Compiler Construction (CC)*. CC ’13. Rome, Italy: Springer, 2013. ISBN: 978-3-642-37050-2.

[70] George Kastrinis and Yannis Smaragdakis. “Hybrid Context-Sensitivity for Points-To Analysis”. In: *Programming Language Design and Implementation (PLDI)*. PLDI ’13. Seattle, WA, USA: ACM, 2013. ISBN: 978-1-4503-2014-6.

[71] George Kastrinis et al. “An efficient data structure for must-alias analysis”. In: *International Conference on Compiler Construction (CC)*. CC ’18. Vienna, Austria: ACM, 2018, pp. 48–58.

[72] Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data Flow Analysis - Theory and Practice*. CRC Press, 2009. ISBN: 978-0-8493-2880-0.

[73] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proc. of the 1st ACM Symp. on Principles of Programming Languages*. POPL ’73. Boston, Massachusetts, USA: ACM, 1973, pp. 194–206.

[74] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[75] John R. Koza. “Genetic Programming as a means for Programming Computers by Natural Selection”. In: *Statistics and Computing* (1994). DOI: 10.1007/bf00175355.

[76] V. Krishnan and J. Torrellas. “A chip-multiprocessor architecture with speculative multithreading”. In: *IEEE Transactions on Computers* 48.9 (1999), pp. 866–880.

[77] Volodymyr Kuznetsov et al. “Efficient State Merging in Symbolic Execution”. In: *Proc. of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. Beijing, China, 2012, pp. 193–204.

[78] Monica S. Lam et al. “Context-sensitive program analysis as database queries”. In: *Symposium on Principles of Database Systems (PODS)*. PODS ’05. Baltimore, Maryland, USA: ACM, 2005, pp. 1–12. ISBN: 1-59593-062-0.

[79] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. “Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study”. In: *International Conference on Software Engineering (ICSE)*. ICSE ’17. Buenos Aires, Argentina: IEEE, 2017. ISBN: 978-1-5386-3868-2.

[80] Chris Lattner, Andrew Lenharth, and Vikram Adve. “Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World”. In: *Programming Language Design and Implementation (PLDI)*. PLDI ’07. San Diego, California, USA: ACM, 2007. ISBN: 978-1-59593-633-2.

[81] Ondřej Lhoták. “Program Analysis using Binary Decision Diagrams”. PhD thesis. McGill University, 2006.

[82] Ondřej Lhoták and Kwok-Chiang Andrew Chung. "Points-to Analysis with Efficient Strong Updates". In: *Principles of Programming Languages (POPL)*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 3–16. ISBN: 978-1-4503-0490-0.

[83] Ondrej Lhoták and Laurie J. Hendren. "Context-Sensitive Points-to Analysis: Is It Worth It?" In: *International Conference on Compiler Construction (CC)*. CC '06. Vienna, Austria: Springer, 2006, pp. 47–64. ISBN: 3-540-33050-X.

[84] Ondřej Lhoták and Laurie J. Hendren. "Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation". In: *ACM Trans. Softw. Eng. Methodol.* 18.1 (2008), pp. 1–53. ISSN: 1049-331X. DOI: 10.1145/1391984.1391987.

[85] Yi Li et al. "Symbolic Optimization with SMT Solvers". In: *Principles of Programming Languages (POPL)*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 606–618. ISBN: 9781450325448.

[86] Yue Li, Tian Tan, and Jingling Xue. "Effective Soundness-Guided Reflection Analysis". In: *International Symposium on Static Analysis (SAS)*. SAS '15. Saint-Malo, France: Springer, 2015, pp. 162–180.

[87] Yue Li et al. "Precision-guided context sensitivity for pointer analysis". In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. OOPSLA '18. Boston, Massachusetts, USA: ACM, 2018, 141:1–141:29. ISBN: 978-1-4503-6031-9.

[88] Yue Li et al. "Scalability-first pointer analysis with self-tuning context-sensitivity". In: *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ESEC/FSE '18. Lake Buena Vista, Florida, USA: ACM, 2018, pp. 129–140. ISBN: 978-1-4503-5573-5.

[89] Yue Li et al. "Self-Inferencing Reflection Resolution for Java". In: *European Conference on Object-Oriented Programming (ECOOP)*. ECOOP '14. Uppsala, Sweden: Springer, 2014, pp. 27–53. ISBN: 978-3-662-44201-2.

