NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Recovering Structural Information for Better Static Analysis

George D. Balatsouras

ATHENS

APRIL 2017

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Ανάκτηση Δομικής Πληροφορίας προς Καλύτερη Στατική Ανάλυση

Γεώργιος Δ. Μπαλατσούρας

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2017

PhD THESIS

Recovering Structural Information for Better Static Analysis

George D. Balatsouras

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

Panos Rondogiannis,
Professor NKUA

Manolis Koubarakis,
Professor NKUA

Konstantinos Sagonas,
Associate Professor NTUA

Alex Delis,
Professor NKUA

Mema Roussopoulos,
Associate Professor NKUA

Nikolaos Papaspyrou,
Associate Professor NTUA

Examination Date: May 2, 2017

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Ανάκτηση Δομικής Πληροφορίας προς Καλύτερη Στατική Ανάλυση

Γεώργιος Δ. Μπαλατσούρας

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

Νικόλαος Παπασπύρου,
Αναπληρωτής Καθηγητής ΕΜΠ

Κωνσταντίνος Σαγώνας,
Αναπληρωτής Καθηγητής ΕΜΠ

Ημερομηνία Εξέτασης: 2 Μαΐου 2017

ABSTRACT

Static analysis aims to achieve an understanding of program behavior, by means of automatic reasoning that requires only the program's source code and not any actual execution. To reach a truly broad level of program understanding, static analysis techniques need to create an abstraction of memory that covers all possible executions. Such abstract models may quickly degenerate after losing essential structural information about the memory objects they describe, due to the use of specific programming idioms and language features, or because of practical analysis limitations. In many cases, some of the lost memory structure may be retrieved, though it requires complex inference that takes advantage of indirect uses of types. Such recovered structural information may, then, greatly benefit static analysis.

This dissertation shows how we can recover structural information, first (i) in the context of C/C++, and next, in the context of higher-level languages without direct memory access, like Java, where we identify two primary causes of losing memory structure: (ii) the use of reflection, and (iii) analysis of partial programs. We show that, in all cases, the recovered structural information greatly benefits static analysis on the program.

For C/C++, we introduce a structure-sensitive pointer analysis that refines its abstraction based on type information that it discovers on-they-fly. This analysis is implemented in `cclyzer`, a static analysis tool for LLVM bitcode. Next, we present techniques that extend a standard Java pointer analysis by building on top of state-of-the-art handling of reflection. The principle is similar to that of our structure-sensitive analysis for C/C++: track the use of reflective objects, *during* pointer analysis, to gain important insights on their structure, which can be used to "patch" the handling of reflective operations on the running analysis, in a mutually recursive fashion. Finally, to address the challenge of analyzing partial Java programs in full generality, we define the problem of "program complementation": given a partial program we seek to provide definitions for its missing parts so that the "complement" satisfies all static and dynamic typing requirements induced by the code under analysis. Essentially, complementation aims to recover the structure of phantom types. Apart from discovering missing class members (i.e., fields and methods), satisfying the subtyping constraints leads to the formulation of a novel typing problem in the OO context, regarding type hierarchy complementation. We offer algorithms to solve this problem in various inheritance settings, and implement them in JPhantom, a practical tool for Java bytecode complementation.

SUBJECT AREA: Programming Languages, Static Analysis

KEYWORDS: Pointer Analysis; Object-Oriented Programming; Type Hierarchy; Reflection

ΠΕΡΙΛΗΨΗ

Η στατική ανάλυση στοχεύει στην κατανόηση της συμπεριφοράς του προγράμματος, μέσω αυτοματοποιημένων τεχνικών συμπερασμού που βασίζονται καθαρά στον πηγαίο κώδικα του προγράμματος, αλλά δεν προϋποθέτουν την εκτέλεσή του. Για να πετύχουν αυτές οι τεχνικές μία ευρεία κατανόηση του κώδικα, καταφεύγουν στη δημιουργία ενός αφηρημένου μοντέλου της μνήμης, το οποίο καλύπτει όλες τις πιθανές εκτελέσεις. Αφηρημένα μοντέλα τέτοιου τύπου μπορεί γρήγορα να εκφυλιστούν, αν χάσουν σημαντική δομική πληροφορία των αντικειμένων στη μνήμη που περιγράφουν. Αυτό συνήθως συμβαίνει λόγω χρήσης συγκεκριμένων προγραμματιστικών ιδιωμάτων και χαρακτηριστικών της γλώσσας προγραμματισμού, ή λόγω πρακτικών περιορισμών της ανάλυσης. Σε αρκετές περιπτώσεις, ένα σημαντικό μέρος της χαμένης αυτής δομικής πληροφορίας μπορεί να ανακτηθεί μέσω σύνθετης λογικής, η οποία παρακολουθεί την έμμεση χρήση τύπων, και να χρησιμοποιηθεί προς όφελος της στατικής ανάλυσης του προγράμματος.

Στη διατριβή αυτή παρουσιάζουμε διάφορους τρόπους ανάκτησης δομικής πληροφορίας, πρώτα (1) σε προγράμματα C/C++, κι έπειτα, σε προγράμματα γλωσσών υψηλότερου επιπέδου που δεν προσφέρουν άμεση πρόσβαση μνήμης, όπως η Java, όπου αναγνωρίζουμε δύο βασικές πηγές απώλειας δομικής πληροφορίας: (2) χρήση ανάκλασης και (3) ανάλυση μερικών προγραμμάτων. Δείχνουμε πως, σε όλες τις παραπάνω περιπτώσεις, η ανάκτηση τέτοιας δομικής πληροφορίας βελτιώνει άμεσα τη στατική ανάλυση του προγράμματος.

Παρουσιάζουμε μία ανάλυση δεικτών για C/C++, η οποία βελτιώνει το επίπεδο της αφαίρεσης, βασιζόμενη σε πληροφορία τύπου που ανακαλύπτει κατά τη διάρκεια της ανάλυσης. Παρέχουμε μία υλοποίηση της ανάλυσης αυτής, στο `cclyzer`, ένα εργαλείο στατικής ανάλυσης για LLVM bitcode. Έπειτα, παρουσιάζουμε επεκτάσεις σε ανάλυση δεικτών για Java, κτίζοντας πάνω σε σύγχρονες τεχνικές χειρισμού μηχανισμών ανάκλασης. Η βασική αρχή είναι παραπλήσια με την περίπτωση της C/C++: καταγράφουμε τη χρήση των ανακλαστικών αντικειμένων, κατά τη διάρκεια της ανάλυσης δεικτών, ώστε να ανακαλύψουμε βασικά δομικά τους στοιχεία, τα οποία μπορούμε να χρησιμοποιήσουμε έπειτα για να βελτιώσουμε τον χειρισμό των εντολών ανάκλασης στην τρέχουσα ανάλυση, με αμοιβαία αναδρομικό τρόπο. Τέλος, ως προς την ανάλυση μερικών προγραμμάτων Java, ορίζουμε το γενικό πρόβλημα της ((*συμπλήρωσης προγράμματος*)): δοθέντος ενός μερικού προγράμματος, πως να εφεύρουμε ένα υποκατάστατο του κώδικα που λείπει, έτσι ώστε αυτό να ικανοποιεί τους περιορισμούς των στατικών και δυναμικών τύπων που υπονοούνται από τον υπάρχοντα κώδικα. Ή διαφορετικά, πως να ανακτήσουμε τη δομή των τύπων που λείπουν. Πέραν της ανακάλυψης των μελών (πεδίων και μεθόδων) των κλάσεων που λείπουν, η ικανοποίηση των περιορισμών υποτυπισμού μας οδηγεί στον ορισμό ενός πρωτότυπου αλγοριθμικού προβλήματος: τη συμπλήρωση ιεραρχίας τύπων. Παρέχουμε αλγορίθμους που λύνουν το πρόβλημα αυτό σε διάφορα είδη κληρονομικότητας (μονής, πολλαπλής, μεικτής) και τους υλοποιούμε στο JPhantom, ένα νέο εργαλείο συμπλήρωσης Java bytecode κώδικα.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Γλώσσες Προγραμματισμού, Στατική Ανάλυση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ανάλυση Δεικτών, Αντικειμενοστρεφής Προγραμματισμός, Ιεραρχία Τύπων, Ανάκλαση

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. I immensely appreciate his endless encouragement, patience, and motivation; his advice and guidance have been priceless. I hope that I can have such a huge impact on a person, as he had on me.

I also thank Alex Delis, Panos Rondogiannis, Mema Roussopoulos, Manolis Koubarakis, Nikos Papaspyrou, and Kostis Sagonas 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. It was a wonderful experience and I'm indebted to all those who made it happen.

A special thanks to my labmates and colleagues: Aggelos Biboudis, Kostas Ferles, George Kollias, George Kastrinis, George Fourtounis, Neville Grech, Anastasis Antoniadis, Efthymios Hadjimichael, Petros Pathoulas, Dimitris Galipos, Stamatis Kolovos, Konstantinos Triantafyllou, and Kostas Saidis. I'm grateful for our interactions, discussions, and arguments over every possible aspect of programming languages and software engineering that intrigued us, and for all the fun we had.

I thank Sofia for standing by me in all times and always being positive and understanding.

Finally, I would like to thank my parents, Dimitris and Aggeliki, for being so supportive and reassuring over all these years.

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Η διατριβή αυτή αφορά τον ευρύτερο τομέα της στατικής ανάλυσης προγραμμάτων, η οποία αποσκοπεί στην αυτόματη κατανόηση του προγράμματος με βάση την εξέταση του πηγαίου του κώδικα, αλλά δίχως να προϋποθέτει την εκτέλεσή του. Σκοπός της συγκεκριμένης διατριβής είναι η διερεύνηση μεθόδων βελτίωσης της ποιότητας της πληροφορίας στατικών αναλύσεων, μέσω της ανάκτησης πληροφορίας περί της δομής των αντικειμένων που δημιουργούνται στη μνήμη. Οι ισχυρότερες εκ των στατικών αναλύσεων για αντικειμενοστεφείς γλώσσες προγραμματισμού χρειάζεται να κατασκευάσουν ένα αφηρημένο μοντέλο της μνήμης, όπου εικονικά αντικείμενα αναπαριστούν (μία ή περισσότερες) διακριτές δεσμεύσεις αντικειμένων. Έτσι, μπορούν να υπολογίσουν μία εκτίμηση της συμπεριφοράς του προγράμματος με σκοπό είτε τη μηχανικά υποβοηθούμενη κατανόηση, είτε την εύρεση σφαλμάτων, ή τη βελτιστοποίηση της απόδοσης του προγράμματος.

Η γνώση της δομής των αντικειμένων αυτών, η οποία συνήθως συνοψίζεται στον τύπο του αντικειμένου, μπορεί να χαθεί μερικώς (1) λόγω χρήσης συγκεκριμένων προγραμματιστικών ιδιωμάτων, (2) όταν η γλώσσα είναι αρκετά χαμηλού επιπέδου (π.χ., C/C++) δίνοντας άμεση πρόσβαση στη μνήμη (π.χ., μέσω αριθμητικής δεικτών), (3) κατά την ανάλυση μερικών προγραμμάτων (δηλαδή, προγραμμάτων για τα οποία δεν διαθέτουμε ολόκληρο τον κώδικα), ή (4) κατά τη χρήση μηχανισμών ανάκλασης (reflection).

Η απώλεια δομικής πληροφορίας σε αρκετές περιπτώσεις μειώνει σημαντικά την αξία της πληροφορίας που παράγει η στατική ανάλυση. Η κύρια θέση της διατριβής είναι η εξής:

> Υπάρχει *υπονοούμενη δομική πληροφορία* στο πρόγραμμα, όσον αφορά τη μνήμη που αυτό δεσμεύει, η οποία μπορεί να βελτιώσει τη ποιότητα του αφηρημένου μοντέλου της μνήμης, όπως αυτό κατασκευάζεται από στατική ανάλυση του προγράμματος. Η δομική αυτή πληροφορία δεν είναι άμεσα διαθέσιμη, αλλά μπορεί να ανακτηθεί μέσω σύνθετου συμπερασμού, κυρίως βάσει της ανίχνευσης έμμεσης χρήσης τύπων στο πρόγραμμα.

Οι τεχνικές που προτείνονται για την ανάκτηση δομικής πληροφορίας είναι οι εξής:

- Για προγράμματα C/C++ (ως τυπικό παράδειγμα γλώσσας με άμεση πρόσβαση στη μνήμη):

  Προτείνουμε την επέκταση του αφηρημένου μοντέλου μνήμης, ώστε αυτό να διέπεται από μεγαλύτερη διακριτότητα των αντικειμένων που δημιουργεί, αναδεικνύοντας βασικά στοιχεία της εσωτερικής τους δομής. Συγκεκριμένα, αυτό περιλαμβάνει τη δημιουργία διακριτών αντικειμένων που αναπαριστούν πεδία, θέσεις πινάκων, καθώς και πολλαπλούς τύπους του ίδιου αντικειμένου, και το κατάλληλο χειρισμό τους ώστε να προσθέσουν στην ακρίβεια της ανάλυσης.

Όσον αφορά τους τύπους κάθε αντικειμένου, αυτοί ανιχνεύονται δυναμικά κατά την διάρκεια της ανάλυσης, παρακολουθώντας την κανονική ροή των αρχικών αντικειμένων εφόσον αυτά έχουν άγνωστο τύπο. Οι δυναμικές αυτές τεχνικές καταλήγουν σε έναν αμοιβαία αναδρομικό υπολογισμό, όμοιο με αυτό της δυναμικής κατασκευής του γράφου κλήσεων (on-the-fly call-graph construction).

Με την επέκταση αυτή του μοντέλου της μνήμης, η ανάλυση μπορεί να διατηρήσει πλήρη ακρίβεια κατά την εικονική κλήση μεθόδων σε αντικειμενοστρεφή κώδικα, ακόμα κι αν αυτές έχουν μεταφραστεί σε πολλαπλές χαμηλού επιπέδου εντολές, το οποίο είναι αναμενόμενο στην περίπτωση μίας χαμηλού επιπέδου γλώσσας όπως η C/C++.

- Για προγράμματα Java (ως τυπικό παράδειγμα γλώσσας υψηλότερου επιπέδου):

  Η βασική απώλεια δομικής πληροφορίας στην περίπτωση της Java, ως υψηλού επιπέδου γλώσσα που δεν παρέχει απευθείας πρόσβαση στην μνήμη, είναι η ανάλυση μερικών προγραμμάτων, δηλαδή προγραμμάτων τα οποία έχουν αναφορές σε κλάσεις/μεθόδους οι οποίες λείπουν από το πρόγραμμα προς ανάλυση. Σε αυτή την περίπτωση, μπορούμε να ανακτήσουμε τουλάχιστον κάποια πληροφορία τύπου και σχέσεων κληρονομικότητας των κλάσεων που απουσιάζουν, καθώς και ένα ελάχιστο υποσύνολο των μελών τους, με βάση τη χρήση τους στο υπάρχον μέρος του προγράμματος. Έτσι, μπορεί να κατασκευαστεί ένα πλήρες πρόγραμμα που να πληρεί τις εγγυήσεις ορθότητας του Java Verifier.

  Η βασική δυσκολία σε αυτή την κατασκευή έγκειται στην συμπλήρωση της ιεραρχίας των κλάσεων. Οι υπάρχουσες σχέσεις υποτυπισμού θα πρέπει να συμπληρωθούν έτσι ώστε να σχηματίσουμε μία πλήρη ιεραρχία που να μην εισάγει κυκλικές εξαρτήσεις και να ικανοποιεί λοιπούς περιορισμούς (π.χ., μία κλάση στη Java μπορεί να κληρονομήσει μόνο μία κλάση, ενώ δεν ισχύει το ίδιο για ένα interface). Το πρόβλημα αυτό ανάγεται σε θεμελιώδη αλγοριθμικά προβλήματα θεωρίας γράφων με πιθανώς ευρύτερο ενδιαφέρον. Παρουσιάζονται αλγόριθμοι προς επίλυση αυτών των προβλημάτων.

  Μία δεύτερη περίπτωση απώλειας δομικής πληροφορίας προγραμμάτων Java είναι η χρήση του μηχανισμού ανάκλασης (reflection), ο οποίος δίνει τη δυνατότητα σε ένα πρόγραμμα να παρατηρεί δυναμικά τη δομή των κλάσεων και των αντικειμένων στη μνήμη κι επιτρέπει ακόμα και την τροποποίησή τους, χωρίς να προϋποθέτει κάποια στατική γνώση των τύπων ή της γενικότερης μορφής τους. Παρότι κώδικας που χρησιμοποιεί ανάκλαση μπορεί να είναι εντελώς αγνωστικός ως προς τα αντικείμενα που χειρίζεται, μία στατική ανάλυση θα πρέπει να εκτιμήσει σωστά τη μορφή τους, ώστε να είναι σε θέση να προσεγγίσει τη δυναμική συμπεριφορά του προγράμματος. Προτείνουμε μία σειρά τεχνικών για τη δυναμική ανίχνευση των τύπων και της δομής αυτών των αντικειμένων.

Το περιεχόμενο της διατριβής αποτελείται από επτά κεφάλαια. Το πρώτο κεφάλαιο περιέχει μία σύντομη εισαγωγή περί του αφηρημένου μοντέλου μνήμης των στατικών αναλύσεων και των περιπτώσεων όπου χάνεται βασική δομική πληροφορία των αντικειμένων. Επίσης, παρουσιάζεται η ερευνητική αλλά και η πρακτική συνεισφορά της διατριβής.

Στατική ανάλυση και ανάκτηση δομικής πληροφορίας για C/C++. Το δεύτερο κεφάλαιο μελετά τις τεχνικές ανάκτησης δομικής πληροφορίας σε χαμηλού επιπέδου γλώσσες με άμεση πρόσβαση στη μνήμη, όπως η C/C++. Τα βασικά χαρακτηριστικά της C/C++ που προκαλούν απώλεια δομικής πληροφορίας είναι:

– η δυνατότητα αποθήκευσης της διεύθυνσης μνήμης ενός πεδίου (ή θέσης πίνακα) κάποιου αντικειμένου

– οι χαμηλού επιπέδου ρουτίνες δέσμευσης μνήμης (π.χ., `malloc()`) που αγνοούν τους τύπους των αντικειμένων που κατασκευάζουν

– τα εμφωλευμένα αντικείμενα.

Παρουσιάζεται ένα ανανεωμένο αφηρημένο μοντέλο, με βασικό χαρακτηριστικό τη μεγαλύτερη διακριτότητα των αντικειμένων που κατασκευάζει, το οποίο επιτρέπει την ανεμπόδιστη καταγραφή των τύπων σε αρκετές περιπτώσεις όπου κάτι τέτοιο δεν θα ήταν δυνατόν με τις καθιερωμένες τεχνικές. Για να εξασφαλίσουμε κάτι τέτοιο, βασιζόμαστε σε δυναμικές τεχνικές διασύνδεσης αντικειμένων με υπάρχοντες τύπους, των οποίων η ισχύς έγκειται στο ότι δρουν ταυτόχρονα ως καταναλωτές αλλά και παρασκευαστές της πληροφορίας περιεχομένων των δεικτών που υπολογίζει η βασική ανάλυση. Συγκεκριμένα, όταν η ανάλυση ανιχνεύει ότι η ίδια οδηγία δέσμευσης μνήμης, για την οποία δεν γνωρίζουμε τον τύπο του αντικειμένου που κατασκευάζει, χρησιμοποιεί τη μνήμη αυτή με πολλούς διαφορετικούς τύπους, τότε για κάθε έναν από τους τύπους αυτούς, η ανάλυση κατασκευάζει δυναμικά ένα νέο αφηρημένο αντικείμενο και το συσχετίζει με την αρχική οδηγία. Γνωρίζοντας πλέον τον τύπο των αντικειμένων αυτών, η ανάλυση είναι σε θέση να χειριστεί με ακρίβεια τη διευθυνσιοδότηση εσωτερικών πεδίων και θέσεων πινάκων τους, τα οποία επίσης αναπαρίστανται ως διακριτά αντικείμενα με πλήρη γνώση του τύπου τους.

Παρουσιάζουμε επίσης επεκτάσεις της ανάλυσης για (1) αριθμητική δεικτών, (2) αναγνώριση ταυτοτικών διευθύνσεων μνήμης, (3) δομική συμβατότητα τύπων και (4) χειρισμό λειτουργιών αντιγραφής μνήμης. Χρησιμοποιούμε κανόνες συμπερασμού για να παρουσιάσουμε το σύνολο των τεχνικών μας.