[90] Donglin Liang, Maikel Pennings, and Mary J. Harrold. "Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs". In: *Program Analysis For Software Tools and Engineering (PASTE)*. PASTE '05. Lisbon, Portugal: ACM, 2005, pp. 6–12. ISBN: 1-59593-239-9.

[91] Percy Liang and Mayur Naik. "Scaling abstraction refinement via pruning". In: *Programming Language Design and Implementation (PLDI)*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 590–601. ISBN: 978-1-4503-0663-8.

[92] Benjamin Livshits. "Improving Software Security with Precise Static and Runtime Analysis". PhD thesis. Stanford University, 2006.

[93] Benjamin Livshits, John Whaley, and Monica S. Lam. "Reflection Analysis for Java". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. APLAS '05. Tsukuba, Japan: Springer, 2005, pp. 139–160.

[94] Benjamin Livshits et al. "In Defense of Soundiness: A Manifesto". In: *Commun. ACM* 58.2 (2015), pp. 44–46. ISSN: 0001-0782. DOI: 10.1145/2644805.

[95] Yi Lu et al. “An Incremental Points-To Analysis with CFL-Reachability”. In: *International Conference on Compiler Construction (CC)*. CC ’13. Rome, Italy: Springer, 2013. ISBN: 978-3-642-37050-2.

[96] A. De Lucia. “Program slicing: methods and applications”. In: *Proc. of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*. 2001, pp. 142–149.

[97] Kin-Keung Ma et al. “Directed Symbolic Execution”. In: *Proc. of the 18th International Conference on Static Analysis*. SAS ’11. Venice, Italy, 2011, pp. 95–111.

[98] Xiaodong Ma, Ji Wang, and Wei Dong. “Computing Must and May Alias to Detect Null Pointer Dereference”. In: *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. ISoLA ’08. Porto Sani, Greece: Springer, 2008, pp. 252–261. ISBN: 978-3-540-88478-1.

[99] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. “A Framework For Efficient Modular Heap Analysis”. In: *Foundations and Trends in Programming Languages* 1.4 (2015), pp. 269–381. ISSN: 2325-1107. DOI: 10.1561/2500000020.

[100] Magnus Madsen, Benjamin Livshits, and Michael Fanning. “Practical static analysis of JavaScript applications in the presence of frameworks and libraries”. In: *Foundations of Software Engineering (SIGSOFT FSE)*. FSE ’13. Saint Petersburg, Russian Federation: ACM, 2013, pp. 499–509. ISBN: 978-1-4503-2237-9.

[101] Zohar Manna and Richard Waldinger. “Knowledge and Reasoning in Program Synthesis”. In: IJCAI ’75. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1975, pp. 288–295.

[102] Zohar Manna and Richard J. Waldinger. “Toward Automatic Program Synthesis”. In: *Commun. ACM* (1971), pp. 151–165.

[103] Matthew Might, Yannis Smaragdakis, and David Van Horn. “Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-oriented Program Analysis”. In: *Programming Language Design and Implementation (PLDI)*. PLDI ’10. Toronto, Ontario, Canada: ACM, 2010, pp. 305–315. ISBN: 978-1-4503-0019-3.

[104] Ana Milanova, Wei Huang, and Yao Dong. “CFL-Reachability and Context-Sensitive Integrity Types”. In: *International Conference on Principles and Practices of Programming on the Java Platform (PPPJ)*. PPPJ ’14. Cracow, Poland: ACM, 2014, pp. 99–109.

[105] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. “Parameterized object sensitivity for points-to analysis for Java”. In: *ACM Trans. Softw. Eng. Methodol.* 14.1 (2005), pp. 1–41. ISSN: 1049-331X. DOI: http://doi.acm.org/10.1145/1044834.1044835.

[106] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. “Parameterized object sensitivity for points-to and side-effect analyses for Java”. In: *International Symposium on Software Testing and Analysis (ISSTA)*. ISSTA ’02. Roma, Italy: ACM, 2002, pp. 1–11. ISBN: 1-58113-562-9.

[107] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN: 1-55860-320-4.

[108] Mayur Naik, Alex Aiken, and John Whaley. "Effective static race detection for Java". In: *Programming Language Design and Implementation (PLDI)*. PLDI '06. Ottawa, Ontario, Canada: ACM, 2006, pp. 308–319. ISBN: 1-59593-320-4.