Παρέχουμε το εργαλείο `cclyzer`[1] για στατική ανάλυση προγραμμάτων LLVM bitcode (μία ενδιάμεση γλώσσα για C/C++ που χρησιμοποιείται από τον μεταγλωττιστή clang), το οποίο περιλαμβάνει υλοποιήσεις των τεχνικών μας στη γλώσσα Datalog. Για την αξιολόγηση του συνόλου των τεχνικών που παρουσιάστηκαν, συγκρίνουμε με μία από τις πιο διαδεδομένες αναλύσεις για C/C++ με δυνατότητα διάκρισης πεδίων [104, 105] και δείχνουμε πως οι τεχνικές μας αυξάνουν σημαντικά την ακρίβεια της ανάλυσης.

Χειρισμός ανάκλασης για στατική ανάλυση Java. Το τρίτο κεφάλαιο μελετά τις τεχνικές ανάκτησης δομικής πληροφορίας σε πρόγραμματα Java, τα οποία κάνουν χρήση του μηχανισμού ανάκλασης (reflection). Ο μηχανισμός αυτός επιτρέπει σε προγράμματα Java

[1] Το `cclyzer` είναι λογισμικό ανοικτού κώδικα, προσβάσιμο στη διεύθυνση: `https://github.com/plast-lab/cclyzer`

να προσομοιώσουν τη συμπεριφορά δυναμικών γλωσσών κι επιτρέπουν τη συγγραφή πλήρως πολυμορφικού κώδικα που δεν χρειάζεται να γνωρίζει τίποτα για τους στατικούς τύπους του προγράμματος. Η απουσία των στατικών τύπων, ωστόσο, θέτει αρκετές δυσκολίες στη στατική ανάλυση του προγράμματος.

Ένα παράδειγμα χρήσης ανάκλασης είναι το παρακάτω:

```
String className = ... ;
Class c = Class.forName(className);
Object o = c.newInstance();
String methodName = ... ;
Method m = c.getMethod(methodName, ...);
m.invoke(o, ...);
```

Σε αυτές τις περιπτώσεις, μία στατική ανάλυση αδυνατεί να προβλέψει με ακρίβεια τη μορφή των αντικειμένων που θα δημιουργηθούν και τις μεθόδους που θα κληθούν δυναμικά, καθώς αυτό θα χρειαζόταν γνώση των τιμών των συμβολοσειρών (π.χ., της `className`) που χρησιμοποιούνται για την ανάκτηση κλάσεων, πεδίων, ή και μεθόδων.

Οι τεχνικές που παρουσιάζουμε για την ανάκτηση δομικής πληροφορίας αντικειμένων που σχετίζονται με κώδικα ανάκλασης είναι οι εξής:

- Μερική ανάλυση των συμβολοσειρών που χρησιμοποιούνται για ανάκτηση κλάσεων, πεδίων και μεθόδων με χρήση ανάκλασης. Οι τεχνικές μας αποσκοπούν στην εύρεση υπο-συμβολοσειρών αυτών των ονομάτων (οι οποίες, σε αντίθεση με τις συμβολοσειρές του πλήρους ονόματος, εμφανίζονται αυτούσιες στο πρόγραμμα) και να παρακολουθήσουν τη ροή τους ακόμα κι όταν αυτή ξεπερνάει τα όρια συναρτήσεων και περιλαμβάνει αποθήκευση σε άλλα αντικείμενα.

- Τεχνικές όμοιες με αυτές που προτείνουμε για C/C++ βασίζονται στη παρακολούθηση της χρήσης των αφηρημένων αντικειμένων, ως προς τους τύπους με τους οποίους χρησιμοποιούνται (και τα πεδία ή μεθόδους που προσπελαύνουν) και την αξιοποίηση της πληροφορίας αυτής για να καθορίσουν ποια ήταν τελικά τα ονόματα (τύπων, μεθόδων, κ.ά.) που χρησιμοποιήθηκαν για τη δημιουργία αυτών των αντικειμένων, αλλά η ανάλυση προηγουμένως δεν ήταν σε θέση να γνωρίζει. Με τη γνώση αυτή, η ανάλυση είναι πλέον σε θέση να διορθώσει τον προηγούμενο χειρισμό της ανάκλασης και να κατασκευάσει αντικείμενα με ακριβέστερη γνώση της δομής τους.

- Αντίστοιχη τεχνική που προτείνουμε παρατηρεί επίσης την χρήση των αφηρημένων αντικειμένων αλλά δεν διορθώνει προηγούμενο χειρισμό, παρά μόνο επεμβαίνει τοπικά (στο σημείο που η ανάλυση αποκτά σαφέστερη γνώση για τη δομή και το τύπο τους).

Επίσης, γίνεται μία σύγκριση της πραγματικής δυναμικής συμπεριφοράς των προγραμμάτων αναφοράς DaCapo 9.12-Bach, με το αποτέλεσμα στατικής ανάλυσης που χρησιμοποιεί τις τεχνικές που παρουσιάζουμε. Η σύγκριση μεταξύ του δυναμικού και των στατικών γράφων κλήσεων αποτυπώνει τη βελτίωση στην ορθότητα της στατικής ανάλυσης που επιφέρουν οι τεχνικές μας.

Συμπλήρωση μερικών προγραμμάτων και ιεραρχίας τύπων. Το τέταρτο κεφάλαιο μελετά το πρόβλημα της συμπλήρωσης ιεραρχίας κλάσεων. Η συμπλήρωση ιεραρχίας προκύπτει κατά το γενικότερο πρόβλημα συμπλήρωσης μερικών προγραμμάτων Java, το οποίο προτείνουμε ως μία γενική λύση στην ανάγκη ανάλυσης μερικών προγραμμάτων. Η ανάγκη αυτή προκύπτει από τη δυνατότητα που προσφέρει η Java για εκτέλεση μερικών προγραμμάτων (μέσω της δυναμικής φόρτωσης κλάσεων), εφόσον τα μέρη του προγράμματος που λείπουν δεν είναι αναγκαία κατά την εκτέλεση. Η δυνατότητα αυτή έχει δημιουργήσει τη τάση ευρείας χρήσης βιβλιοθηκών που συχνά καθιστούν μη πρακτική, αν όχι ανέφικτη, την ανάλυση του πλήρους προγράμματος. Κατά μία έννοια, η συμπλήρωση μερικού προγράμματους ισοδυναμεί με την ανάκτηση της χαμένης δομικής πληροφορίας για τους τύπους που απουσιάζουν και ανακατασκευάζονται ως μέρος του ((συμπληρώματος)).

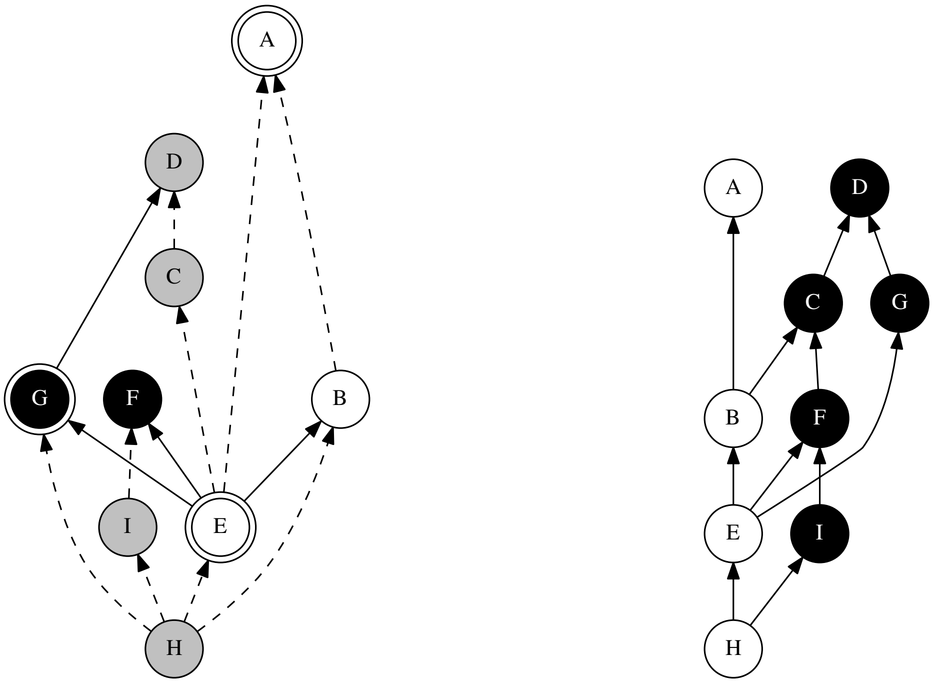
Η συμπλήρωση ιεραρχίας αφορά την ικανοποίηση ενός συγκεκριμένου υποσυνόλου περιορισμών που προκύπτουν κατά τη συμπλήρωση προγράμματος: των περιορισμών υποτυπισμού (του τύπου, η κλάση A πρέπει να είναι υποτύπος της κλάσης B). Το πρόβλημα της συμπλήρωσης ιεραρχίας εξετάζεται σε περιβάλλοντα μονής, πολλαπλής, και μεικτής κληρονομικότητας. Σε κάθε περίπτωση, προσφέρουμε μία γραφοθεωρητική μοντελοποίηση του προβλήματος, καθώς και αλγόριθμο που το επιλύει.

Το Σχήμα 1 παρουσιάζει ένα παράδειγμα του προβλήματος για μεικτή κληρονομικότητα. Στα αριστερά έχουμε την είσοδο του προβλήματος που περιλαμβάνει την υπάρχουσα μερική ιεραρχία καθώς και τους περιορισμούς υποτυπισμού ως διακεκομμένες ακμές. Στα δεξιά έχουμε μία πιθανή λύση του προβλήματος: μία πλήρη ιεραρχία, η οποία ικανοποιεί όλους τους περιορισμούς υποτυπισμού στα αριστερά (δηλαδή για κάθε διακεκομμένη ακμή στα αριστερά, υπάρχει ένα αντίστοιχο μονοπάτι στην πλήρη ιεραρχία που παρουσιάζεται δεξιά). Αυτοί οι περιορισμοί θα πρέπει να ικανοποιηθούν δίχως να αλλάξουν οι εξερχόμενες ακμές των διαθέσιμων τύπων (αφού αυτοί αντιστοιχούν σε κώδικα που ήδη διαθέτουμε). Το μαύρο μέρος της πλήρης ιεραρχίας (το οποίο αντιστοιχεί στα interfaces) θα πρέπει τελικά να σχηματίζει έναν κατευθυνόμενο ακυκλικό γράφο (λόγω πολλαπλής κληρονομικότητας), ενώ το λευκό μέρος (το οποίο αντιστοιχεί στις κλάσεις) θα πρέπει να είναι ένα δέντρο (λόγω μονής κληρονομικότητας). Παρουσιάζουμε αρκετά παραδείγματα που δείχνουν πως η συμπλήρωση ιεραρχίας κλάσεων είναι σαφώς το δυσκολότερο βήμα στο γενικότερο πρόβλημα της συμπλήρωσης μερικών προγραμμάτων.

Για την αξιολόγηση των τεχνικών μας, υλοποιήσαμε το εργαλείο JPhantom[2], για συμπλήρωση μερικών προγραμμάτων Java, το οποίο παρέχει υλοποιήσεις όλων των αλγορίθμων συμπλήρωσης ιεραρχίας τύπων που παρουσιάζουμε. Το JPhantom δέχεται ως είσοδο Java bytecode, στη μορφή ενός JAR αρχείου. Αφού επεξεργαστεί το αρχείο αυτό και ανιχνεύσει όλους τους υπάρχοντες περιορισμούς για τους τύπους που λείπουν, υπολογίζει μία πλήρη ιεραρχία τύπων κι έπειτα κατασκευάζει το συμπλήρωμα του προγράμματος με βάση τα προηγούμενα.

Τέλος, δείχνουμε αποτελέσματα της εφαρμογής του JPhantom σε σύνθετα και ρεαλιστικά

[2] Το JPhantom είναι λογισμικό ανοικτού κώδικα, προσβάσιμο στη διεύθυνση: https://github.com/gbalats/jphantom



(α΄) Γράφος περιορισμών **(β΄) Λύση (πλήρης ιεραρχία)**

Σχήμα 1: Παράδειγμα ενός γράφου περιορισμών ιεραρχίας τύπων για την πλήρη Java. Οι διπλοί κύκλοι αντιστιχούν σε υπάρχοντες τύπους (classes/interfaces), των οποίων οι εξερχόμενες ακμές στη λύση είναι προκαθορισμένες και αναπαρίστανται ως κανονικές ακμές στον αρχικό γράφο. Οι διακεκομμένες ακμές εκφράζουν τους υπάρχοντες περιορισμούς υποτυπισμού. Οι λευκοί κόμβοι αναπαριστούν κλάσεις, οι μαύροι κόμβοι αναπαριστούν intefaces, και οι γκρίζοι κόμβοι αναπαριστούν τύπους οι οποίοι αρχικά είναι αγνώστου είδους.

προγράμματα. Το JPhantom είναι σε θέση να διεκπεραιώσει τη συμπλήρωση των περισσότερων προγραμμάτων με αρκετά χαμηλό χρόνο εκτέλεσης. Ενδεικτικά, αναφέρουμε ότι:

- η συμπλήρωση της βιβλιοθήκης logback-classic ολοκληρώνεται σε λιγότερο από 2 δευτερόλεπτα, ενώ παράγει 148 κλάσεις συμπληρώματος και ικανοποιεί 212 διαφορετικούς περιορισμούς υποτυπισμού
- η συμπλήρωση της βιβλιοθήκης jruby (μεγέθους 19MB) απαιτεί 14 δευτερόλεπτα και αποτελεί τον μεγαλύτερο χρόνο εκτέλεσης που έχουμε δει στη πράξη.

Δείχνουμε επίσης πως η συμπλήρωση μερικού προγράμματος (δηλαδή, η ανάκτηση της χαμένης δομικής πληροφορίας για τους τύπους που απουσιάζουν) βελτιώνει τη στατική ανάλυση του προγράμματος, όπως αυτή πραγματοποιείται από το εργαλείο Doop. Συγκεκριμένα, συγκρίνουμε τρεις αναλύσεις:

- αυτή του αρχικού (πλήρους) προγράμματος,

- την ανάλυση του μερικού προγράμματος (χωρίς συμπλήρωση), και
- την ανάλυση του μερικού προγράμματος με συμπλήρωση.

Μετρώντας τις προσβάσιμες μεθόδους (δηλαδή αυτές που η εκάστοτε ανάλυση υπολογίζει πως είναι δυνατόν να φθάσει κάποια πιθανή εκτέλεση) βλέπουμε ότι, δίχως συμπλήρωση, η ανάλυση του μερικού προγράμματος αποκλίνει εξαιρετικά από την ανάλυση του πλήρους προγράμματος (λόγω ελλιπούς χειρισμού). Από την άλλη, η συμπλήρωση αντιμετωπίζει σε μεγάλο βαθμό αυτό το πρόβλημα και καταφέρνει να προσεγγίσει σημαντικά τη πλήρη εικόνα.

Στο πέμπτο κεφάλαιο διερευνούμε σχετική ερευνητική δουλειά, για τα τρία βασικά μέρη του συνόλου των τεχνικών που παρουσιάσαμε. Έπειτα, αναφέρουμε πηγές σχετικές με τον ευρύτερο τομέα της στατικής ανάλυσης προγραμμάτων και παρουσιάζουμε διαφορετικές μεθοδολογίες με ιδιαίτερο ενδιαφέρον. Κλείνοντας, στο έκτο και τελευταίο κεφάλαιο παρουσιάζονται μελλοντικές ερευνητικές κατευθύνσεις και τελική εκτίμηση της διατριβής.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

> Smokey, my friend, you are entering a world of pain.
>
> *Walter Sobchak*

Static program analysis is a vast field with broad uses; an umbrella term for many different methodologies (Hoare logic [41, 59, 102, 110], model checking [25, 26, 37, 106], symbolic execution [19, 60, 70, 103], abstract interpretation [27–29], data-flow analysis [64, 68, 69, 96, 108, 117], and so on) that aim to automatically obtain an understanding of a program's behavior, without running it. Nowadays, one form or another of static analysis can be found in many different contexts: compilers, IDEs, editors, linters, or even dedicated bug finders and security analyzers. The ends of a static analysis tool are equally diversified, ranging from bug finding and program verification to optimization, or even aided program comprehension.

Along with static analysis tools, programming languages have evolved as well, becoming more high-level throughout the years, introducing many layers of abstraction, before eventually translating the program to the machine's native opcodes. High-level languages are appealing because they are easier to program in, and maintain. Less programming effort (e.g., in terms of lines of code) is needed to express some computation. Virtual machines have even abstracted away the platform where the code will run. Instead, programs of managed languages are translated to machine code for some *virtual machine*, and hence may run on any platform that provides a backend that emulates this virtual machine.

Software has evolved too. Complex design patterns, immense libraries, frameworks implementing inversion of control, over-involved build tools, and many other complicacies pose significant challenges to program understanding.

As one would expect, static analyses have struggled to keep up with the ever-increasing complexity of software and the programming languages it is written in; the very task of automated program understanding has become daunting, yet even more valuable.

The most promising and powerful of existing static analysis techniques rely on the creation of some *abstract memory model* of the program. What objects will the memory contain, at some state of execution? What will their structure be like? A faithful abstract representation of the actual memory is, however, a demanding task; its precision often decisive for the value of whatever the static analysis is aiming to eventually compute (be it the identification of complex bug patterns or the opportunities for effective optimizations).

Thesis.

> There is *implicit structural information* in the program, about the memory it will allocate, that can improve the quality of the abstract memory model constructed by static analysis. This structural information is not readily available,

but may be recovered via inference, primarily by tracking the use of types in the program.

We provide a number of techniques that recover such lost memory structure, in two different settings: (1) in C/C++ programs, as a typical case of low-level code with direct memory access, where the program's memory structure is often lost due to specific programming idioms and the inherent low-level nature of the language, and (2) in Java programs, where, despite the high-level nature of the language, structural information may be lost (a) for *partial programs* (i.e., libraries or any programs that lack some of their parts), which, in the form of Java Archives (JARs), constitute the main distributable code entity of this managed language, or (b) due to Java's *reflection* mechanism, which allows runtime inspection of classes, interfaces, fields and methods, and can be used to instantiate new objects, invoke methods, get/set field values, and so on, without exact static type information (e.g., the name of the method to be invoked can be created dynamically using plain string operations).

1.1 Impact

In this section, we will briefly explain the main contributions of this dissertation, from both a scientific and a practical perspective.

1.1.1 Scientific Contributions

A weakly-typed language, such as C or C++, exposes pointers as numeric values and allows the programmer to perform arbitrary arithmetic on them. These *pointer arithmetic* capabilities can be used to bypass the language's type system. Objects may be allocated in memory without any local information about their intended type, at the allocation site. In fact, the norm for most heap-allocating routines (e.g., `malloc()`) is to return just an array of bytes. These allocations, while in this *untyped state*, flow to various other instructions and may be even stored to type-agnostic raw pointer variables. Normally, when such an allocation was intended to create an object of type T, a cast instruction or an implicit conversion will be used prior to any other instruction that expected an object of this specific type.

Pointer analysis is a static program analysis that determines the values of pointer variables or expressions. For each pointer, it computes a set of memory allocations that the pointer may point to. We refer to this as the *points-to set* of a variable. Since computing an exact model of memory is undecidable, a static analysis needs to sacrifice precision for computability. Thus, the memory allocations of pointer analysis are mere abstractions; a single allocation may represent many concrete objects during some program execution. One such popular abstraction represents memory objects by their allocation sites. Hence, any number of concrete objects allocated at the same instruction correspond to a single abstract object.

In the case of C/C++ programs, the first scientific contribution is a *revised abstract memory model* that differs from the classic allocation-site abstraction approach, by introducing many more abstract objects (not just one per allocation site). Such a finer granularity, in terms of memory abstraction, is a key step for the analysis to maintain strong type information for its abstract objects. After all, the same allocation site can be used in C to create allocations of more than one distinct type. Also, as will be explained later, C allows a pointer variable to point to some field of an allocated object. We tackle this by representing fields and array indices of abstract objects as separate abstract (sub-)objects with their own points-to sets. Hence, the pointer analysis can differentiate between pointing to some abstract object, and pointing to one of its fields (or array indices). This is commonly known as *field sensitivity*. Due to C's exposure of pointers, a field-sensitive pointer analysis is much harder to implement than in a language that does not provide direct memory access (e.g., Java). Our revised memory model aims to extend the domain of abstract objects to naturally express field sensitivity for C and C++.