[109] Rupesh Nasre. "Exploiting the structure of the constraint graph for efficient points-to analysis". In: *International Symposium on Memory Management (ISMM)*. ISMM '12. Beijing, China: ACM, 2012, pp. 121–132. ISBN: 978-1-4503-1350-6.

[110] Durica Nikolić and Fausto Spoto. "Definite Expression Aliasing Analysis for Java Bytecode". In: *International Colloquium on Theoretical Aspects of Computing (ICTAC)*. ICTAC '12. Bangalore, India: Springer, 2012, pp. 74–89. ISBN: 978-3-642-32942-5.

[111] Peter O'Hearn. "A Primer on Separation Logic (and Automatic Program Verification and Analysis)". In: (2012).

[112] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Proc. of the 15th International Workshop on Computer Science Logic*. Vol. 2142. CSL '01. Paris, France: Springer, 2001, pp. 1–19. ISBN: 3-540-42554-3.

[113] Corina S. Pasareanu and Neha Rungta. "Symbolic PathFinder: symbolic execution of Java bytecode". In: *Proc. of the 25th IEEE/ACM International Conf. on Automated Software Engineering*. ASE '10. Antwerp, Belgium: ACM, 2010, pp. 179–180. ISBN: 978-1-4503-0116-9.

[114] Corina Pasareanu and Neha Rungta. "Symbolic PathFinder: Symbolic Execution of Java Bytecode". In: 2010, pp. 179–180.

[115] Christine Paulin-Mohring. "Introduction to the Coq Proof-Assistant for Practical Software Verification". In: (2012). DOI: 10.1007/978-3-642-35746-6_3.

[116] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. "A Systematic Approach to Probabilistic Pointer Analysis". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. APLAS '07. Singapore: Springer, 2007, pp. 335–250.

[117] A. Pnueli and R. Rosner. "On the Synthesis of a Reactive Module". In: *Principles of Programming Languages (POPL)*. POPL '89. Austin, Texas, USA: ACM, 1989, pp. 179–190.

[118] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR". In: *Proc. of the 5th International Symp. on Programming*. Vol. 137. Torino, Italy: Springer, 1982, pp. 337–351. ISBN: 3-540-11494-7.

[119] Thomas Reps. "Program Analysis via Graph Reachability". In: *International Symposium on Logic Programming (ILPS)*. ILPS '97. Port Washington, New York, USA: MIT Press, 1997, pp. 5–19.

[120] Thomas Reps. "Program analysis via graph reachability". In: *Information & Software Technology* 40 (1998), pp. 701–726. DOI: 10.1016/S0950-5849(98)00093-7.

[121] Thomas Reps. "Shape Analysis as a Generalized Path Problem". In: *Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. PEPM '95. Le Jolla, California, USA: ACM, 1995, pp. 1–11.

[122] Thomas W. Reps. "Demand interprocedural program analysis using logic databases". In: *Applications of Logic Databases*. Kluwer Academic Publishers, 1994, pp. 163–196.

[123] Thomas W. Reps. "Solving Demand Versions of Interprocedural Analysis Problems". In: *International Conference on Compiler Construction (CC)*. CC '94. Edinburgh,UK: Springer, 1994, pp. 389–403. ISBN: 3-540-57877-3.

[124] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability". In: *Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, 1995, pp. 49–61. ISBN: 0-89791-692-1.

[125] Thomas Reps et al. "Speeding up Slicing". In: *Foundations of Software Engineering (SIGSOFT FSE)*. SIGSOFT '94. New Orleans, Louisiana, USA: ACM, 1994, pp. 11–20.

[126] Andrew Reynolds, Cesare Tinelli, and Clark Barrett. "Constraint Solving for Finite Model Finding in SMT Solvers". In: *Theory and Practice of Logic Programming* 17.4 (2017), pp. 516–558.

[127] John C. Reynolds. "Intuitionistic Reasoning about Shared Mutable Data Structure". In: *Millennial Perspectives in Computer Science*. Palgrave, 2000, pp. 303–321.

[128] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proc. of the 17th IEEE Symp. on Logic in Computer Science*. LICS '02. Copenhagen, Denmark, 2002, pp. 55–74. ISBN: 0-7695-1483-9.

[129] Xavier Rival. *Comment on "What Is Soundness in Static Analysis" post, in the PL Enthusiast blog*. `http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/#comment-1265`. 2017.