The second scientific contribution in the C/C++ setting is a technique to enhance pointer analysis precision by *on-the-fly* associating and maintaining type information for all abstract objects in memory. By the term "on-the-fly", we mean that any object-type association is performed simultaneously with the pointer analysis itself (in a manner similar to on-the-fly call-graph construction). The pointer analysis uses the inferred types of abstract objects to produce new points-to facts or filter spurious inferences due to type incompatibility. The points-to sets, on the other hand, drive the creation of new object-type associations that may again alter the produced points-to sets, and so on—all partaking in an interdependent recursive cycle of computation.

We use this technique to collect type hints—indications that some abstract object has type T—and for each type discovered we (on-the-fly again) create a new (typed) variant of the original abstract object. Thus, the same allocation site may produce multiple abstract objects for different types, while those types will be determined through the pointer analysis itself. The plethora of abstract objects generated by this technique is in line with the fine-grained property of our revised abstract memory model.

As an example of a type hint, which demonstrates how these two techniques interact, consider a field access `((P*) p)->f`. Due to analysis imprecision, the analysis may be unable to reason about the type of the abstract object(s) that `p` points to (as it could have been allocated via a generic `malloc()` call with no type indication). Or, it may even have computed that `p` points to objects of incompatible type (that do not contain any `f` field, whatsoever). However, given the present static type information, the analysis will mark `P` as one of the possible types of the base abstract object $\widehat{obj}$ (for any $\widehat{obj}$ that `p` may point to), if the type of the latter is yet unknown. Other objects with known yet incompatible types will be filtered out. Thus, the analysis will create a new *typed* abstract object $\widehat{obj}_P$ of type `P`, which will also flow to the points-to set of variable `p`. This object will now be eligible as the base address for accessing field `f` (type compatibility is guaranteed by the compiler). Finally, the analysis will compute that the expression `p->f` points to the (typed) abstract subobject that represents field `f` of $\widehat{obj}_P$. Hence, at field accesses the analysis will always be able to recover potentially lost structural information.

In the realm of Java, the challenges are quite different. Java is a statically and strongly typed language that does not expose pointers. All objects (except those of primitive types) are allocated on the heap, and accessed via references (allocated on the stack). References are the disciplined equivalent of a C pointer, and allow no pointer arithmetic at all. All heap allocations of Java have a single (dynamic) type, declared at the allocation site. Objects of composite types can only contain references to other objects and there is no way to store a reference to an object's field. Hence, pointer analysis can be expressed via a much simpler memory model, based on the allocation-site abstraction.

However, Java has another crucial difference from C/C++: it is a managed language. All Java code is translated to a platform-independent IR, which is Java bytecode, to be executed by a Java Virtual Machine (JVM). Using just-in-time (JIT) compilation, the JVM will translate the bytecode to machine code—more precisely, the JVM will jit-compile some parts of the bytecode, specifically, the most frequently called methods or methods with long-running loops (also known as *hot spots*), and interpret the rest of it [71].

Java also introduces the concept of a Java Archive (JAR), which is a bundle of class files (compilation units in bytecode format), and possibly other files as well, using a ZIP file format. Since JARs contain essentially bytecode, they are platform-independent as well. Build tools, such as Apache Maven [9], Gradle [45], and Apache Ant [8] have been developed that provide dependency management for Java projects, by automatically downloading Java libraries (in the form of JARs) from online repositories. A Java project needs only provide a list of dependencies, in the form of a well-defined library name and a version number, and its build tool will handle the rest (such as resolving the libraries, and downloading the relevant JARs, including any transitive dependencies they might have).

Aside from the fact that C/C++ is not intended to run on a virtual machine, there are many other reasons why such automatic dependency management and distribution of compiled artifacts is not as common as in Java. To list a few complications:

- A C/C++ library would also need to distribute its header files, so that one would be able to compile against it. There are no header files in Java.
- Aside from providing several versions of a library for different platforms, one would have to provide many versions for different binary compatibility standards as well (Itanium, MSVC8, MSVC9, etc).
- Due to ABI changes, even different versions of the language (e.g., C++98 vs C++11) can break binary compatibility in some cases, for code compiled by different compilers or even from different versions of the same compiler.
- By design, Java class files tend to be quite small in size (a few kilobytes at most). For instance, size is one of the main reasons why Java bytecode is a stack-based representation (i.e., it uses a stack instead of variables to contain the operands of each instruction). C++ object files are considerably less compact. An alternative IR, specifically

designed to reduce code size, could be a necessity, to be able to maintain repositories that contain vast collections of precompiled libraries.

- Java has no explicit link phase that combines compilation units to form an executable program. All classes are linked dynamically in Java (via class loaders), when they are loaded into the JVM. Classes are loaded on demand and the runtime system does not need to know about specific filesystem paths, at all. One could even compile a class against one version of a library, but provide another version at runtime, as long as the relevant signatures match. In C++, compilation involves linking as well.

  The only practice that remotely resembles Java's dynamic class loading is shared libraries (or dynamic-link libraries (DLLs), in Windows). However, those have their own pitfalls. For instance, a single unresolved symbol (missing DLL) will forbid the program from executing at all. Due to complex dependency chains, even identifying the missing DLL is often a difficult task.

All these limitations would make distributing compiled artifacts of C/C++ only marginally better than distributing the code itself.

Now that we have established some of the reasons that account for the prevalence of JARs, we can switch our focus to static analysis again. As far as static analysis is concerned, JARs can be thought of as *partial programs*, since they only contain a subset of the program's classes. In the Java world, where JARs are the most easily obtainable artifact (for the aforementioned reasons), it would be too restrictive from the part of a static analysis to require a whole program to analyze.

Moreover, requiring the whole program (which could comprise a multitude of libraries due to transitive dependencies) could be inconsequential as well. A program often uses an external library A, which in turn depends on another library B, but only needs a subset of A's functionality that does not touch B in any way. Library B is a transitive dependency but may be entirely redundant in any possible execution of the program. (As we have already noted, a C/C++ program cannot even execute in case of undefined symbols, even those due to missing transitive dependencies.)

> The analysis of partial Java programs is, thus, meaningful as some missing parts of code are neither required nor needed for a program to run.

This raises the question:

> What are the challenges of statically analyzing partial Java programs, as in the form of JAR files, or any non-complete (w.r.t. the whole program) collection of class files?

One of the main challenges is that any partial program may fail to satisfy even basic soundness guarantees, as those presumed by the Java verifier itself. Static analysis tools are rarely robust enough to analyze such programs without risking disruptive effects to their results—that is, if they succeed at running at all. Handling *phantom types* (e.g., classes referenced in the partial program, with missing definitions), for which no structural information exists, can throw off even basic assumptions or invariants of a static analyzer.

The most vital aspect of the missing structural information is the *class hierarchy*, the knowledge of subtyping relationships among the various types defined in the program. A partial program provides only a part of the complete class hierarchy; however, many more subtyping relationships are implied in the code itself. For instance, calling a known method that expects a parameter of type `A`, with an argument of type `AImpl`, implies that `AImpl` is a (transitive) subtype of `A`, even in the case that any of the definitions of these two class types are missing. The (complete) original class hierarchy is guaranteed to satisfy this constraint.

Hence, we outline the problem of *class hierarchy complementation* of partial Java programs:

> Given a partial program, how to compute a complete class hierarchy that satisfies any implied type constraint, as expressed in the Java bytecode specification [83].

To compute such a complete class hierarchy is far from trivial. If not done correctly, we could easily end up introducing cyclic dependencies between types (e.g., `A` is a subtype of `B` and `B` is a subtype of `A`), which would violate the language semantics. We can express this problem in pure graph-theoretic terms. The result is two interesting, if not fundamental, graph-theoretic problems that could as well arise in completely different settings due to their generality:

Multiple Inheritance. Given a directed acyclic graph, with a subset of "fixed" nodes (which correspond to our *known* non-phantom classes), and a set of binary path constraints among the nodes (of the form Y reachable from X), how can we extend the graph by adding new edges that do not originate from the "fixed" nodes, so that (i) the graph remains acyclic, and (ii) all path constraints are satisfied (i.e., for each constraint between nodes X and Y, there exists a path from X to Y in the final graph).

Single Inheritance. The problem statement is the same as in the previous setting, with one more constraint: the output graph should be a directed *tree* (instead of a DAG).

As to the solution of the class hierarchy complementation problem, we provide algorithms to solve it in both the multiple and single inheritance cases. More specifically, (1) we present a polynomial-time algorithm that solves any instance of the problem in the multiple inheritance setting, as well as a proof of correctness. For the single inheritance setting, (2) we provide a polynomial-time algorithm for a slightly simplified setting (yet practically quite common): when no phantom supertypes for fixed (i.e., *non-phantom*) nodes are allowed. For the general (single inheritance) case, (3) we provide an algorithm that may perform an exponential search in the worst case, but with many heuristics to improve its performance. Also, for languages such as Java, with single inheritance but multiple subtyping and distinguished class vs. interface types, (4) we demonstrate how the problem can be decomposed into separate single- and multiple-subtyping instances.

Finally, another ubiquitous feature of Java programs that accounts for leaked structural information in most kinds of static analyses is Java's *reflection*. Reflection allows programmers to dynamically inspect objects and classes, find out what methods and fields they declare, and access or modify them in whatever way possible. Given that a Java program can reflectively obtain a member of a class object given just run-time strings, for a static analysis to determine what objects are involved in reflective operations it would need some form of sophisticated string value analysis at least. Even that could prove insufficient in cases where the strings involved come from external sources (e.g., properties files) or are constructed using such low level operations that cannot be modeled precisely enough by the value analysis at hand.

A technique that can be used to recover missing types in reflective operations, without any need for string analysis, is similar to the one we use in the C/C++ setting to discover the types of untyped abstract objects on-the-fly by inspecting their normal flow in the pointer analysis itself. Specifically, we can treat casts (that reflectively generated objects flow to) as type hints for their respective class objects, if we lack more precise type information. The principle is the same: to interleave, into the main points-to analysis, logic that associates types to statically untyped abstract objects, so that these two analysis components can profit from their symbiotic relationship (one being both consumer and producer of the other).

In conclusion, we briefly list the main scientific contributions of this dissertation in both the C/C++ and Java settings:

– a revised abstract memory model for field-sensitive points-to analysis of C/C++ programs

– a technique to recover missing structural information and enhance C/C++ pointer analysis precision by *on-the-fly* associating and inferring missing type information for abstract objects in memory

– a technique to recover missing structural information in Java programs that use reflection that is based on the same principle as in the C/C++ analysis but targets objects involved in reflective operations

– the graph-theoretic modeling of the class hierarchy complementation problem for partial Java programs

– algorithms that solve the class hierarchy complementation problem for both single and multiple inheritance, as well as Java's mixed inheritance (i.e., single inheritance/multiple subtyping) setting.

1.1.2 Practical Contributions

Aside from the scientific contributions of this work, there are significant practical aspects as well. Our techniques are reified in two tools offering immediate real-world benefits. Before we go into these tools and consider their respective gains, we first discuss an important point in the design space of static analyzers, in general, and in that of our tools in particular.

A crucial engineering choice of any static analysis framework is to determine its interaction with the language's build system(s), if any, and the exact point when the analyzer can intervene in the software's build cycle in order to analyze it. This will also have direct repercussions on the nature of the analysis inputs.

For instance, a static analysis tool may choose to completely ignore the compilation and build process, and directly analyze *source code*—this is an approach most often followed by tools performing superficial (mostly syntactic) analysis, such as linters. Being able to analyze software by requiring (only) its source code, can be a blessing or a curse. From a technical standpoint, source code is often very difficult to analyze, given that a language is most often designed to be expressive and may contain a large number of (possibly redundant) syntactic constructs; plain syntactic sugar. A much more minimal core, with the same expressiveness, yet easier to analyze, can often be obtained by some transformation. In fact, the technique of *lowering* the source language, in a series of steps, to a more fundamental form with simpler syntax each time is a standard strategy of compilers, before they finally transform the end result (which, near the end, should be a very simple IR) to machine code. Thus, a static analysis tool that directly analyzes source code could benefit greatly by hooking to the compiler or performing analysis post compilation. On the other hand, being close to the source code can be valuable for the tool if it needs to report its findings to the end user. The identification of a bug can have little to no value, if the programmer is not able to easily understand how and where it can manifest. Thus, reporting a bug by identifying it in some low-level IR (that the programmer knows nothing about) is meaningless, unless the problem can be traced back to the original source code. Apart from technical matters, another factor to consider is the availability of the source code. A programmer that uses a static analysis tool may not be able, or willing, to provide source code in the first place.

A diametrically opposed alternative is to analyze the final product of the build process: an *executable binary*. There are more advantages to such an approach, other than code (un)availability. First, the WYSINWYX phenomenon (i.e., "What You See Is Not What You eXecute") may account for many missed bugs and vulnerabilities, when the analysis is performed on source code [12]. The main reasons for such discrepancies, are:

- platform-specific compiler choices
- post-compilation modifications to programs
- (strictly) undefined behavior that is, however, allowed by the compiler
- dynamically linked libraries (DLLs), which are typically not available in source-code form
- inlined assembly code.

Also, analyzing binaries has, in general, wider applicability, since the same analysis can handle any number of compiled language(s).

Finally, there is a range of options depending on the language being analyzed, that lie between analyzing source code and binaries. That is, a static analyzer may opt to target an intermediate representation (IR), such as Java Bytecode for languages running on the JVM [83]. The advantages of analyzing Java bytecode, for instance, are the following:

- Java bytecode is, syntactically, much simpler than Java and hence easier to analyze
- most libraries are available in bytecode format; thus, the analysis does not need to provide stubs that model library behavior
- the analysis may, in principle, support any language that runs on the JVM (since it will be compiled to bytecode).

However, analyzing bytecode shares many of the downsides of both the source-code and binary approaches. The WYSINWYX phenomenon may still arise, and requiring to have a working build for a project may be too optimistic in some cases. Hence, all three approaches have their respective benefits and limitations; none is clearly superior to another.

The first major practical contribution of this work is an implementation of our techniques for analyzing C/C++ programs in `cclyzer`,[1] a static analysis tool for LLVM bitcode. LLVM bitcode is a low-level RISC-like intermediate representation, used by the LLVM compiler infrastructure [72] that we will thoroughly present in Chapter 2. Hence, instead of analyzing source or binaries, we chose this IR as our analysis target for reasons similar to those presented for preferring Java bytecode. LLVM bitcode is already being used by a number of tools for many different types of static analysis [50, 55, 75, 76, 78, 129].

Besides field sensitivity and our revised abstract memory model to fully support it, Chapter 2 introduces a limited form of array sensitivity, so that the analysis can differentiate between different array indices in some cases. The combination of these techniques, all implemented in `cclyzer`, are powerful enough to allow analyzing C/C++ programs as though written in a higher-level language, while maintaining a good level of precision. Consider the invocation of a virtual method in C++. In LLVM bitcode, or in any object layout that adheres to the Itanium C++ ABI [62] for that matter, virtual tables are represented as constant arrays of function pointers. Also, an object (i.e., class instance) contains a v-pointer field to its respective v-table. Thus, a virtual call is translated to a series of instructions:

- a load instruction that dereferences the object's v-pointer to get its v-table
- an instruction that adds a relative offset to the start of the v-table, to go to the v-table slot that corresponds to the declared method of the call
- a second load instruction that dereferences this specific v-table slot to get the actual (possibly overriden) method that will be called.

A virtual call in Java bytecode would instead by translated to an `invokevirtual` instruction, without exposing the object layout internals or the implementation of dynamic dispatch. However, due to the low-level nature of C/C++, this is not an option for any IR generic enough to support the full language. Therefore, a practical contribution of `cclyzer` is that

[1] `cclyzer` is publicly available at `https://github.com/plast-lab/cclyzer`

the analysis it performs is able to precisely resolve the method being called, given such translations, as long as it can determine the dynamic type of the receiver object. This is the same level of precision as one would expect from a typical points-to analysis targeting Java.

Regarding the Java setting and the class hierarchy complementation problem, we have implemented JPhantom,[2] a tool that accepts any partial Java program in the form of a JAR file, and generates a complete program containing skeletal versions of any referenced missing classes and interfaces so that the combined result constitutes verifiable Java bytecode with a complete class hierarchy. This tool does not depend on a specific analysis being run. Rather, it can be used as a preprocessing step for any static analysis tool, to allow the analysis of any partial Java program without having to provide custom solutions for the class hierarchy complementation problem or deal with missing references at all.

1.2 Outline

The rest of this dissertation is organized as follows:

- Chapter 2 presents a structure-sensitive pointer analysis for C/C++ programs that employs a fine-grained object abstraction, in order to preserve and be able to recover missing structural information.

  This chapter is based on research already presented in *“Structure-Sensitive Points-To Analysis for C and C++”* [14], but also includes extensions.

- Chapter 3 examines how the reflection capabilities of Java can hinder traditional pointer analyses, and then presents techniques for analyzing reflection (interwoven into the main pointer analysis) to overcome such limitations.

  This chapter draws material from *“More Sound Static Handling of Java Reflection”* [121].

- Chapter 4 introduces the class hierarchy complementation problem and presents algorithms to solve it, in various inheritance settings. It discusses the design and implementation of JPhantom, a tool that employs such algorithms to perform the actual complementation, and evaluates its performance.

  This chapter presents research previously published in *“Class Hierarchy Complementation: Soundly Completing a Partial Type Graph”* [13].

- Chapter 5 first discusses related work that is specific to the previous chapters, and then expands to various other interesting subjects in the broader realm of static analysis.

  Some parts of this chapter are based on the aforementioned papers [13, 14, 121], and some on the survey *“Pointer Analysis”* [120].

[2] `JPhantom` is publicly available at `https://github.com/gbalats/jphantom`

- Chapter 6 concludes this dissertation by assessing our initial thesis and discussing future work.

2. STRUCTURE-SENSITIVE POINTS-TO ANALYSIS FOR C AND C++

> Smokey, this is not 'Nam. This is bowling. There are rules.
>
> *Walter Sobchak*

In the first chapter, we discussed how a static analysis needs to compute an abstract model of memory, but often fails to provide the right abstractions to handle certain aspects of the language being analyzed. This, in turn, leads to a memory model that lacks essential structural information about objects allocated in memory. In C/C++, as a typical example of a language that provides direct memory access, field-insensitive analyses (providing crude abstractions that even fail to distinguish an object from its fields) have long been the favorite approach of most pointer analyses in the literature, due to their simplicity and speed. Such imprecision is prohibitive for a meaningful analysis of C++ programs, where one must extend beyond field sensitivity to be able to reason about v-tables and virtual calls precisely enough.

This chapter presents a points-to analysis for C/C++ that recovers much of the available high-level structure information of types and objects, by applying two key techniques: (1) It records the type of each abstract object and, in cases when the type is not readily available, the analysis uses an *allocation-site plus type* abstraction to create multiple abstract objects per allocation site, so that each one is associated with a single type. (2) It creates separate abstract objects that represent (a) the fields of objects of either struct or class type, and (b) the (statically present) constant indices of arrays, resulting in a limited form of array-sensitivity.

We apply our approach to the full LLVM bitcode intermediate language and show that it yields much higher precision than past analyses, allowing accurate distinctions between subobjects, v-table entries, array components, and more. Especially for C++ programs, this precision is invaluable for a realistic analysis. Compared to the state-of-the-art past approach, our techniques exhibit substantially better precision along multiple metrics and realistic benchmarks (e.g., 40+% more variables with a single points-to target).

2.1 Overview of Techniques Towards Structure Sensitivity

Points-to analysis computes an abstract model of the memory that is used to answer the following query: *What can a pointer variable point-to, i.e., what can its value be when dereferenced during program execution?* This query serves as the cornerstone of many other static analyses aiming to enhance program understanding or assist in bug discovery (e.g., deadlock detection), by computing higher-level relations that derive from the computed points-to sets. In the literature, one can find a multitude of points-to analyses with varying degrees of precision and speed.

One of the most popular families of pointer analysis algorithms, *inclusion-based* analyses (or Andersen-style analyses [7]), originally targeted the C language, but has been extended over time and successfully applied to higher-level object-oriented languages, such as Java [16, 20, 92, 113, 132]. Surprisingly, precision-enhancing features that are common practice in the analysis of Java programs, such as field sensitivity or online call-graph construction are absent in many analyses of C/C++ [32, 49, 52, 53, 56, 138].

In the case of field sensitivity, the reason behind its frequent omission when analyzing C is that it is much harder to implement correctly than in Java. As noted by Pearce et al. [105], the crucial difference is that, in C/C++, it is possible to have the address of a field taken, stored to some pointer, and then dereferenced later, at an arbitrarily distant program point. In contrast, Java does not permit taking the address of a field; one can only load or store to some field directly. Hence, `load/store` instructions in Java bytecode (or any equivalent IR) need an extra field specifier, whereas in C/C++ intermediate representations (e.g., LLVM bitcode) `load/store` requires only a single address operand. The precise field affected is not explicit, but only possibly computed by the analysis itself.

The effect of such difference in the underlying IRs, as far as pointer analysis is concerned, is far from trivial. In C, the computed points-to sets have an expanded domain, since now the analysis must be able to express that a variable `p` *at some offset* `i` may point-to another variable `q` *at some offset* `j`, with these offsets corresponding to either field components or array elements.

The best-documented approach on how to incorporate field sensitivity in a C/C++ points-to analysis is that of Pearce et al. [104, 105]. The authors extend the constraint-graph of the analysis by adding (positive) weights to edges; the weights correspond to the respective field indices. For instance, the instruction "$\texttt{q = \&(p->f}_\texttt{i}\texttt{)}$" would be encoded as a constraint $q \supseteq p + i$. However, this approach does not take types into account. In fact, types are not even statically available at all allocation sites, since most standard C allocation routines are type-agnostic and return byte arrays that are cast to the correct type at a later point (e.g., `malloc()`, `realloc()`, `calloc()`). Thus, field i is represented with no regard to the type of its base object, even when this base object abstracts a number of concrete objects of different types. As we shall see, the lack of type information for abstract objects is a great source of imprecision, since it results in a prohibitive number of spurious points-to inferences.

We argue that type information is an essential part in increasing analysis precision, even when it is not readily available. The abstract object types should be rigorously recorded in all cases, especially when indexing fields, and used to filter the points-to sets. In this spirit, we present a *structure-sensitive* analysis for C/C++ that employs a number of techniques in this direction, aiming to retrieve high-level structure information for abstract objects in order to increase analysis precision:

1. First, the analysis records the type of an abstract object when this type is available at the allocation site. This is the case with stack allocations, global variables, and calls to C++'s `new()` heap allocation routine.

2. In cases where the type is not available (as in a call to `malloc()`), the analysis deviates from the allocation-site abstraction and creates multiple abstract objects per allocation site: one for every type that the object could have. Thus, each abstract object of type `T` now represents the set of all concrete objects of type `T` allocated at this site. To determine the possible types for a given allocation site, the analysis creates a special type-less object and records the cast instructions it flows to (i.e., the types it is cast to), using the existing points-to analysis. This is similar to the use-based *back-propagation* technique used in past work [80, 85, 124], in a completely different context—handling Java reflection. We will examine this technique in detail, in Chapter 3.

3. The field components of abstract objects are represented as abstract objects themselves, as long as their type can be determined. That is, an abstract object `SO` of struct type `S` will trigger the creation of abstract object $\texttt{SO.f}_\texttt{i}$, for each field $\texttt{f}_\texttt{i}$ in `S`. (The aforementioned special objects trigger no such field component creation, since they are typeless.) Thus, the recursive creation of subobjects is bounded by the type system, which does not allow the declaration of types of infinite size.

4. Finally, the analysis treats array elements similarly to field components (i.e., by representing them as distinct abstract objects, if we can determine their type), as long as their respective indices statically appear in the source code. That is, an abstract object `AO` of array type `[T×N]` will trigger the creation of abstract object `AO[c]`, if the constant `c` is used to index into type `[T×N]`. The object `AO[*]` is also created, to account for indexing at unknown (variable) indices.

As we shall see, the last point offers some form of array-sensitivity as well and is crucial for analyzing C++ code, lowered to an intermediate representation such as LLVM bitcode, in which all the object-oriented features have been translated away. To be able to resolve virtual calls, an analysis must precisely reason about the exact v-table index that a variable may point to, and the method that such an index may itself point-to. That is, a precise analysis should not merge the points-to sets of distinct indices of v-tables.

In summary, the work presented in this chapter makes the following contributions:

- It presents a structure-sensitive pointer analysis that employs key techniques, essential in retrieving high-level structure information of heap objects, thus significantly increasing the precision of the analysis.

- The analysis is implemented and evaluated in `cclyzer`[1], a new pointer analysis framework that operates on LLVM Bitcode. The pointer analysis is expressed in a fully declarative manner, using Datalog.

- We evaluate the precision of our structure-sensitive analysis by comparing to a reimplementation of the Pearce et al. [104, 105] analysis, also operating over the full

[1] `cclyzer` is publicly available at `https://github.com/plast-lab/cclyzer`

LLVM bitcode language. We show that our techniques provide a major precision enhancement for realistic programs.

2.2 C/C++ Pointer Analysis Background and Limitations of Past Approaches

We next discuss essential aspects of precise pointer analysis for C and C++, as well as the key features of the LLVM bitcode intermediate language.

2.2.1 Language Level Intricacies and Issues

Research on pointer analysis in the last decade has shifted much of its focus from the low-level C language to higher-level object-oriented (OO) languages, such as Java [16, 20, 92, 113, 132]. To a large extent, the industry's paradigm shift to object oriented programming and Java's rising popularity naturally ignited a similar interest shift in the research community.

In points-to analysis, however, one could argue that object-oriented languages in general, and Java, in particular, are better targets than C, for a number of reasons. First, the points-to abstraction [36] is more suited to OO programming, where dynamic object allocations are more common. Furthermore, Java offers a clear distinction: only variable *references* are allocated on the stack, whereas the allocated objects themselves are stored on the heap. Also, class fields can only contain references to other objects, not entire subobjects. Thus, variables point to (heap) objects and objects can only point to each other through their fields. This leads to a clear memory abstraction as well, where objects are commonly represented by their allocation site. A points-to analysis in Java has to compute two sets of edges: (i) a set of unlabeled edges from variables to abstract heap objects, and (ii) a set of field-labeled edges between abstract objects.

This is not the case for C/C++, where:

1. Objects can be allocated both on the stack and on the heap.
2. An object can contain another *subobject* as a field component. In fact, a field may even contain a fixed-size array of subobjects.
3. Any such subobject can have its address taken and stored to some variable, which can be dereferenced later (as can any normal pointer variable) to return the subobject's exact address (i.e., the address of the base object plus the relative byte offset of the given subobject).

Figure 2.1 illustrates the above points. The `Outer` struct type contains a 3-element array of `Inner` subobjects via its field `in`. Unlike in Java, all these subobjects are stored inside the `Outer` instance's allocation; no dereference is needed to access them. On Figure 2.1b, variable `ptr` will hold the address of some subobject of variable (or stack-allocated object) `obj` of the `Outer` type. Variable `ptr` is then used later to store to this field of `obj`. (Note that the two instructions, the store instruction at line 4 and the instruction that returns the field

```
1  typedef struct Inner {
2     int **x;
3     int *y;
4  } Inner;
5
6  typedef struct Outer {
7     void *x;
8     Inner in[3];
9  } Outer;
```

(a) Nested struct declaration

```
1  Outer obj;             // alloc: $\widehat{o_1}$
2  int *g = malloc(...); // alloc: $\widehat{o_2}$
3  int ***ptr = &(obj.in[1].x); ...
4  *ptr = &g;
5  void *q = obj.x;
```

(b) Complex Field Access

```
1  Inner i;
2  Inner *ip = &i;
3  ip = (Inner *) &ip->y;
```

(c) Positive Weight Cycles

Figure 2.1: C example with nested struct types

address at line 3, can even reside in different functions.) In a precise analysis, this should establish that the `in[1].x` field of abstract object $\widehat{o_1}$ (representing the stack allocation for `obj` at line 1), may point to abstract object $\widehat{o_2}$ (representing the heap allocation of line 2).

In contrast, a *field-insensitive* approach (which is common among C/C++ analyses [32, 49, 52, 53, 56, 138]) is to not record offsets at all. This affords simplicity, at the expense of significant loss of precision. A field-insensitive analysis would disregard any offsets of any field or array accesses it encounters and simply compute that $\widehat{o_1}$ points-to (somewhere inside) $\widehat{o_2}$. Any subsequent instruction that accesses *any* field of $\widehat{o_1}$ would have to consider $\widehat{o_2}$ as a possible target. In the case of line 5, the field-insensitive analysis would (over-)conservatively infer that variable `q` may point to $\widehat{o_2}$.

The line of work by Pearce et al. [104, 105] introduces a form of *field sensitivity*, such that the analysis differentiates between different fields of an object by representing them with distinct symbolic offsets. For instance, the i-th field of `p` is encoded as $p + i$. Thus, the effect of an *address-of-field* instruction such as "`q = &(p->f`$_{\mathtt{i}}$`)`"— f$_{\mathrm{i}}$ being the name of the i-th field of `p`—would add the edge (p, q) labeled with i to a constraint graph, to encode that $q \supseteq p + i$: the points-to set of variable q is a superset of that of the i-th field of any object pointed-to by p.

There are several issues with this approach:

1. First, it is not clear how the approach generalizes to nested structures, as in Figure 2.1a. Had a heap allocation $\hat{o}$ (of unknown type) flowed to the points-to set of variable `p`, how could an expression like $p+i$ differentiate between the i-th field of $\hat{o}$ and the i-th field of $\hat{o}$'s first subobject? (Note that the two fields could be of entirely incompatible types.)

2. As Pearce et al. note, imprecision in the analysis may introduce positive weight cycles that lead to infinite derivations, if no other action is taken. For instance, in Figure 2.1c:

   i. Due to the instruction "`ip = &i;`", the points-to set of `ip` should include at least `i`: $ip \supseteq \{i\}$.
   ii. Due to instruction "`ip = (Inner *) &ip->y;`", the corresponding constraint, $ip \supseteq ip + 1$, would induce: $ip \supseteq \{i,\ i.y,\ i.y.y,\ i.y.y.y,\ ...\}$. Of course, an object like $i.y.y$ would make no sense given that no such field exists.

   As a way to overcome this, Pearce et al. assign unique indices to all (local) program variables and their fields, and also record their symbolic ranges (that is, the index where the enclosing lexical scope of each variable ends). Then, they ensure that field accesses only reference memory locations within the same enclosing scope. However, this does not prohibit all redundant derivations: $ip + 1$ may still add to the points-to set irrelevant variables or fields that happen to be in the same enclosing scope.

   Also, this does not work well for heap allocations, since their type, and hence the number of their fields, is unknown. Instead, they are assumed to define as many fields as the largest struct in the program, which will also lead to many redundant derivations.

3. This approach greatly decreases the analysis precision in the presence of factory methods or wrapper functions for allocation routines. Consider the `xmalloc()` function of *GNU Coreutils* in Figure 2.2, which is consistently used instead of `malloc()` to check if the allocation succeeded and abort the program otherwise. The allocation site it contains will represent the union of all struct types, dynamically allocated via `xmalloc()`, by the same abstract object. The i-th field of this abstract object will then represent the i-th field of this union type, losing essential type information by merging unrelated fields (whose types we statically know to be completely different).

```c
/* Allocate N bytes of memory dynamically, with error checking. */
void * xmalloc (size_t n) {
  void *p = malloc (n);
  if (!p && n != 0) xalloc_die ();
  return p;
}
```

Figure 2.2: Generic `malloc()` wrapper with error checking that aborts the program when allocation fails

The common denominator of all these limitations is that they lose any association between abstract objects and their types, due to cases in which type information is not readily available (as in heap allocations). What we propose instead is that the analysis strictly record types for all abstract objects (any abstract object must have a *single* type) and use this type information to filter redundant derivations that arise from analysis imprecision. For heap allocations specifically, where a single allocation site could be used to allocate objects of many different types, we propose a deviation from the standard allocation-site abstraction that creates multiple abstract objects per allocation site (one for each different type allocated there).

2.2.2 The LLVM IR

Our analysis targets C/C++ programs translated to LLVM bitcode. LLVM bitcode is a low-level intermediate representation, similar to an abstract assembly language, and forms the core of the LLVM umbrella project. It defines an extensive *strongly-typed* RISC instruction set, and has the following distinguishing features:

- Instead of a fixed set of registers, it uses an infinite set of temporaries, called *virtual registers*. At the register allocation phase, some of the virtual registers will be replaced by physical registers while the rest will be spilled to memory. All virtual registers are kept in SSA form.

- Program variables are divided into two categories:

  i. variables whose address is taken and can be referenced by pointers
  ii. variables that can never be referenced by pointers.

  The latter are converted to SSA, whereas the former are kept in memory by using: (i) `alloca` instructions to allocate the required space on stack, and (ii) load/store instructions to access or update, respectively, the variable contents, at any point (hence escaping SSA form). This technique has been termed "*partial SSA*" [50].

- Like address-taken variables, global variables are also kept in memory and are always represented by a pointer to their "content" type. However, their space is allocated using a global initializer instead of an `alloca` instruction.

The example of Figure 2.3 illustrates these points regarding the LLVM translation. Figure 2.3a shows the original source code, while Figure 2.3b shows the corresponding LLVM bitcode. Local variable `p` is stored in memory (since its address is taken) and virtual register `%p` holds its address. `%p`'s value can be updated multiple times, using `store` instructions. Likewise, global variable `gv` (of type `int*`) is also kept in memory and pointer `@gv` (of type `int**`) is used to access it. As will be clear later, our analysis follows the variable representation conventions of LLVM and decouples memory allocations from virtual registers (or global variable references). Figure 2.3c depicts the relevant points-to relationships, which capture that `gv` points to `p`. Dashed edges are used to represent *variable points-to edges* (whose source is a virtual register), while solid edges are *dereference edges* between abstract objects.

2.3 Structure-Sensitive Approach

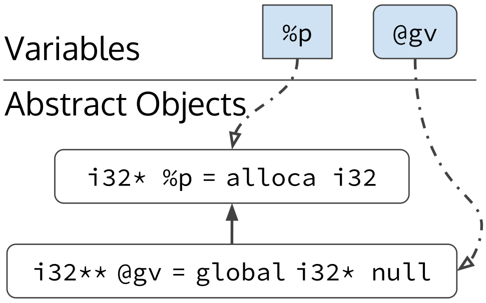
Our analysis approach adds more detail to object abstractions, which serve both as sources and as targets of points-to edges, allowing a more detailed representation of the heap. Although our approach is applicable to C/C++ analysis in general, it is best to see it in conjunction with the LLVM bitcode intermediate language. Just as LLVM bitcode is a

```
int *gv;

void f()
{
  int p = 3;
  gv = &p;
}
```

(a) C source

```
i32** @gv = global i32* null

define void @f() {
  i32* %p = alloca i32
  store i32 3, i32* %p
  store %p, i32** @gv
}
```

(b) LLVM translation



(c) Points-to graph

Figure 2.3: Partial SSA Example

strongly-typed intermediate language, we assign types and offsets to every abstract object value and its points-to relationships. The challenge is that, unlike in the LLVM bitcode type system, such information is not readily available by local inspection of the code—it needs to be propagated by the analysis reasoning itself.

We next discuss the various abstractions of our analysis, in representing its input and output relations. Then, we express the main aspects of our analysis as a set of inference rules.

2.3.1 Abstractions

Figure 2.4 presents the input and output domains of our analysis. We represent functions as a subset of global entities. Thus, G contains all symbols referencing global entities—everything starting with symbol "`@`" in LLVM bitcode. Set V holds temporaries only (i.e., virtual registers), and not global variables. We represent the union of these two sets with P, which stands for *pointer variables* (i.e., any entity whose value may hold some memory address). Our analysis only introduces the set of abstract objects O, that correspond to memory locations.

T	set of program types
L	set of instruction labels
C	program (integer) constants
V	set of virtual registers
G	set of global variables
$F \subseteq G$	program functions
$P = V \cup G$	pointer variables
O	set of abstract objects

Figure 2.4: Analysis Domains

LLVM Instruction	*Operand Types*	*Description*
$\texttt{p} = \texttt{alloca T}, \, nbytes$	$V \times T \times C$	Stack Allocations
$\texttt{p} = \texttt{malloc} \; nbytes$	$V \times (V \cup C)$	Heap Allocations
$\texttt{p} = (\texttt{T}) \, \texttt{q}$	$V \times T \times (P \cup C)$	(No-op) Casts
$\texttt{p} = \texttt{phi}(l_1 : a_1, l_2 : a_2)$	$V \times (L \mapsto (P \cup C))^2$	SSA Phi Node
$\texttt{p} = * \, \texttt{q}$	$V \times P$	Load from Address
$* \, \texttt{p} = \texttt{q}$	$P \times (P \cup C)$	Store to Address
$\texttt{p} = \texttt{\&q->} f$	$V \times P \times C$	Address-of-field
$\texttt{p} = \texttt{\&q} \, [idx]$	$V \times P \times (V \cup C)$	Address-of-array-index
$\texttt{p} = a_0(a_1, a_2, \ldots, a_n)$	$V \times (F \cup V) \times (P \cup C)^n$	Function Call
$\texttt{return p}$	$P \cup C$	Function Return

Figure 2.5: LLVM IR Instruction Set. We also prepend a label $l \in L$ to each instruction (that we omit in this figure). Each such label can be used to uniquely identify its instruction.

The LLVM IR defines an extensive instruction set. However, only a small subset is relevant for the purposes of pointer analysis. Figure 2.5 presents a simplified version of these relevant instructions. The first two instructions are used to allocate memory on the stack and on the heap, respectively. As previously discussed, `alloca` instructions are used for address-taken variables. They accept an extra type argument (absent in `malloc` instructions), which specifies the exact type of the allocation (virtual registers are strongly typed), and the allocation size is a constant. Next, we have `cast` instructions, used solely to satisfy LLVM's type checker since they do not change any memory contents, and `phi` instructions that choose a value depending on the instruction's predecessor. Apart from the standard `load`/`store` instructions, we have two more instructions that, given a memory address operand, return a new address by adding a relative offset that corresponds to either a field or an array element. (Only `load` instructions dereference memory, however.) Finally, we have call and return instructions. Call instructions may also accept a variable (function pointer), as their first argument.

Abstract Objects. Our analysis defines several different kinds of abstract objects that express the exact nature of the allocation. Any abstract object must fall into one of the following categories:

- $\widehat{o_i}$ A stack or heap allocation for instruction (allocation site) $i \in L$.
- $\widehat{o_{i,\mathtt{T}}}$ A (heap) allocation for instruction $i \in L$, specialized for type $\mathtt{T} \in T$.
- $\widehat{o_g}$ A global allocation for global variable or function $g \in G$.
- $\widehat{o.fld}$ A field subobject that corresponds to field "fld" of base object $\hat{o} \in O$.
- $\widehat{o[c]}$ An array subobject that corresponds to the element at constant index $c \in C$ of base object $\hat{o} \in O$.
- $\widehat{o[*]}$ An array subobject that corresponds to any elements at unknown indices of base object $\hat{o} \in O$.

When not using any special notation, we shall refer to a generic abstract object that could be of any of the above forms. This also applies to the base object of the last three categories (which, thus, serve as recursive definitions), allowing us to define arbitrarily complex subobjects such as $\widehat{o.f[4].g[*]}$.

By representing field and array subobjects as separate abstract objects themselves, the handling of instructions that return addresses anywhere but at the beginning of some allocation becomes straightforward. As we shall see at Section 2.3.2, all our analysis has to do is return the relevant abstract object that represents the given subobject of its base allocation. This abstract subobject will have its own distinct points-to set, which will be tracked separately from that of its base allocation or any of the rest of its fields. Thus, it will allow the analysis to retain a certain degree of precision that would be otherwise impossible.

Our analysis computes four main relations:

Variable points-to edges. Edge $\mathtt{p} \mapsto \hat{o} \in P \times O$ records that pointer variable (either virtual register or global variable) $\mathtt{p}$ may point to abstract object $\hat{o}$. Note that virtual registers that correspond to source variables will always point to a single object: the corresponding stack allocation. Temporaries introduced by LLVM bitcode, though, may point to many abstract objects.