[130] Amir Roth and Gurindar S. Sohi. "Speculative Data-Driven Multithreading". In: *International Symposium on High-Performance Computer Architecture (HPCA)*. HPCA '01. Nuevo Leone, Mexico: IEEE Computer Society, 2001, p. 37.

[131] Atanas Rountev and Satish Chandra. "Off-line variable substitution for scaling points-to analysis". In: *Programming Language Design and Implementation (PLDI)*. PLDI '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 47–56. ISBN: 1-58113-199-2.

[132] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. "Parametric shape analysis via 3-valued logic". In: *ACM Transactions on Programming Languages and Systems* 24.3 (2002), pp. 217–298. ISSN: 0164-0925. DOI: `10.1145/514188.514190`.

[133] Bernhard Scholz et al. "On fast large-scale program analysis in Datalog". In: *International Conference on Compiler Construction (CC)*. CC '16. Barcelona, Spain: ACM, 2016, pp. 196–206. ISBN: 978-1-4503-4241-4.

[134] Koushik Sen and Gul Agha. "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools". In: 2006, pp. 419–423.

[135] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Enginge for C". In: *Proc. of the 10th European Software Engineering Conference Held Jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE '13. Lisbon, Portugal, 2005, pp. 263–272.

[136] Lei Shang, Xinwei Xie, and Jingling Xue. "On-Demand Dynamic Summary-Based Points-to Analysis". In: *International Symposium on Code Generation and Optimization (CGO)*. CGO '12. New York, NY, USA: ACM, 2012, pp. 264–274.

[137] Micha Sharir and Amir Pnueli. "Two Approaches to Interprocedural Data Flow Analysis". In: *Program flow analysis: theory and applications*. Prentice-Hall, Inc., 1981. Chap. 7, pp. 189–233.

[138] Micha Sharir and Amir Pnueli. "Two approaches to interprocedural data flow analysis". In: *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981. Chap. 7, pp. 189–234. ISBN: 0137296819.

[139] David E. Shaw, William R. Swartout, and C. Cordell Green. "Inferring LISP Programs from Examples". In: IJCAI '75. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1975, pp. 260–267.

[140] Olin Shivers. "Control-Flow Analysis of Higher-Order Languages". PhD thesis. Carnegie Mellon University, 1991.

[141] Yannis Smaragdakis and George Balatsouras. "Pointer Analysis". In: *Foundations and Trends in Programming Languages* 2.1 (2015), pp. 1–69. ISSN: 2325-1107. DOI: 10.1561/2500000014.

[142] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. "Pick Your Contexts Well: Understanding Object-Sensitivity". In: *Principles of Programming Languages (POPL)*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 17–30. ISBN: 978-1-4503-0490-0.

[143] Yannis Smaragdakis and George Kastrinis. "Defensive Points-To Analysis: Effective Soundness via Laziness". In: *European Conference on Object-Oriented Programming (ECOOP)*. ECOOP '18. Amsterdam, The Netherlands: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 23:1–23:28.

[144] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. "Introspective Analysis: Context-sensitivity, Across the Board". In: *Programming Language Design and Implementation (PLDI)*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 485–495. ISBN: 978-1-4503-2784-8.

[145] Yannis Smaragdakis et al. "More Sound Static Handling of Java Reflection". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. APLAS '15. Pohang, Korea: Springer, 2015.

[146] David Canfield Smith. "Pygmalion: A Creative Programming Environment". PhD thesis. Stanford University, 1975.

[147] Armando Solar-Lezama and Rastislav Bodik. *Program synthesis by sketching*. Citeseer, 2008.

[148] Johannes Späth et al. "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java". In: *European Conference on Object-Oriented Programming (ECOOP)*. ECOOP '16. Rome, Italy: Springer, 2016, 22:1–22:26. ISBN: 978-3-95977-014-9.

[149] Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. "A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading". In: *Programming Language Design and Implementation (PLDI)*. PLDI '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 196–207. ISBN: 1-58113-199-2.

[150] Manu Sridharan and Rastislav Bodík. "Refinement-based context-sensitive points-to analysis for Java". In: *Programming Language Design and Implementation (PLDI)*. PLDI '06. Ottawa, Ontario, Canada: ACM, 2006, pp. 387–400. ISBN: 1-59593-320-4.