Dereference edges. Edge $\widehat{po} \rightsquigarrow \hat{o} \in O \times O$ records that abstract object $\widehat{po}$ may point to abstract object $\hat{o}$. Any object that has a non-empty points-to set (i.e., the object has outgoing dereference edges) may represent a pointer. Dereference edges can only be established by `store` instructions.

Abstract object types. The partial function *type* $: O \nrightarrow T$ records the type of an abstract object. An abstract object can be associated with one type at most, or none at all. Since our analysis uses types to filter redundant derivations, the more types it establishes for abstract objects, the more points-to edges it will compute. Figure 2.6 establishes some basic type relations between the subobjects created by the analysis.

$$\text{Struct Type}\ \frac{\textit{type}(\hat{o}) = \texttt{S} \qquad \textit{type}(\texttt{S}.f) = \texttt{F}}{\textit{type}(\widehat{o.f}) = \texttt{F}} \qquad \text{Array Type}\ \frac{\textit{type}(\hat{o}) = [\texttt{T}] \qquad c \in C}{\textit{type}(\widehat{o[*]}) = \texttt{T} \qquad \textit{type}(\widehat{o[c]}) = \texttt{T}}$$

Figure 2.6: Basic Type Inferences for Abstract Objects.

Call-graph edges. Edge $i \xrightarrow{calls} f \in L \times F$ records that invocation site i may call function f. This also accounts for indirect calls that use function pointers.

2.3.2 Techniques - Rules

Figure 2.7 presents the main aspects of the analysis as a set of inference rules. The first two rules handle stack and heap allocation instructions. All they do is create a new abstract object representing the given allocation site, and assign it to the target variable. In the case of stack allocation, we also record the type of the object, since it is available at the allocation site. The next pair of rules handle global allocations for global variables and functions, respectively, in a similar way. In contrast to the previous rules, we create abstract objects for all global entities, regardless of any instructions (since their allocation in LLVM bitcode is implicit), and record their types.

For cast instructions, we copy any object that flows in the points-to set of the source variable to the points-to set of the target variable. Phi instructions are treated similarly, but we have to consider both of the instruction's operands, regardless of their corresponding labels, since our result must be an over-approximation.

Store instructions are the only way in which the analysis establishes dereference edges. For a store instruction, $*\,\mathtt{p} = \mathtt{q}$, we have to perform the following:

1. First, find the corresponding abstract objects that the two instruction operands point to, by following their outgoing variable points-to edges. Namely: (i) the memory allocation of the value to be stored (abstract object $\hat{o}$), and (ii) the memory allocation that $\hat{o}$ is going to be stored into (abstract object $\widehat{po}$).

2. Then, establish a dereference edge between any two such abstract objects returned, expressing that object $\widehat{po}$ may point to object $\hat{o}$.

The first step simply bypasses the indirection introduced by LLVM bitcode, where operands are represented as virtual registers that point to memory locations. Load instructions perform the opposite operation, and thus are treated symmetrically. For instruction $\mathtt{p} = *\,\mathtt{q}$, we first (i) find the corresponding abstract object that the address operand may point to (abstract object $\widehat{po}$), (ii) then follow any outgoing dereference edge of object $\widehat{po}$ to get any memory location $\widehat{po}$ may point to (object $\hat{o}$), and finally (iii) establish a new variable points-to edge for target variable $\mathtt{p}$, recording that $\mathtt{p}$ may now also point to object $\hat{o}$.

$$\text{Stack}\ \frac{i:\ \mathtt{p} = \mathtt{alloca\ T},\ nbytes}{\mathtt{p} \mapsto \widehat{o_i} \qquad \textit{type}(\widehat{o_i}) = \mathtt{T}} \qquad \text{Heap}\ \frac{i:\ \mathtt{p} = \mathtt{malloc}\ nbytes}{\mathtt{p} \mapsto \widehat{o_i}}$$

$$\text{Global}\ \frac{f \in F}{f \mapsto \widehat{o_f} \qquad \textit{type}(\widehat{o_f}) = \textit{type}(f)} \qquad \frac{\mathtt{g} \in (G\ \ F) \qquad \textit{type}(\mathtt{g}) = \mathtt{T*}}{\mathtt{g} \mapsto \widehat{o_g} \qquad \textit{type}(\widehat{o_g}) = \mathtt{T}}$$

$$\text{Cast}\ \frac{i:\ \mathtt{p} = (\mathtt{T})\,\mathtt{q} \qquad \mathtt{q} \mapsto \hat{o}}{\mathtt{p} \mapsto \hat{o}} \qquad \text{Phi}\ \frac{i:\ \mathtt{p} = \mathtt{phi}(l_1 : a_1,\, l_2 : a_2)}{\forall j:\ a_j \mapsto \hat{o} \ \Rightarrow\ p \mapsto \hat{o}}$$

$$\text{Load}\ \frac{i:\ \mathtt{p} = *\mathtt{q} \qquad \mathtt{q} \mapsto \widehat{po} \qquad \widehat{po} \rightsquigarrow \hat{o}}{\mathtt{p} \mapsto \hat{o}} \qquad \text{Store}\ \frac{i:\ *\mathtt{p} = \mathtt{q} \qquad \mathtt{p} \mapsto \widehat{po} \qquad \mathtt{q} \mapsto \hat{o}}{\widehat{po} \rightsquigarrow \hat{o}}$$

$$\text{Field}\ \frac{i:\ \mathtt{p} = \mathtt{\&q\text{-}>}f \qquad \mathtt{q} \mapsto \hat{o} \qquad \textit{type}(\hat{o}) = \mathtt{S} \qquad \textit{type}(\mathtt{q}) = \mathtt{S*}}{\mathtt{p} \mapsto \widehat{o.f}}$$

$$\text{Array – Const}\ \frac{i:\ \mathtt{p} = \mathtt{\&q}[c] \qquad \mathtt{q} \mapsto \hat{o} \qquad \textit{type}(\hat{o}) = [\mathtt{T}] \qquad \textit{type}(\mathtt{q}) = [\mathtt{T}]*}{\mathtt{p} \mapsto \widehat{o[c]}}$$

$$\text{Array – Var}\ \frac{i:\ \mathtt{p} = \mathtt{\&q}[\mathtt{j}] \qquad \mathtt{q} \mapsto \hat{o} \qquad \textit{type}(\hat{o}) = [\mathtt{T}] \qquad \textit{type}(\mathtt{q}) = [\mathtt{T}]*}{\mathtt{p} \mapsto \widehat{o[*]}}$$

$$\text{Call}\ \frac{i:\ \mathtt{p} = a_0(a_1, a_2, \ldots, a_n) \qquad a_0 \mapsto \widehat{o_f} \qquad f \in F}{i \xrightarrow{calls} f(p_1, p_2, \ldots, p_n) \qquad \forall j:\ a_j \mapsto \hat{o} \ \Rightarrow\ p_j \mapsto \hat{o}}$$

$$\text{Ret}\ \frac{i:\ \mathtt{p} = a_0(\ldots) \qquad i \xrightarrow{calls} f(\ldots) \qquad j:\ \mathtt{return\ q} \qquad j \in \textit{body}(f) \qquad \mathtt{q} \mapsto \hat{o}}{\mathtt{p} \mapsto \hat{o}}$$

$$\text{Heap-bp}\ \frac{i:\ \mathtt{p} = \mathtt{malloc}\ nbytes \qquad j:\ \mathtt{w} = (\mathtt{T*})\,\mathtt{q} \qquad \mathtt{q} \mapsto \widehat{o_i}}{\mathtt{p} \mapsto \widehat{o_{i,\mathtt{T}}} \qquad \textit{type}(\widehat{o_{i,\mathtt{T}}}) = \mathtt{T}}$$

Figure 2.7: Inference Rules

The next three rules (Field, Array–Const, Array–Var) model field sensitivity. The rule handling field accesses, such as $\mathtt{p} = \mathtt{\&q\text{-}>}f$, finds any object $\hat{o}$ that base variable $\mathtt{q}$ may point to, and returns $\hat{o}$'s relevant field subobject $\widehat{o.f}$. However, a key element is that $\hat{o}$ is only considered as a base object if its type matches the declared (struct) type of $\mathtt{q}$ (recall that LLVM bitcode is strongly typed). This precludes any untyped heap allocations as possible base objects. Otherwise, the analysis would end up creating untyped field subobjects too, further fueling imprecision. Thus, we are able to maintain an important invariant of our structure-sensitive analysis: *only create field (or array) subobjects whose types we are* *able to determine*. Effectively, LLVM bitcode imposes *strong typing on variables*, while our analysis extends the treatment to *abstract objects*.

Array element accesses are treated similarly and they, too, maintain this invariant. However, we distinguish array accesses using a constant index from those using a variable (i.e., unknown) index. In the former case, we return the array subobject $\widehat{o[c]}$, which represents the subobject at index c. In the latter case, we return $\widehat{o[*]}$, which represents the *unknown* index. Essentially, this treatment allows our analysis to track independently the points-to sets of array indices that are statically known to be different, yielding a form of *array-sensitivity*.

Call and return instructions as modeled as assignments: (i) from any actual argument a_j to its respective formal parameter f_j, and (ii) from any returned value `q` to the target variable of the call instruction `p`. Like cast instructions, they simply copy the points-to sets from the assignment's source to its target. However, the rule that handles call instructions also records call-graph edges. When the function operand a_0 may point to abstract object o_f, representing function f, we record an edge from the given call site to function f. This handles both direct and indirect calls (i.e., via function pointers).

How to produce type information for unknown objects. Our analysis only allows taking the address of fields of objects whose type is known. This prevents loading and storing from/to fields of objects without types. Such objects can only be used as identity markers. Yet C and C++ allow the creation of untyped objects. Their handling is a key element of the analysis.

The Heap-bp rule implements the *use-based back-propagation* technique [80, 85, 124], which creates multiple abstract objects per (untyped) allocation site. The rule states that when an (untyped) heap object $\widehat{o_i}$ (allocated at instruction i) flows to some cast instruction j, where it is cast to type `T`, we augment the points-to set of i's target variable `p` with a new abstract object $\widehat{o_{i,\mathtt{T}}}$, specialized for the given type. The insight behind this rule is that, even when the program performs an allocation via a type-agnostic routine like `malloc()`, the allocation will be later cast to its intended type before being used. By using this technique, the original untyped allocation will be prevented from creating any untyped subobjects, but as soon as the possible type of the allocation is discovered, the new abstract typed object will succeed where the untyped one has failed. Note that instructions i and j could occur in distant parts of the program, as long as the analysis can establish that the object allocated at instruction i flows to j.

This treatment successfully deals with generic allocation wrappers or factory methods. In this case, the wrapped allocation will flow to multiple cast instructions, and thus create multiple typed variations of the original object. However, in each case, only the object with the correct matching type will be used as a base for any subsequent address-of-field instructions. The rest of the objects will be filtered, since they are indeed irrelevant.

2.3.3 Partial Order of Abstract Objects

As the observant reader may have noticed, the rules of Figure 2.7 about accesses or array elements are not sound. Consider the example of Figure 2.8. Variable `p` points to a heap allocation. Three different store instructions take place: (i) one that stores `&i` to index 1, (ii) one that stores `&j` to index 3, and (iii) one that stores `&k` to some variable index. When loading from index 1, the analysis has to return both `&i` and `&k` (since the value of variable `idx` may be equal to 1), but not `&j`, which is stored to a different index. Conversely, when loading from a variable index, the analysis has to return all three addresses, since the index could be equal to any constant.

```
int i, j, k, idx;
...
int **p = malloc(...);
p[1] = &i;
p[3] = &j;
p[idx] = &k;
int *x = p[1];  // yields {i, k}
int *y = p[2];  // yields {k}
int *z = p[j];  // yields {i, j, k}
```

Figure 2.8: Accessing array elements.

Using our array-sensitive approach, we ensure that indices 1, 3, and "$*$" (unknown) are associated with separate points-to sets that are not merged. To handle loads correctly, though, we have to be able to reason about implicit associations of abstract objects, due to possible index aliases. Thus, we say that object $\widehat{o[*]}$ "generalizes" object $\widehat{o[c]}$ (for the same base object $\hat{o}$), since loading from $\widehat{o[*]}$ must always return a superset of the objects returned by loading from $\widehat{o[c]}$, for any constant c. This concept extends even to deeply nested subobjects. For instance, an object $\widehat{o.f_1[*][2].f_2[*]}$ generalizes object $\widehat{o.f_1[4][2].f_2[*]}$.

We can think of this binary relation between abstract objects as a partial order over domain O and define it appropriately.

Definition 2.1. *Abstract Object Generalization Order.* An abstract object $\hat{y} \in O$ *generalizes* an abstract object $\hat{x}$, denoted $\hat{x} \sqsubseteq \hat{y}$, if and only if:

$$
\begin{gathered}
\hat{x} = \hat{y} \\
\lor \\
(\,\hat{x} = \widehat{p[*]} \;\lor\; \hat{x} = \widehat{p[c]}\,) \;\land\; \hat{y} = \widehat{q[*]} \;\land\; \hat{p} \sqsubseteq \hat{q} \\
\lor \\
(\hat{x} = \widehat{p.f} \;\land\; \hat{y} = \widehat{q.f} \;\land\; \hat{p} \sqsubseteq \hat{q}) \\
\lor \\
(\hat{x} = \widehat{p[c]} \;\land\; \hat{y} = \widehat{q[c]} \;\land\; \hat{p} \sqsubseteq \hat{q})
\end{gathered}
$$

Intuitively, $\widehat{o_1} \sqsubseteq \widehat{o_2}$ holds when $\widehat{o_1}$ can be turned to $\widehat{o_2}$ by substituting any of its constant array indices with “$*$”. Figure 2.9 gives an example of such ordering. The direction of the edges is from the less to the more general object.

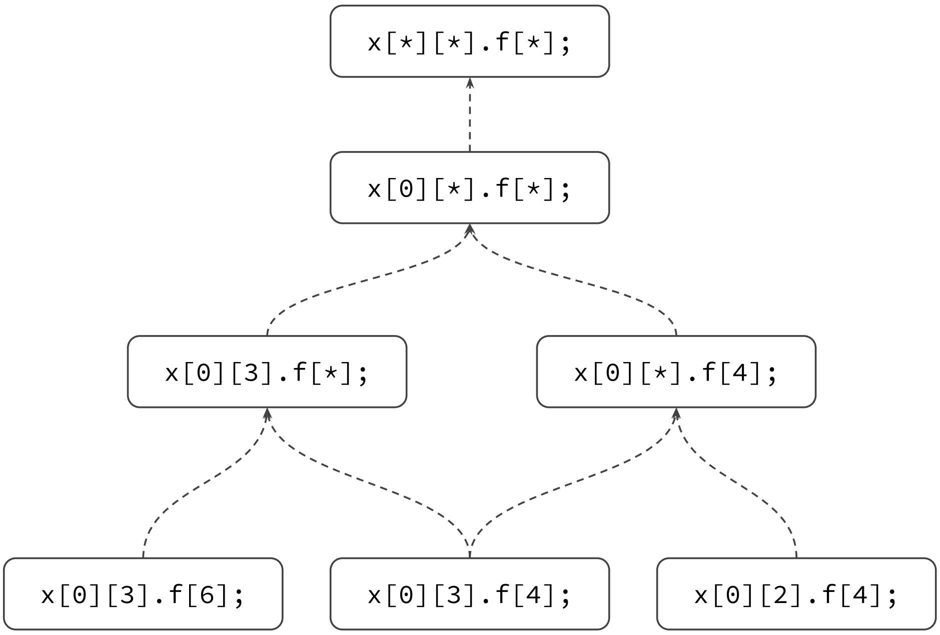


Figure 2.9: Abstract Object Ordering – Example: Nodes are abstract objects. An edge $(\,\hat{s}, \hat{t}\,)$ denotes that object $\hat{s}$ is generalized by object $\hat{t}$ (i.e., $\hat{s} \sqsubseteq \hat{t}$).

Given this partial order, it suffices to add the two rules of Figure 2.10 to account for possible index aliases. The first rule states that the points-to set of a (less general) object, such as $\widehat{o[c]}$, is a superset of the points-to set of any object that generalizes it, such as $\widehat{o[*]}$. The second rule modifies the treatment of load instructions, so that they may return anything in the points-to set of not just the object we load from (such as $\hat{o}[*]$), but also of objects that it generalizes (such as $\widehat{o[c]}$). In this way, the general and specific points-to sets are kept distinct, while their subset relationship is maintained.

$$\text{Match}\ \frac{\widehat{o_1} \sqsubseteq \widehat{o_2} \qquad \widehat{o_2} \rightsquigarrow \hat{o}}{\widehat{o_1} \rightsquigarrow \hat{o}} \qquad \text{Load II}\ \frac{i :\ \mathtt{p} = *\mathtt{q} \qquad \mathtt{q} \mapsto \widehat{o_2} \qquad \widehat{o_1} \sqsubseteq \widehat{o_2} \qquad \widehat{o_1} \rightsquigarrow \hat{o}}{\mathtt{p} \mapsto \hat{o}}$$

Figure 2.10: Associating array subobjects via their partial order.

2.3.4 Soundness

As stated by Avots et al. [10]: "*A C pointer alias analysis cannot be strictly sound, or else it would conclude that most locations in memory may point to any memory location.*" As in the PCP points-to analysis [10], our approach tries to maintain precision at all times, even if this means that the analysis is not sound in some cases. Instead of trying to be as conservative as possible, we choose to opt for precision and increase soundness by selectively supporting well-established code patterns or idioms (such as using `malloc()` to allocate many objects of different types).

The soundness assumptions of our analysis are that: (i) objects are allocated in separate memory spaces [10], and (ii) every (concrete) object has a single type throughout its lifetime. Hence, our analysis would be unsound when a union type is used to modify the same concrete object using two different types, since this violates the second assumption. However, our analysis would be a good fit for programs that use *discriminated unions* (e.g., unions that depend on a separate *tag* field to determine the exact type of the object), since it would create a different abstract object for every type of the union, so that each such abstract object would represent the subset of concrete objects with the same tag value.

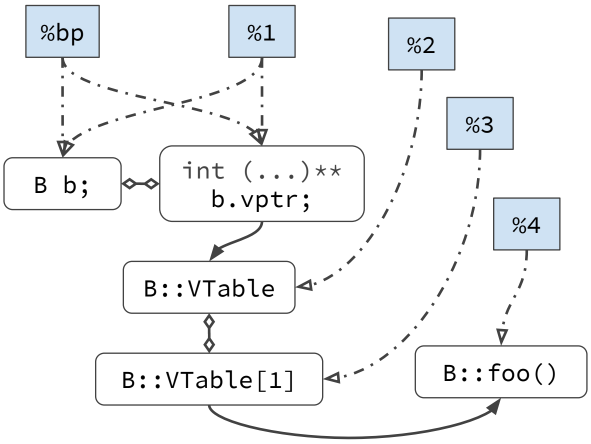
In general, the single-type-per-lifetime assumption is reasonable for most objects, but would be prohibitive in some cases—especially so when the code relies on low-level assumptions about the byte layout of the objects. For instance, our base approach would not be able to meaningfully analyze code that uses a custom memory allocator. Instead, the analysis would need to be extended so that it models calls to the allocator by creating new abstract objects.

Finally, the analysis must be able to discover all associated types for any given object, to retain its soundness. For simplicity, we have only considered cast instructions as places where the analysis discovers new types, but it is easy to supply additional type hints by considering more candidates. For instance, an exception object of unknown type may be allocated and then thrown, by calling the `cxa::throw()` function in the C++ exception handling ABI, without any intervening cast. However, we can use the accompanying *typeinfo* object (always supplied as the second argument to `cxa::throw()`) to recover its true type and hence create a typed abstract exception object. To the best of our knowledge, such special treatment is needed only in rare cases, and the analysis can be easily extended to handle them.

2.4 Analyzing C++

LLVM bitcode is a representation well-suited for C. However, for OO languages such as C++, high-level features are translated to low-level constructs. A classic example is dynamic dispatch, through virtual methods. Virtual-tables are represented as constant arrays of function pointers, and virtual calls are, in turn, translated to a series of indirect access instructions.

```
%class.B = type { i32 (...)**, ...}

;; translation of bp->foo(), for
;; B *bp;
%1 = bitcast %bp to i32 (%class.B*)***
%2 = load i32 (%class.B*)** %1
%3 = getelementptr i32 (%class.B*)** %2, 1
%4 = load i32 (%class.B*)* %3
call i32 %4 (%class.B* %bp)
```



(a) C++ virtual call compiled to LLVM bitcode **(b) Points-to graph**

Figure 2.11: C++ Virtual Call Example

Figure 2.11a presents (a simplified version of) the LLVM bitcode for such a translation. A virtual call has to (i) load the v-pointer of the class instance (at offset 0), (ii) index into the returned v-table (at the corresponding offset of the function being called), (iii) then load the returned function pointer to get the exact address of the function, and (iv) finally call the function. By employing the techniques we have described so far, our structure-sensitive analysis is well-equipped to deal with such an involved pattern, and precisely resolve the function to be called.

Figure 2.11b shows what our analysis computes (assuming %bp points to variable b). Only a minor addition is required: anything that points to an object should also point to its first field (at byte offset 0). Hence, both %bp and %1 (after the cast) will point both to (stack-allocated) object $\widehat{b}$, and to its v-pointer field subobject $\widehat{b.vptr}$. The first load instruction will return the v-table. Indexing into the v-table will return the corresponding array element subobject, which will maintain its own independent (singleton) points-to set, due to array-sensitivity. Finally, the second load instruction will return the exact function that the v-table points to, at the given offset.

2.5 Enhancements

Section 2.3 presented only the essential parts of a structure-sensitive points-to analysis, but there can be many enhancements worth discussing at this point. In this section, we will list a few of the most valuable ones for a practical implementation.

2.5.1 Pointer Arithmetic

In C/C++, given a pointer "`P *ptr`", expressions such as (i) "`(*ptr).fld`", (ii) "`ptr->fld`", and (iii) "`ptr[0].fld`" are equivalent (and will be translated to the same LLVM bitcode instruction). Hence, a pointer analysis must be able to reason about such code patterns and map them to the same abstract (sub)object. In LLVM IR, all such field accesses would be canonicalized to the third form. In fact, all pointers in LLVM bitcode are treated as pointers to arrays of objects (even when they point to a single one).[2] Regarding the possible choices of analysis inputs (discussed in Chapter 1), such translations are certainly a point in favor of choosing an intermediate representation such as LLVM bitcode, so that the analysis need not concern itself with superfluous syntactic constructs.

However, we have not so far delved into the various pointer arithmetic idioms that are supported by C/C++ and that we would like for our structure-sensitive analysis to support as well. A pointer in C is a memory address, which is a numeric value. One can perform arithmetic operations on a pointer just as with other numeric values. For instance, the expression "`ptr+3`" is equivalent to "`&ptr[3]`" (which corresponds to the aforementioned 3rd form, as per LLVM's canonicalization). Again LLVM's translation can be very helpful in narrowing the complexity of the input language. To see how our analysis can be extended to support pointer arithmetic, we first have to briefly present LLVM's GEP instruction, since this is what all types of element accesses (and pointer arithmetic operations) are normally translated to.

The GEP Instruction. The actual LLVM bitcode instruction that is responsible for all types of element accesses is `getelementptr` (GEP). The GEP instruction accepts a base pointer argument and one or more index arguments. It can be used to retrieve any inner element of an arbitrarily nested structure. Its general form is:

```
%ptr = getelementptr %base, %idx1, %idx2, %idx3, ...
```

Assuming that `%base` may point to an array of objects (in the general case), the first index selects an element of this array. (Expressions such as "`ptr->fld`"—being equivalent to "`ptr[0].fld`"—implicitly refer to the first element and thus produce a zero first index.) Any index after the first corresponds to a field or array element access. Even though the

[2] This is exactly the reason for the first (frequently zero) index of the often misunderstood `getelementptr` LLVM bitcode instruction (see `http://llvm.org/docs/GetElementPtr.html`).

generic form of GEP may contain multiple indices to be able to return the address of deeply nested elements, in practice, such complex GEP instructions are split into multiple chained GEPs with at most 2 indices. Each one will descend in a single field or array index, roughly corresponding to the *address-of-field* and *address-of-array-index* instructions that we have previously shown. Returning the address corresponding to a complex access path such as "`ptr->f[4].g[k]`" will require as many GEPs as the depth of the access path: two field and two array accesses, four in total. Note that GEP instructions do not perform any memory dereference (as the `load` instruction does), but only compute a relative offset.

So far we have not examined GEP instructions whose first index is non-zero (thus retrieving the address of an element other than the first, given a base pointer array). Fortunately, we can further decompose such instructions to a GEP instruction with a single non-zero index and a second GEP instruction of 2 indices, whose first index is always zero.

```
%ptr = getelementptr %b, 7, %v1, 8
```

(a) Complex GEP instruction

```
%t1 = getelementptr %b, 7
%t2 = getelementptr %t1, 0, %v1
%ptr = getelementptr %t2, 0, 8
```

(b) Decomposed GEP instruction

Figure 2.12: Decomposition of GEP instructions

Figure 2.12 demonstrates such a decomposition that breaks a complex GEP instruction into multiple simpler ones. The last two instructions of the decomposed version of Figure 2.12b correspond to our familiar notions of *address-of-array-index* and *address-of-field* instructions. Thus, we can dispense with GEP's overloaded behavior and keep our inference rules almost intact. However, we have to augment the analysis to support a single-index GEP, just like the first instruction of Figure 2.12b. Essentially, such instructions perform a pointer increment operation and are of special interest, since they alone should suffice for most translations of dynamically allocated arrays and pointer arithmetic operations that we are interested in.

Figure 2.13 presents an extension of our analysis to handle such pointer increment operations. The first two rules replace the previous ones (for base allocations) by slightly changing the objects we allocate to resemble the first elements of potential array allocations. In cases where an instruction allocates a single object, such zero offsets may be redundant but facilitate the increment operations of the following rules. The next two rules handle pointer increment operations that use a variable index `j`. In both cases, the result should disregard any prior index of the object pointed by `q`, and replace it with the "$*$" index. The last three rules handle pointer increment operations that add a constant c. The first of them states that adding a constant index to an already unknown index makes no difference and simply propagates the same special object ($\widehat{o[*]}$). In the last two rules, `q` points to an object with a constant index instead ($\widehat{o[k]}$), whereupon the analysis tries to compute a new constant index $k+c$ and point to the relevant object ($o\widehat{[k+c]}$). However, such general treatment could potentially lead to the creation of infinite objects. To avoid this, we only point to $o\widehat{[k+c]}$ if the new index $k+c$ statically appears in the source code

$$\text{Stack}\quad \frac{i:\ \texttt{p} = \texttt{alloca}\ \texttt{T},\ nbytes}{\texttt{p} \mapsto \widehat{o_i[0]} \qquad \textit{type}(\widehat{o_i[0]}) = \texttt{T}} \qquad\qquad \text{Heap}\quad \frac{i:\ \texttt{p} = \texttt{malloc}\ nbytes}{\texttt{p} \mapsto \widehat{o_i[0]}}$$

$$\text{Pointer – Var I}\quad \frac{i:\ \texttt{p} = \texttt{q} + \texttt{j} \qquad \textit{type}(q) = \texttt{T*} \qquad \texttt{q} \mapsto \widehat{o[k]} \qquad \textit{type}(\widehat{o[k]}) = \texttt{T}}{\texttt{p} \mapsto \widehat{o[*]}}$$

$$\text{Pointer – Var II}\quad \frac{i:\ \texttt{p} = \texttt{q} + \texttt{j} \qquad \textit{type}(q) = \texttt{T*} \qquad \texttt{q} \mapsto \widehat{o[*]} \qquad \textit{type}(\widehat{o[*]}) = \texttt{T}}{\texttt{p} \mapsto \widehat{o[*]}}$$

$$\text{Pointer – Const I}\quad \frac{i:\ \texttt{p} = \texttt{q} + c \qquad \textit{type}(q) = \texttt{T*} \qquad \texttt{q} \mapsto \widehat{o[*]} \qquad \textit{type}(\widehat{o[*]}) = \texttt{T}}{\texttt{p} \mapsto \widehat{o[*]}}$$

$$\text{Pointer – Const II}\quad \frac{\begin{array}{c} i:\ \texttt{p} = \texttt{q} + c \qquad \textit{type}(q) = \texttt{T*} \qquad \texttt{q} \mapsto \widehat{o[k]} \qquad \textit{type}(\widehat{o[k]}) = \texttt{T} \\ k + c \in \textit{indices}(\texttt{T}) \end{array}}{\texttt{p} \mapsto \widehat{o[k+c]}}$$

$$\text{Pointer – Const III}\quad \frac{\begin{array}{c} i:\ \texttt{p} = \texttt{q} + c \qquad \textit{type}(q) = \texttt{T*} \qquad \texttt{q} \mapsto \widehat{o[k]} \qquad \textit{type}(\widehat{o[k]}) = \texttt{T} \\ k + c \notin \textit{indices}(\texttt{T}) \end{array}}{\texttt{p} \mapsto \widehat{o[*]}}$$

Figure 2.13: Dealing with pointer arithmetic

(and is associated with type T). Otherwise, the analysis falls back to using $\widehat{o[*]}$.

To identify what static indices are associated with each type we define the function *indices* as follows:

$$\begin{aligned} \textit{indices}(\texttt{T}) = \{0\} &\cup \{c : \exists\ \texttt{p} = \texttt{q} + c\ \wedge\ \textit{type}(q) = \texttt{T*}\} \\ &\cup \{c : \exists\ \texttt{p} = \texttt{\&q}[c]\ \wedge\ \textit{type}(q) = [\texttt{T}]\} \end{aligned}$$

2.5.2 Abstract Object Aliases

There are cases where two abstract objects, such as $\hat{o}$ and $\widehat{o[0]}$ may coincide (i.e., map to the same memory address). This applies to both zero indices of arrays and the first fields of structs. Ignoring such object aliases could lead to unsound results, e.g., when dereferencing an object without taking the points-to sets of its aliases into account.

To handle such cases, the analysis should treat aliased abstract objects as an equivalence class.

Definition 2.2. *Abstract Object Aliases.* We define the *alias* equivalence relation $\sim$ on the set of abstract objects O as the transitive, symmetric, and reflexive closure of the binary relation , so that given two abstract objects $\hat{x}$, $\hat{y} \in O$, $\hat{x}$ $\hat{y}$, if and only if:

$$\hat{y} = \widehat{x[0]}$$
$$\vee$$
$$\hat{y} = \widehat{x.f} \wedge \textit{offsetof}(f) = 0$$

where *offsetof*(f) returns the byte offset of a struct field f.

$$\text{Deref}^{+} \; \frac{\widehat{o_1} \sim \widehat{o_2} \qquad \hat{o} \rightsquigarrow \widehat{o_1}}{\hat{o} \rightsquigarrow \widehat{o_2}} \qquad\qquad \text{InvDeref}^{+} \; \frac{\widehat{o_1} \sim \widehat{o_2} \qquad \widehat{o_1} \rightsquigarrow \hat{o}}{\widehat{o_2} \rightsquigarrow \hat{o}}$$

$$\text{Vpt}^{+} \; \frac{\widehat{o_1} \sim \widehat{o_2} \qquad \mathtt{v} \mapsto \widehat{o_1}}{\mathtt{v} \mapsto \widehat{o_2}}$$

Figure 2.14: Extending the analysis with aliased objects.

Figure 2.14 extends the dereference and variable points-to edges computed by the analysis due to object aliases. The first rule states that whenever an object points to another, it should also point to any aliases of the latter. The second rule inverses this notion: when an object is pointed by another, it should also be pointed by its aliases. The third rule is analogous to the first one, but extends variable points-to edges instead.

2.5.3 Type Compatibility

We have used type equality as a filter for redundant derivations in the inference rules we have presented so far. In practice, however, this could prove too restrictive and lead to the exclusion of perfectly valid derivations. For instance, consider the trailing padding that C compilers append to struct types for proper alignment. The structure type of Figure 2.15a would be transformed to that of Figure 2.15b, as most compilers would append a padding field of 7 bytes so that the total size becomes 16—a multiple of the largest alignment of any of its members (in this case, a multiple of 8 due to field p).

```
struct s1 {
  char *p;      /* 8 bytes */
  char c;       /* 1 byte */
};
```

(a) Unpadded Structure Type

```
struct s1 {
  char *p;      /* 8 bytes */
  char c;       /* 1 byte */
  char pad[7];
};
```

(b) Padded Structure Type

Figure 2.15: Structure Alignment and Padding

However, had a padded struct (or class) type been inherited by another, then the compiler could in some cases use the unpadded version as the base type, since the padding could be redundant if new fields were considered.[3] If strict type equality was required by the analysis, then this could prohibit objects of a derived type to be used as the receiver arguments of inherited methods at places where the analysis employed its type filters (such as in *address-of-field* instructions).

```cpp
struct s1 {
  char *p;
  char c;

  void meth() {
    char *c1 = this->p;
    ...
  }
};

struct s2 : s1 {
  char b;
};
```

(a) C++ Source

```llvm
; Types

%struct.s1 = type { i8*, i8, [7 x i8] }
%struct.s1.base = type { i8*, i8 }
%struct.s2 = type { %struct.s1.base, i8, [6 x i8] }

; Methods

define void @s1_meth(%struct.s1* %this) {
  ; the second 0 offset is that of field p
  %1 = getelementptr %struct.s1* %this, i64 0, i32 0
  ...
}
```

(b) LLVM Bitcode

Figure 2.16: Padding, Inheritance, and Type Incompatibility

Figure 2.16 illustrates this case. The generated LLVM bitcode uses the unpadded version of `s1` as its first field (inherited objects are always translated to normal fields, in bitcode), to reduce its overall padding to just 6 bytes. Had an object $\hat{o}$ of type `s2` flowed to the points-to set of `s1::meth()`'s `this` argument, it would be filtered out in the `getelementptr` instruction due to type inequality of `s1` and `s2`, per our Field inference rule of Section 2.3.

Note that, had the `%struct.s1.base` type not existed for padding reasons, our object alias rules of Figure 2.14 alone would suffice in this case:

- the $\widehat{o.f_{s1}}$ subobject, representing the first field of $\hat{o}$ (i.e., its `s1` base object), would be considered an alias of $\hat{o}$, since its byte offset is zero
- `%this` would, hence, also point to $\widehat{o.f_{s1}}$ besides $\hat{o}$
- `getelementptr` would not filter out $\widehat{o.f_{s1}}$, since its type would be equal to the expected declared type `s1`
- `%1` would finally point to the $\widehat{o.f_{s1}.p}$ subobject.

Such reasoning can be encoded in the following complex derivation:

$$\text{Field}\ \dfrac{\text{Vpt}^{+}\ \dfrac{\widehat{o.f_{s1}} \sim \hat{o} \qquad \texttt{this} \mapsto \hat{o}}{\texttt{this} \mapsto \widehat{o.f_{s1}}} \qquad 11:\ \texttt{\%1} = \texttt{\&this->}p \qquad \mathit{type}(\widehat{o.f_{s1}}) = \mathit{type}(\texttt{this}) = \texttt{s1}}{\texttt{\%1} \mapsto \widehat{o.f_{s1}.p} \qquad \mathit{type}(\widehat{o.f_{s1}.p}) = \texttt{char}\,*}$$

[3] In LLVM IR, the name of the unpadded version of the type would contain a `.base` suffix to distinguish it from the original padded version.

We can relax our type equality constraints by introducing a notion of *type compatibility*. We list the following cases of compatible types:

- a type is type compatible with itself (making the relation reflexive)
- an array type `T[]` is type compatible with another array type `U[]`, if (i) its component type `T` is type compatible with `U`, and (ii) they are either of the same size (e.g., `T[5]`, `U[5]`), or at least one of them does not specify a size (e.g., `T[7]`, `U[]`)
- a function type $\mathtt{R}(\mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_3, \ldots, \mathtt{T}_n)$ is type compatible with function type $\mathtt{S}(\mathtt{U}_1, \mathtt{U}_2, \mathtt{U}_3, \ldots, \mathtt{T}_n)$ if (i) their return types `R` and `S` are type compatible, and (ii) so are their arguments types (i.e., $\mathtt{T}_i$ is type compatible with $\mathtt{U}_i$ for every $i \in 1, \ldots, n$)
- a function type $\mathtt{R}(\mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_3, \ldots, \mathtt{T}_n)$ is also type compatible with the variadic function type $\mathtt{S}(\mathtt{U}_1, \mathtt{U}_2, \mathtt{U}_3, \ldots, \mathtt{U}_m, \ldots)$ if (i) $m < n$, (ii) their return types `R` and `S` are type compatible, and (iii) so is the common subset of their arguments types (i.e., $\mathtt{T}_i$ is type compatible with $\mathtt{U}_i$ for every $i \in 1, \ldots, m$)
- a struct type $\mathtt{S}_1$ (consisting of fields $\mathtt{F}_1, \mathtt{F}_2, \mathtt{F}_3, \ldots, \mathtt{F}_n$, in that order), is type compatible *up to field* k with struct type $\mathtt{S}_2$ (consisting of fields $\mathtt{Q}_1, \mathtt{Q}_2, \mathtt{Q}_3, \ldots, \mathtt{Q}_m$), if field type $\mathtt{F}_i$ is type compatible with $\mathtt{Q}_i$ for every $i \in 1, \ldots, k$; moreover, $\mathtt{S}_1$ and $\mathtt{S}_2$ are type compatible, if they are type compatible up to field m and m equals n
- a struct type $\mathtt{S}_1$ is an *eligible base* of type $\mathtt{S}_2$, if the first field of $\mathtt{S}_2$, $\mathtt{F}_1$, is type compatible with $\mathtt{S}_1$ up to field n—where n is the number of fields of $\mathtt{F}_1$
- a pointer type `T*` is type compatible with another pointer type `U*`, if
  1. its component type `T` is type compatible with `U`,
  2. either `T` or `U` is `char`, or
  3. `U` is (transitively) an *eligible base* of `T`.

Our type compatibility rules are deliberately geared towards structural compatibility (as is our overall structure-sensitive approach). Restricting the type compatibility rules to comply with some specific C/C++ standard would not work, since by the time the compiler has transformed the code to LLVM bitcode, the various transformations and optimizations up to this point would almost certainly violate such rules.

Henceforth, we will use the notation “*typecompat*(`T`, `U`)” to signify that type `T` is type compatible with `U`, according to the previous rules. Even though we will forgo rewriting any previous inference rules of our analysis to make use of the type compatibility relation, the reader should assume that any strict type equality premise clauses therein, such as “$\mathit{type}(\hat{o}) = \mathtt{T}$”, should be replaced with the more generic “$\mathit{typecompat}(\mathit{type}(\hat{o}), \mathtt{T})$”.

As a final note, every struct type can be viewed as an array of bytes (as hinted by our type compatibility rules). A field could be accessed in such a way, via its byte offset. Figure 2.17 identifies such accesses and treats them accordingly.

$$\text{Byte Offset}\quad \frac{\begin{array}{c} i:\ \texttt{p} = \texttt{q} + c \qquad \textit{type}(q) = \texttt{char}\,* \qquad \texttt{q} \mapsto \hat{o} \qquad \textit{type}(\hat{o}) = \texttt{T} \\ \textit{offsetof}(\texttt{T}.f) = c \end{array}}{\texttt{p} \mapsto \widehat{o.f}}$$

Figure 2.17: Accessing field via byte offset.