[151] Manu Sridharan et al. "Demand-driven points-to analysis for Java". In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 59–76. ISBN: 1-59593-031-0.

[152] Matt Staats and Corina Pasareanu. "Parallel Symbolic Execution for Structural Test Generation". In: *Proc. of the 19th International Symposium on Software Testing and Analysis*. ISSTA '10. Trento, Italy, 2010, pp. 183–194.

[153] Bjarne Steensgaard. "Points-to analysis in almost linear time". In: *Principles of Programming Languages (POPL)*. POPL '96. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 32–41. ISBN: 0-89791-769-3.

[154] Phillip D. Summers. "A Methodology for LISP Program Construction from Examples". In: *J. ACM* (1977), pp. 161–175.

[155] Frank Tip. "A Survey of Program Slicing Techniques". In: *Journal of Programming Languages* 3 (1995), pp. 121–189.

[156] Emina Torlak and Rastislav Bodik. "Growing Solver-Aided Languages with Rosette". In: *Proc. of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2013. Indianapolis, Indiana, USA, 2013, pp. 135–152.

[157] Omer Tripp et al. "TAJ: effective taint analysis of web applications". In: *Programming Language Design and Implementation (PLDI)*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 87–97. ISBN: 978-1-60558-392-1.

[158] Raja Vallée-Rai et al. "Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?" In: *International Conference on Compiler Construction (CC)*. CC '00. Berlin, Germany: Springer, 2000, pp. 18–34. ISBN: 3-540-67263-X.

[159] Raja Vallée-Rai et al. "Soot - a Java bytecode optimization framework". In: *Proc. of the 1999 Conf. of the Centre for Advanced Studies on Collaborative research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 125–135. URL: `http://dl.acm.org/citation.cfm?id=781995.782008`.

[160] Richard J. Waldinger and Richard C. T. Lee. "PROW: A Step Toward Automatic Program Writing". In: IJCAI '69. Washington, DC, USA: Morgan Kaufmann Publishers Inc., 1969, pp. 241–252.

[161] Tjark Weber. "Using a SAT Solver as a Fast Decision Procedure for Propositional Logic in an LCF-style Theorem Prover?" In: (2005).

[162] Mark Weiser. "Program Slicing". In: *Proc. of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449.

[163] John Whaley and Monica S. Lam. "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams". In: *Programming Language Design and Implementation (PLDI)*. PLDI '04. Washington, DC, USA: ACM, 2004, pp. 131–144. ISBN: 1-58113-807-5.

[164] John Whaley et al. "Using Datalog with Binary Decision Diagrams for Program Analysis". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. APLAS '05. Tsukuba, Japan: Springer, 2005, pp. 97–118.

[165] Nicky Williams et al. "PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis". In: 2005, pp. 281–292.

[166] Guoqing Xu, Atanas Routnev, and Manu Sridharan. "Scaling CFL-Reachability-Based Points-to Analysis using Context-Sensitive Must-Not-Alias Analysis". In: *European Conference on Object-Oriented Programming (ECOOP)*. ECOOP '09. Berlin, Heidelberg: Springer, 2009, pp. 98–122. ISBN: 978-3-642-03013-0.

[167] Suan H. Yong, Susan Horwitz, and Thomas Reps. "Pointer analysis for programs with structures and casting". In: *Programming Language Design and Implementation (PLDI)*. PLDI '99. Atlanta, Georgia, United States: ACM, 1999, pp. 91–103. ISBN: 1-58113-094-5.

[168] Hao Yuan and Patrick Eugster. "An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees". In: *European Symposium on Programming (ESOP)*. ESOP '09. York, UK: Springer, 2009, pp. 175–189. ISBN: 978-3-642-00589-3. DOI: 10.1007/978-3-642-00590-9.

[169] Qirun Zhang et al. "Fast Algorithms for Dyck-CFL-Reachability with applications to Alias Analysis". In: *Programming Language Design and Implementation (PLDI)*. PLDI '13. Seattle, WA, USA: ACM, 2013, pp. 435–446. ISBN: 978-1-4503-2014-6.

[170] Xin Zheng and Radu Rugina. "Demand-driven alias analysis for C". In: *Principles of Programming Languages (POPL)*. POPL '08. San Francisco, California, USA: ACM, 2008, pp. 197–208. ISBN: 978-1-59593-689-9.