2.5.4 Copying Memory Areas

There are various functions in C that copy memory from a pointer to another (e.g., `memcpy()`, `memmove()`, `bcopy()`, etc). Such operations have an obvious effect on the points-to sets of objects: an object that is copied to another location should have its points-to set copied as well.

$$\text{Memcpy Base}\quad \frac{\begin{array}{c} i:\ \texttt{p} = a_0(a_1,\, a_2,\, \ldots) \\ i \xrightarrow{calls} \texttt{memcpy}\,(\ldots) \qquad a_1 \mapsto \widehat{o_{to}} \qquad a_2 \mapsto \widehat{o_{from}} \\ \textit{type}(\widehat{o_{from}}) = \texttt{T} \qquad \textit{type}(\widehat{o_{to}}) = \texttt{U} \qquad \textit{typecompat}(\texttt{T},\texttt{U}) \end{array}}{\textit{copy}\ (\widehat{o_{from}},\, \widehat{o_{to}})}$$

$$\text{Memcpy Rec I}\quad \frac{\begin{array}{c} \textit{copy}\ (\widehat{o_1},\, \widehat{o_2}) \\ \textit{type}(\widehat{o_1[*]}) = \texttt{T} \qquad \textit{type}(\widehat{o_2[*]}) = \texttt{U} \qquad \textit{typecompat}(\texttt{T},\texttt{U}) \end{array}}{\textit{copy}\ (\widehat{o_1[*]},\, \widehat{o_2[*]})}$$

$$\text{Memcpy Rec II}\quad \frac{\begin{array}{c} \textit{copy}\ (\widehat{o_1},\, \widehat{o_2}) \\ \textit{type}(\widehat{o_1[c]}) = \texttt{T} \qquad \textit{type}(\widehat{o_2[c]}) = \texttt{U} \qquad \textit{typecompat}(\texttt{T},\texttt{U}) \end{array}}{\textit{copy}\ (\widehat{o_1[c]},\, \widehat{o_2[c]})}$$

$$\text{Memcpy Rec III}\quad \frac{\begin{array}{c} \textit{copy}\ (\widehat{o_1},\, \widehat{o_2}) \\ \textit{type}(\widehat{o_1.f}) = \texttt{T} \qquad \textit{type}(\widehat{o_2.f}) = \texttt{U} \qquad \textit{typecompat}(\texttt{T},\texttt{U}) \end{array}}{\textit{copy}\ (\widehat{o_1.f},\, \widehat{o_2.f})}$$

$$\text{Memcpy Deref}^{+}\quad \frac{\textit{copy}\ (\widehat{o_{from}},\, \widehat{o_{to}}) \qquad \widehat{o_{from}} \rightsquigarrow \hat{o}}{\widehat{o_{to}} \rightsquigarrow \hat{o}}$$

Figure 2.18: Handling memory copying.

To support memory copying, we first identify copied objects. The first rule of Figure 2.18 marks that an abstract object $\widehat{o_{from}}$ was copied to another object $\widehat{o_{to}}$, if a (direct or indirect) call to the `memcpy()` routine was made and these were the objects pointed by `memcpy()`'s operands. (Note that we use the objects pointed by the actual and not the formal arguments, to avoid associating objects of unrelated `memcpy` calls. This could also be achieved with context-sensitivity based on call-sites.) Additionally, we require that the objects are of compatible types, to filter imprecision. The next three rules of Figure 2.18 extend our notion of copied objects recursively, by marking subobjects as well (as long as they remain type compatible with each other). The last rule performs the propagation of the points-to set of an object that was copied to another.

2.6 Evaluation

We compare our structure-sensitive analysis to a re-implementation of the Pearce et al. [104, 105] analysis in `cclyzer`, that also operates over the full LLVM bitcode language. We will refer to this analysis as *Pearce*^{c}. Both analyses were implemented using Datalog, and include the enhancements of Section 2.5 and a few more to deal with various features (hidden copies of struct instances due to pass-by-value semantics, constant expressions, etc.) that arise in practice.

For our benchmark suite, we use the 8 largest programs (in terms of bitcode size) in *GNU Coreutils*,[4] and 14 executables from *PostgreSQL*. We use a 64-bit machine with two octa-core Intel Xeon E5-2667 (v2) CPUs at 3.30GHz and 256GB of RAM. The analysis is single-threaded and occupies a small portion of the RAM. We use the LogicBlox Datalog engine (v.3.10.14) and LLVM v.3.7.0.

Figure 2.19 presents some general metrics on the input and output of each analysis: (i) number of call-graph edges (allocation site to function), (ii) number of abstract objects created by the analysis, and (iii) running time (excluding constant overhead that bootstrap both analyses).

Figure 2.20 compares the two analyses in terms of the degree of resolving variable points-to targets. The first column of each analysis lists the percentage of fully resolved variables (virtual registers): *how many point to a single abstract object*. This is the main metric of interest for most analysis clients. The next two columns list the percentage of variables that point to two/three objects.

It is evident that our structure-sensitive analysis fares consistently better in fully resolving variable targets. Our analysis resolves many more variables than *Pearce*^{c} does, for any of the available benchmarks, with an average increase of 36% across all coreutil benchmarks and 58% in the PostgreSQL benchmarks. This is *despite using a finer-grained object abstraction than* Pearce^c: The “abstract objects” column of Figure 2.19 shows that our analysis abstraction has *one to two orders of magnitude more* abstract objects than *Pearce*^{c}. Yet it succeeds at resolving many more variables to a single (and much finer-grained) abstract object. (The only benchmark instance in which *Pearce*^{c} somewhat benefits from its coarse abstract object granularity is ecpg: a full 42.64% of variables point to 3, much coarser than ours, abstract objects.) Note also that the *Pearce*^{c} analysis appears much better than it actually is for meaningful cases, due to large amounts of low-hanging fruit—e.g., global or address-taken variables, which are the single target of some virtual

[4] Our original selection included the 10 largest coreutils, but `dir` and `vdir` turned out to be identical to `ls` and are maintained mostly for backwards-compatibility reasons.

			Structure-sensitive		*Pearce^c*	
Benchmark	Size	call-graph edges	abstract objects	running time	abstract objects	running time
cp	720K	3205	68166	29.25s	3380	13.62s
df	456K	1812	38919	20.68s	2236	11.09s
du	608K	2424	49592	29.77s	3008	21.96s
ginstall	692K	3185	59893	25.12s	3207	14.32s
ls	604K	2654	66469	22.43s	2783	13.35s
mkdir	384K	1466	21900	17.35s	1641	11.43s
mv	648K	2932	55619	25.50s	3015	12.20s
sort	608K	2480	75360	34.25s	2955	21.40s
clusterdb	528K	1390	167605	33.90s	4461	11.89s
createdb	528K	1412	168068	30.58s	4480	11.07s
createlang	572K	1928	133869	25.67s	4275	12.68s
createuser	532K	1435	171115	31.07s	4569	9.31s
dropdb	524K	1361	165966	31.26s	4399	12.72s
droplang	572K	1936	133912	24.38s	4278	12.55s
dropuser	524K	1356	165615	30.45s	4386	12.15s
ecpg	1.2M	5713	59252	38.47s	5219	29.11s
pg-ctl	488K	1615	118689	23.36s	3655	9.14s
pg-dumpall	572K	2110	184276	32.18s	5153	11.95s
pg-isready	464K	1302	108622	21.54s	3343	11.25s
pg-rewind	556K	1943	136915	25.56s	4301	11.48s
pg-upgrade	604K	2501	151967	26.49s	4965	11.80s
psql	1.4M	5925	460522	67.76s	14025	25.28s

Figure 2.19: Input and Output Metrics. The first column is benchmark bitcode size (in bytes). The second column is the number of call-graph edges (as computed by our analysis). The third (resp. fifth) column is the number of abstract objects created. The fourth (resp. sixth) column is the analysis running time.

register, due to the SSA representation.

2.7 Summary

We began this chapter by introducing the needs for a revised abstract memory model for points-to analysis when analyzing C/C++, which can fully support field sensitivity and maintain maximal structural information for its abstract objects. We accomplish this by increasing object granularity to force the creation of typed objects and by fully distinguishing subobjects as well. We give a brief overview of our techniques in Section 2.1, and present some limited background about the peculiarities of C/C++ and LLVM IR in Section 2.2, regarding the complications they pose to the problem of pointer analysis. We describe our structure-sensitive points-to analysis in depth in Section 2.3, and discuss how the techniques it employs make it suitable for analyzing C++ programs in Section 2.4. In Section 2.5 we present various enhancements, essential for a realistic implementation. Finally, we evaluate our approach by comparing it to a standard field-sensitive analysis in Section 2.6.

	Structure-sensitive			*Pearce*^{c}		
Benchmark	(%) $\lvert pt(v)\rvert \to 1$	2	3	(%) $\lvert pt(v)\rvert \to 1$	2	3
cp	35.42	11.56	9.03	24.02	2.91	3.51
df	35.98	13.15	8.37	26.28	1.98	4.38
du	37.06	10.51	7.54	25.60	2.00	2.95
ginstall	36.31	14.24	8.28	27.15	7.44	3.14
ls	33.23	6.09	8.81	26.90	3.57	2.67
mkdir	36.11	8.43	9.65	23.02	2.00	4.35
mv	35.09	13.71	8.97	24.58	6.78	3.04
sort	29.20	5.25	9.65	22.37	1.47	2.53
average	34.49	9.51	8.79	25.37	3.53	3.19
clusterdb	40.86	8.42	7.93	24.46	2.79	3.85
createdb	40.82	9.11	7.95	24.54	2.83	4.31
createlang	42.72	8.87	11.89	25.62	4.10	4.78
createuser	40.33	8.85	8.75	24.07	3.18	4.44
dropdb	40.59	8.69	7.96	23.97	2.91	4.00
droplang	42.68	8.86	11.88	25.67	4.10	4.75
dropuser	40.36	8.72	8.01	23.86	2.86	4.02
ecpg	16.72	1.22	0.52	15.14	0.30	42.64
pg-ctl	41.31	8.46	8.50	25.31	3.31	4.05
pg-dumpall	40.52	7.10	7.21	27.74	3.10	4.61
pg-isready	39.89	8.12	7.87	23.59	2.92	4.03
pg-rewind	44.74	7.55	8.56	31.39	2.75	3.76
pg-upgrade	41.12	8.35	9.34	27.73	2.95	3.70
psql	38.62	5.81	9.33	25.61	2.31	3.20
average	39.38	7.72	4.55	24.91	2.89	6.87

Figure 2.20: Variable points-to sets. Proportion of resolved variables (that point to one abstract object), as well as variables with two or three points-to targets.

3. MORE SOUND STATIC HANDLING OF JAVA REFLECTION

> There are ways, Dude. You don't wanna know about it, believe me.
>
> *Walter Sobchak*

In Chapter 2, we targeted the problem of lost structural information in C/C++ programs by employing a pointer analysis that recovers lost memory structure via a variety of techniques. In this chapter and the next, we shift our focus to Java: a higher-level, strongly-typed language with no capabilities for direct memory access. Still, essential structural information is often lost in Java programs too, yet for different reasons. As stated in Chapter 1, a source of analysis imprecision, especially in determining the types of abstract objects constructed by the analysis, lies in the use of Java's reflection mechanism: the ability to inspect and dynamically retrieve classes, methods, attributes, etc. at runtime.

By using the Reflection API, Java programs can encompass dynamic behavior. However, statically reasoning about the behavior of software that uses reflection can be especially cumbersome. Unfortunately, reflection is ubiquitous in large Java programs. Any handling of reflection will be approximate, and overestimating its reach in a large codebase can be catastrophic for precision and scalability. In this chapter, we present an approach for handling reflection with improved empirical soundness (as measured against prior approaches and dynamic information), again, in the context of a points-to analysis. Our approach is based on the combination of string-flow and points-to analysis from past literature augmented with (a) substring analysis and modeling of partial string flow through string builder classes; (b) new techniques for analyzing reflective entities based on information available at their use-sites (similar to those presented in Chapter 2). In experimental comparisons with prior approaches, we demonstrate a combination of both improved soundness (recovering the majority of missing call-graph edges) and increased performance.

3.1 Intro: Static Analysis and Java Reflection

Whole-program static analysis is the engine behind several modern programming facilities for program development and understanding. Compilers, bug detectors, security checkers, modern development environments (with automated refactorings, slicing facilities, and auto-complete functionality), and a myriad other tools routinely employ static analysis machinery. Even the seemingly simple effort of computing a program's call-graph (i.e., which program function can call which other) requires sophisticated analysis in order to achieve precision in a modern language.

Yet, static whole-program analysis suffers in the presence of common dynamic features, especially reflection. When a Java program accesses a class by supplying its name as a run-time string, via the `Class.forName` library call, the static analysis has very few available courses of action: It needs to either conservatively over-approximate (e.g., assume that *any* class can be accessed, possibly limiting the set later, after the returned object is used), or to perform a string analysis that will allow it to infer the contents of the `forName` string argument. Both options can be detrimental to the scalability of the analysis: the conservative over-approximation may never become constrained enough by further instructions to be feasible in practice; precise string analysis is impractical for programs of realistic size. It is telling that *no practical Java program analysis framework in existence handles reflection soundly* [86], although other language features are modeled soundly.[1]

Full soundness is not practically achievable, but it can still be approximated for the well-behaved reflection patterns encountered in regular, non-adversarial programs. Therefore, it makes sense to treat soundness as a continuous quantity: something to improve on, even though we cannot perfectly reach. To avoid confusion, we use the term *empirical soundness* for the quantification of how much of the dynamic behavior the static analysis covers. Computable metrics of empirical soundness can help quantify how close an analysis is to the fully sound result. Based on such metrics, one can make comparisons (e.g., "more sound") to describe soundness improvements.

The second challenge of handling reflection in a static analysis is *scalability*. The online documentation of the IBM Wala library [40] concisely summarizes the current state of the practice, for *points-to analysis* in the Java setting.

> *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.*

The same caveats routinely appear in the research literature. Multiple published points-to analysis papers analyze well-known benchmarks with reflection disabled [2, 3, 66, 123].

A representative quote [123] illustrates:

> *Hsqldb and jython could not be analyzed with reflection analysis enabled [...] — hsqldb cannot even be analyzed context-insensitively and jython cannot even be analyzed with the 1obj analysis. This is due to vast imprecision introduced when reflection methods are not filtered in any way by constant strings (for classes, fields, or methods) and the analysis infers a large number of reflection objects to flow to several variables. [...] For these two applications, our analysis has reflection reasoning disabled. Since hsqldb in the DaCapo benchmark code has its main functionality called via reflection, we had to configure its entry point manually.*

In this chapter, we describe an approach to analyzing reflection in the Java points-to analysis setting. Our approach requires no manual configuration and achieves significantly

[1] In our context, *sound* = over-approximate, i.e., guaranteeing that all possible behaviors of reflection operations are modeled.

higher empirical soundness without sacrificing scalability, for realistic benchmarks and libraries (DaCapo Bach and Java 7). In experimental comparisons with the recent Elf system [80] (itself improving over the reflection analysis of the Doop framework [20]), our algorithm discovers most of the call-graph edges missing (relative to a dynamic analysis) from Elf's reflection analysis. This improvement in empirical soundness is accompanied by *increased* performance relative to Elf, demonstrating that near-sound handling of reflection is often practically possible. Concretely, our work in this chapter:

· introduces key techniques in static reflection handling that contribute greatly to empirical soundness. The techniques generalize past work from an intra-procedural to an inter-procedural setting and combine it with a string analysis;

· shows how scalability can be addressed with appropriate tuning of the above generalized techniques;

· thoroughly quantifies the empirical soundness of a static points-to analysis, compared to past approaches and to a dynamic analysis;

· is implemented and evaluated on top of an existing open framework (Doop [20]).

3.2 Points-to Analysis in Java

In Chapter 2, we presented a points-to analysis for C/C++ that includes various enhancements to make it structure-sensitive. Before presenting our enhancements towards better handling of Java's reflection, we first present a typical points-to analysis for Java and discuss its fundamental differences from analyzing C/C++.

The domains of the analysis include:

· variables, V
· (class) types, T
· fields, F
· methods, M
· abstract (heap) objects, H
· instruction labels, L
· and strings.

Figure 3.1 lists the basic Java instructions, relevant to a points-to analysis. We note some key differences from the C/C++ setting:

– there are no pointer types, or any way to directly operate on memory addresses

– there is a clear distinction between variables (allocated on the stack) and objects (allocated on the heap)

– loads and stores need a field operand

Java Instruction	*Operand Types*	*Description*
`p = new C()`	$V \times T$	Heap Allocations
`p = (T) q`	$V \times T \times V$	Casts
`p = q`	$V \times V$	Assignments
`p = q.f`	$V \times V \times F$	Field Loads
`p.f = q`	$V \times F \times V$	Field Stores
`p = v.meth(...)`	$V \times V \times M \times V^n$	Virtual Calls
`p = C.meth(...)`	$V \times T \times M \times V^n$	Static Calls
`return p`	V	Method Returns

Figure 3.1: Java Instruction Set. We also prepend a label $l \in L$ to each instruction (that we omit in this figure). Each such label can be used to uniquely identify its instruction.

– there are two types of method call instructions: (i) virtual calls (that perform dynamic dispatch based on the dynamic type of the receiver), and (ii) static calls.

Figure 3.2 presents a standard Java points-to analysis [47, 66, 133], expressed in inference rules (such as those of Chapter 2). The main relations it computes are:

Variable points-to edges. Edge $\texttt{v} \mapsto \hat{o} \in V \times H$ records that variable `v` may point to abstract object $\hat{o}$.

Field points-to edges. Edge $\widehat{o_b} \xrightarrow{\texttt{fld}} \hat{o} \in H \times F \times H$ records that abstract object $\widehat{o_b}$ may point to abstract object $\hat{o}$, via field `fld`. Field points-to edges can only be established by field store instructions.

Abstract object types. The partial function *type* : $H \nrightarrow T$ records the type of an abstract object. Reflection aside, each abstract object will be associated with a single type that will be readily available at the allocation site.

Call-graph edges. Edge $i \xrightarrow{calls} m \in L \times M$ records that invocation site i may call method m (after the dynamic dispatch has been resolved).

Note that, without reflection, a points-to analysis needs only create a single kind of abstract objects that represents heap allocations based on the allocation site. There is no need for abstract subobjects, as in the C/C++ setting, since (i) heap objects can only contain references to other objects but cannot embed the actual allocations; complex structures need to be dispersed through the heap and accessed via multiple field loads (ii) a variable can only point to the start of a heap allocation. However, more types of abstract objects that serve our reflection-related enhancements will be introduced later.

The first rule of Figure 3.2 creates a typed abstract object that represents the given allocation site, and assigns it to the target variable. The next two rules, handling cast and move instructions, simply copy the points-to set of the source to that of the target variable. For

$$\text{Alloc}\ \frac{i:\ \texttt{p} = \texttt{new}\ \texttt{T}\,()}{\texttt{p} \mapsto \widehat{o_i} \qquad \textit{type}(\widehat{o_i}) = \texttt{T}}$$

$$\text{Cast}\ \frac{i:\ \texttt{p} = (\texttt{T})\,\texttt{q} \qquad \texttt{q} \mapsto \widehat{o_i} \qquad \textit{type}(\widehat{o_i}) = \texttt{T'} \qquad \texttt{T'<:T}}{\texttt{p} \mapsto \widehat{o_i}}$$

$$\text{Move}\ \frac{i:\ \texttt{p} = \texttt{q} \qquad \texttt{q} \mapsto \widehat{o_i}}{\texttt{p} \mapsto \widehat{o_i}}$$

$$\text{Load}\ \frac{i:\ \texttt{p} = \texttt{q}\,.\,\texttt{f} \qquad \texttt{q} \mapsto \widehat{o_b} \qquad \widehat{o_b} \xrightarrow{\texttt{f}} \hat{o}}{\texttt{p} \mapsto \hat{o}}$$

$$\text{Store}\ \frac{i:\ \texttt{p}\,.\,\texttt{f} = \texttt{q} \qquad \texttt{p} \mapsto \widehat{o_b} \qquad \texttt{q} \mapsto \hat{o}}{\widehat{o_b} \xrightarrow{\texttt{f}} \hat{o}}$$

$$\text{VCall}\ \frac{i:\ \texttt{p} = \texttt{v}\,.\,\texttt{meth}(\ldots) \qquad \texttt{v} \mapsto \hat{o} \qquad \textit{type}(\hat{o}) = \texttt{T} \qquad \textit{lookup}(\texttt{T},\ \texttt{meth}) = \texttt{meth}'}{i \xrightarrow{calls} \texttt{meth}'\,(\ldots) \qquad \texttt{this}_{\,\texttt{meth}'} \mapsto \hat{o}}$$

$$\text{SCall}\ \frac{i:\ \texttt{p} = \texttt{C}\,.\,\texttt{meth}(\ldots)}{i \xrightarrow{calls} \texttt{C}\,.\,\texttt{meth}\,(\ldots)}$$

$$\text{Args}\ \frac{i:\ \texttt{p} = x\,.\,\texttt{meth}(\texttt{a}_1,\ \texttt{a}_2,\ \ldots,\ \texttt{a}_n) \qquad i \xrightarrow{calls} \texttt{meth}'\,(\texttt{v}_1,\ \texttt{v}_2,\ \ldots,\ \texttt{v}_n)}{\forall j:\ \texttt{a}_j \mapsto \hat{o}\ \Rightarrow\ \texttt{v}_j \mapsto \hat{o}}$$

$$\text{Ret}\ \frac{\begin{array}{c} i:\ \texttt{p} = x\,.\,\texttt{meth}(\ldots) \\ i \xrightarrow{calls} \texttt{meth}'\,(\ldots) \qquad j:\ \texttt{return}\ \texttt{q} \qquad j \in \textit{body}(\texttt{meth}') \\ \texttt{q} \mapsto \hat{o} \end{array}}{\texttt{p} \mapsto \hat{o}}$$

Figure 3.2: Inference Rules for Java Points-to Analysis

cast instructions, however, we also perform a type check to filter objects of incompatible types (by checking if the type of the allocation is a subtype of the type of the variable). Since Java is strongly-typed, this is the only place where we can benefit from such a type check; any other points-to edges established by the rest of the rules are guaranteed to be type-compatible.

The next two rules handle load and store instructions. To load from a field, we have to follow the relevant field points-to edge of any base object that we may load from. Inversely, storing to a field establishes such field points-to edges between any heap objects that the base and source variable may point to.

For virtual calls, we first have to determine the type(s) of the receiver object(s), and determine the actual method that will be called after method resolution. (For this purpose, we assume the existence of a *lookup* function : $T \times M \to M$ that performs the actual resolution.) Then, we can establish a call-graph edge for the given allocation site, as well as a points-to edge for variable `this` of the resolved method. Static calls are simpler, since they require no method resolution and have no receivers. The last two rules apply to both virtual and static calls, and model call and return instructions as (interprocedural) assignments (similarly to the C/C++ setting), regarding method arguments and returned values.

3.3 Joint Reflection and Points-To Analysis

Next, we extend our abstracted model of the points-to analysis with an inter-related reflection analysis. The model is a light reformulation of the analysis introduced by Livshits et al. [84, 85]. The main insight of the Livshits et al. approach is that reflection analysis relies on points-to information, because the different key elements of a reflective activity may be dispersed throughout the program. A typical pattern of reflection usage is with code such as:

```
String className = ... ;
Class c = Class.forName(className);
Object o = c.newInstance();
String methodName = ... ;
Method m = c.getMethod(methodName, ...);
m.invoke(o, ...);
```

All of the above statements can occur in distant program locations, across different methods, invoked through virtual calls from multiple sites, etc. Thus, a whole-program analysis with an understanding of heap objects is required to track reflection with any amount of precision. This suggest the idea that reflection analysis can leverage points-to analysis—it is a client for points-to information. At the same time, points-to analysis needs the results of reflection analysis—e.g., to determine which method gets invoked in the last line of the above example, or what objects each of the example's local variables point to. Thus, under the Livshits et al. approach, reflection analysis and points-to analysis become mutually recursive, or effectively a single analysis.

Recall that, in the C/C++ setting of Chapter 2, we used the same insight to associate untyped abstract objects with their possible types. The points-to analysis is used as both a producer and consumer of type information: new type-object associations drive the creation of new abstract objects, altering the points-to results that may, again, produce new type information, and so on.

We consider the core of the analysis algorithm, which is representative and handles the most common features, illustrated in our above example: creating a reflective object rep-

$$\text{Class.forName}\ \frac{i:\ \texttt{c} = \texttt{Class.forName}\,(\texttt{s}) \qquad \texttt{s} \mapsto \widehat{str} \qquad \textit{fqn}(\,\texttt{T}\,) = \widehat{str} \qquad \texttt{T} \in T}{\texttt{c} \mapsto \widehat{cls}_{\texttt{T}}}$$

$$\text{Class.newInstance}\ \frac{i:\ \texttt{p} = \texttt{c.newInstance}\,() \qquad \texttt{c} \mapsto \widehat{cls}_{\texttt{T}}}{\texttt{p} \mapsto \widehat{o_{i,\texttt{T}}}}$$

$$\text{Class.getMethod}\ \frac{\begin{array}{c} i:\ \texttt{m} = \texttt{c.getMethod}\,(\texttt{s}) \qquad \texttt{s} \mapsto \widehat{str} \qquad \textit{fqn}(\,\texttt{meth}\,) = \widehat{str} \\ \texttt{c} \mapsto \widehat{cls}_{\texttt{T}} \qquad \textit{lookup}(\texttt{T},\ \texttt{meth}) = \_ \end{array}}{\texttt{m} \mapsto \widehat{mth}_{\texttt{meth}}}$$

$$\text{Method.invoke}\ \frac{\begin{array}{c} i:\ \texttt{p} = \texttt{m}\,.\,\texttt{invoke}(\texttt{r},\ \texttt{a}_1,\ \texttt{a}_2,\ \ldots) \qquad \texttt{m} \mapsto \widehat{mth}_{\texttt{meth}} \qquad \texttt{r} \mapsto \widehat{o_b} \\ \textit{type}(\widehat{o_b}) = \texttt{T} \qquad \textit{lookup}(\texttt{T},\ \texttt{meth}) = \texttt{meth}' \end{array}}{\begin{array}{c} i \xrightarrow{calls} \texttt{meth}'(\texttt{v}_1,\ \texttt{v}_2,\ \ldots) \qquad \texttt{this}_{\texttt{meth}'} \mapsto \widehat{o_b} \\ \forall j:\ \texttt{a}_j \mapsto \hat{o} \ \Rightarrow\ \texttt{v}_j \mapsto \hat{o} \end{array}}$$

Figure 3.3: Handling Java Reflection. We assume the existence of an overloaded function *fqn* : $(T \cup F \cup M) \to H$, that, given a class, field, or method, returns the string constant containing its fully qualified name. E.g. *fqn*(`Object`) = `"java.lang.Object"`.

resenting a class (a *class object*) given a name string (library method `Class.forName`), creating a new object given a class object (library method `Class.newInstance`), retrieving a reflective method object given a class object and a signature (library method `Class.getMethod`), and reflectively calling a virtual method on an object (library method `Method.invoke`). This treatment ignores several other APIs, which are handled similarly. These include, for instance, fields, constructors, other kinds of method invocations (static, special), reflective access to arrays, other ways to get class objects, and more.

The Livshits et al. reflection analysis can be expressed as a four-rule addition to the points-to analysis of Section 3.2. However, we first have to extend our domain of abstract heap objects with some special kinds of *reflective objects*. These new kinds of objects consist of the following:

- $\widehat{cls}_{\texttt{T}}$ Represents the reflective class object for class type $\texttt{T} \in T$.
- $\widehat{mth}_{\texttt{m}}$ Represents the reflective object for method $\texttt{m} \in M$.
- $\widehat{o_{i,\texttt{T}}}$ Represents objects of type $\texttt{T} \in T$ that are allocated with a `newInstance()` call at invocation site $i \in L$.

The first rule of Figure 3.3 models a `forName` call, which returns a class object given a string representing the class name. It states that if the argument of a `forName` call points to an object that is a string constant containing the name of class type `T`, then the target variable of the `forName` call is inferred to point to the respective reflection object for class type T.

The second rule reads: if the receiver object of a newInstance call is a class object for class type T, and the newInstance call is assigned to variable p, then make p point to the special (i.e., invented) abstract object $\widehat{o_{i,\mathtt{T}}}$ that designates objects of type T allocated at the newInstance call site. Note the analogy between this kind of abstract objects, $\widehat{o_{i,\mathtt{T}}}$, and the abstract objects of Chapter 2, representing the various typed variants of a seemingly untyped allocation (e.g, malloc()). In both cases, the same allocation site may produce multiple abstract objects, one per each type associated with this site, as computed by the pointer analysis itself.

The third rule gives semantics to getMethod calls. It states that if such a call is made with receiver c (for "base") and first argument s (the string encoding the desired method's signature), and if the analysis has already determined the objects that c and s may point to, then, assuming c points to a string constant encoding the signature of some method, meth, that exists inside the type that c points to ("_" stands for "any" value), the variable m holding the result of the getMethod call points to the reflective object, $\widehat{mth}_{\mathtt{meth}}$, for this method signature.

Finally, all reflection information can contribute to inferring more call-graph edges. The last rule encodes that a new edge can be inferred from the invocation site, i, of a reflective invoke call to a method meth$'$, if the receiver, m, of the invoke points to a reflective object encoding method meth, and the argument, r, of the invoke points to an object, $\widehat{o_b}$, of a class in which the lookup of meth's signature produces the method meth$'$. Method parameters are handled similarly to ordinary method calls (i.e., as interprocedural assignments).

The four rules of Figure 3.3 are a small part of a realistic implementation of reflection handling, but they offer a faithful model of the core of the analysis—other additions handle more reflective calls and more language types (e.g., arrays) but represent engineering, rather than conceptual handling.

3.4 Techniques for Empirical Soundness

We next present our main techniques for higher empirical soundness.

3.4.1 Generalizing Reflection Inference via Substring Analysis

An important way of enhancing the empirical soundness of our analysis is via richer string flow. The logic discussed in Section 3.3 only captures the case of entire string constants used as parameters to a forName call. The parameter of forName could be any string expression, however. It is interesting to attempt to deduce whether such an expression can refer to a class name. Similarly, strings representing field and method names are used in reflective calls—we already encountered the getMethod call in Section 3.3.

```
boolean setAttrVal(..., ElemTemplateElement el) {
    String setterString = getSetterMethodName();
    Object val = processValue(..., el);

    Object[] args = new Object[]{ val };
    Class[] argTypes = new Class[]{ val.getClass() };

    Method meth = el.getClass().getMethod(setterString, argTypes);
    meth.invoke(el, args);
}

public String getSetterMethodName() {
    StringBuffer outBuf = new StringBuffer();
    outBuf.append("set");

    for (int i = 0; i < m_name.length(); i++) {
        char c = m_name.charAt(i);
        if ('-' == c) {
            i++;
            c = m_name.charAt(i);
            c = Character.toUpperCase(c);
        }
        else if (0 == i) {
            c = Character.toUpperCase(c);
        }
        outBuf.append(c);
    }
    return outBuf.toString();
}
```

Figure 3.4: Example of reflection leveraging partial strings.

Reflection Usage Example. The (simplified) code excerpt shown in Figure 3.4, found in the *xalan* DaCapo benchmark, demonstrates the need for substring analysis in order to resolve reflective method invocations. The methods shown belong to class `org.apache .xalan.processor.XSLTAttributeDef`, which represents an attribute for an element in an XSLT stylesheet. The method `setAttrVal()` computes and sets the value of this attribute, for a given element (of type `ElemTemplateElement`), via reflection (calls `getClass`, `getMethod`, `invoke`). In order to achieve this, it first has to determine the exact name of the setter method of the element, by calling `getSetterMethodName()`. The attribute contains a field `m_name`, which holds the local name of the attribute without any prefix. The method simply transforms this local name to a setter method by adding a “`set`” prefix, removing dashes, and changing it to camel case. (Reflective calls have to be generic, which explains why patterns such as this, relying on naming conventions and employing some basic string transformation, are common in practice.)

Note that, in order to resolve the setter method, one needs to track the flow of the “`set`” prefix through the `StringBuffer` object and use it to match against any possible setter methods of `ElemTemplateElement`. Ideally, we would like to narrow down the setter meth-

$$\text{String Builder}\quad \frac{\begin{array}{c} i:\ \texttt{b}_1\texttt{.append}\,(\texttt{s}) \\ j:\ \texttt{r} = \texttt{b}_2\texttt{.toString}\,() \qquad \texttt{b}_1 \mapsto \widehat{o_b} \qquad \texttt{b}_2 \mapsto \widehat{o_b} \\ \textit{type}(\widehat{o_b}) = \texttt{StringBuilder} \qquad \texttt{s} \mapsto \widehat{o_r} \qquad \textit{matches}(\,\widehat{o_r},\ \_\,) \end{array}}{\texttt{r} \mapsto \widehat{o_r}}$$

$$\text{Class Substr}\quad \frac{i:\ \texttt{c} = \texttt{Class.forName}\,(\texttt{s}) \qquad \texttt{s} \mapsto \widehat{o_r} \qquad \textit{matches}(\,\widehat{o_r},\ \texttt{T}\,) \qquad \texttt{T} \in T}{\texttt{c} \mapsto \widehat{cls}_{\texttt{T}}}$$

Figure 3.5: Extending reflection handling with substring matching

ods to be called to just one, but this would require sophisticated reasoning about the computation performed inside `getSetterMethodName()`. Such reasoning is outside the scope of this dissertation and fairly foreign to scalable over-approximate techniques, such as pointer analysis. Furthermore, the exact value of `m_name` could be missing or merged with many other (irrelevant) string constants.

Substring matching approach. In order to estimate what classes, fields, or methods a string expression may represent, we implement *substring matching*: all string constants in the program text are tested for prefix and suffix matching against known class, method, and field names. (We use tunable thresholds to limit the matches: e.g., member prefixes, resp. suffixes, need to be at least 3, resp. 5, characters long. These settings reflect a balance between expected usage and spurious matches.)

The strings that may refer to such entities are handled with more precision than others during analysis. For instance, a points-to analysis (e.g., in the Doop or Wala frameworks) will typically merge most strings into a single abstract object—otherwise the analysis will incur an overwhelmingly high cost because of tracking numerous string constants. Strings that may represent class/interface, method, or field names are prevented from such merging. Furthermore, the flow of such strings through factory objects is tracked.

String concatenation in Java is typically done through `StringBuffer` or `StringBuilder` objects. The common concatenation operator, +, reduces to calls over such factory objects. To evaluate whether reflection-related substrings may flow into factory objects, we leverage the points-to analysis itself, pretending that an object flow into an `append` method and out of a `toString` method is tantamount to an assignment.

Figure 3.5 contains a simplified version of the logic. It assumes that we have already computed the following:

matches : $H \times T \rightarrow \{\,true,\ false\,\}$ a predicate that is true if a heap object is a string constant that matches a class type (as described above), or false otherwise.

The first rule of Figure 3.5 states: if a call to `append` and a call to `toString` are over the same string builder object, $\widehat{o_b}$, (accessed by different vars, $\texttt{b}_1$ and $\texttt{b}_2$, at possibly disparate parts of the program) then all the potentially reflection-related objects that are pointed to by the parameter, s, of append are inferred to be pointed by the variable r that accepts the result of the toString call. The second rule augments the treatment of forName instructions to relax the association between string constants and class types, by also allowing partial strings to map to their matching types.

In this way, the flow of partial string expressions through the program is tracked. By appropriately adjusting the *matches* predicate, we can estimate which reflective entities can be returned at the site of a forName call. (Calls to getMethod call can be similarly extended.) In this way, the joint points-to and reflection analysis is enhanced with substring reasoning without requiring any changes to the base logic of Section 3.3. String flow through buffers becomes just an enhancement of the points-to logic, which is already leveraged by reflection analysis.

An interesting aspect of the above approach is that it is easily configurable, in commonly desirable ways. Our above rule for handling partial string flow through string factory objects does not concern itself with how string factory objects (h_f) are represented inside the analysis. Indeed, string factory objects are often as numerous as strings themselves, since they are implicitly allocated on every use of the + operator over strings in a Java program. Therefore, a pointer analysis will often merge string factory objects, with the appropriate user-selectable flag.[2] The rule for string flow through factories is unaffected by this treatment. Although precision is lost if all string factory objects are merged into one abstract object, the joint points-to and reflection analysis still computes a fairly precise outcome: "does a partial string that matches some class/method/field name flow into some string factory's append method, and does some string factory's toString result flow into a reflection operation?" If both conditions are satisfied, the class/method/field name matched by the partial string is considered to flow into the reflection operation.

3.4.2 Use-Based Reflection Analysis

Our second technique for statically analyzing reflection calls leverages the way objects returned by reflective calls are later used in the program. We call the approach *use-based reflection analysis* and it integrates two sub-techniques: a *back-propagation* mechanism and a (forward) *object invention* mechanism. We discuss these next.

3.4.2.1 Inter-procedural Back-Propagation

An important observation regarding reflection handling is that it is one of the few parts of a static analysis that are typically *under-approximate* rather than *over-approximate*. A static points-to analysis is primarily a *may* analysis: it computes a conservative over-approximation of the analyzed program's behavior. This is usually impossible to do in the presence of reflection: the analysis cannot know all the values that a string expression

[2] E.g., For instance, this is enabled with the flag SMUSH_STRINGS in Wala [40] and MERGE_STRING_BUFFERS in Doop. Both flags are on by default for precise (i.e., costly) analyses.

can assume. Of course, the analysis could over-approximate such values (e.g., assume that *any* string is possible) but such treatment is catastrophic for precision and scalability: a single reflective call would lead to vast imprecision propagating through the program. No actual, implemented whole-program analysis attempts such over-approximation [86]. Instead, analyses choose to purposely treat reflective calls under-approximately: when the arguments of the reflection call are possible to infer, they are taken into account; other potential values are ignored.

Our first use-based reflection analysis technique back-propagates information from the use-site of a reflective result *to the original reflection call that got under-approximated*. Such an under-approximated call can be a:

– `Class.forName` call, as seen earlier: returns a dynamic representation of a class, given a string.
– `Class.get[Declared]Method` call, as seen earlier: returns a dynamic representation of a method, given a class and a string.
– `Class.get[Declared]Field` call: returns a dynamic representation of a field, given a class and a string.

The example below, which we will refer to repeatedly in later sections, shows how the use of a non-reflection object can inform a reflection call's analysis:

```
Class c1 = Class.forName(className);
...      // c2 aliases c1
Object o1 = c2.newInstance();
...      // o2 aliases o1
e = (Event) o2;
```

Typically (e.g., when `className` does not point to a known constant) the `forName` call will be under-approximated (rather than, e.g., assuming it will return any class in the system). The idea is to then treat the cast as a hint: it suggests that the earlier `forName` call should have returned a class object for `Event`. This reasoning, however, should be *inter-procedural* with an understanding of heap behavior. The above statements could be in distant parts of the program (separate methods) and aliasing is part of the conditions in the above pattern. Further, note that the related objects are twice-removed: we see a cast on an *instance* object and need to infer something about the `forName` site that *may* have been used to create the class that got used to allocate that object. This propagation should be as precise as possible: lack of precision will lead to too many class objects returned at the `forName` call site, affecting scalability.

Therefore, we see again the need to employ points-to analysis, this time in order to detect the relationship between cast sites and `forName` sites, so that the latter can be better resolved and we can improve the points-to analysis itself—a mutual recursion pattern. The high-level structure of our technique (for this pattern) is as follows:

- At the site of a `forName` call, create a marker object (of type `java.lang.Class`), to stand for all unknown objects that the invocation may return.

$$\text{Mark Class}\quad \frac{i :\ \texttt{c} = \texttt{Class.forName}\,(\texttt{p})}{\texttt{c} \mapsto \widehat{cls}_{\texttt{i,*}}}$$

$$\text{Mark Object}\quad \frac{j :\ \texttt{p} = \texttt{c.newInstance}\,() \qquad \texttt{c} \mapsto \widehat{cls}_{\texttt{i,*}}}{\texttt{p} \mapsto \widehat{o_{i,*}}}$$

$$\text{Back Prop}\quad \frac{i :\ \texttt{c} = \texttt{Class.forName}\,(\texttt{x}) \qquad j :\ \texttt{p} = (\texttt{T})\,\texttt{q} \qquad \texttt{T'<:T} \qquad \texttt{q} \mapsto \widehat{o_{i,*}}}{\texttt{c} \mapsto \widehat{cls}_{\texttt{T'}}}$$

Figure 3.6: Extending reflection handling with back propagation

- The special object flows freely through the points-to analysis, taking full advantage of inter-procedural reasoning facilities.
- At the site of a `newInstance` invocation, if the receiver is our special object, the result of `newInstance` is also a special object (of type `java.lang.Object` this time) that remembers its `forName` origins.
- This second special object also flows freely through the points-to analysis, taking full advantage of inter-procedural reasoning facilities.
- If the second special object (of type `java.lang.Object`) reaches the site of a cast, then the original `forName` invocation is retrieved and augmented to return the cast type or its subtypes as class objects.

The algorithm for the above treatment can be elegantly expressed via rules that are mutually recursive with the base points-to analysis. The rules for the `forName`-`newInstance`-cast pattern are representative.

As before, we have to introduce new kinds of abstract objects to implement this technique:

– $\widehat{cls}_{\texttt{i,*}}$ A marker class object (of no specific type) that is produced by calling `forName` at an allocation site $i \in L$.

– $\widehat{o_{i,*}}$ Represents all objects (again, of no specific type) returned by a `newInstance` call, which was, in turn, performed on the special (marker) object returned by a `forName` call, at invocation site $i \in L$.

Figure 3.6 contains the rules that we have to add. The first rule makes the variable that was assigned the result of a `forName` invocation point to the special object representing all missing objects from this invocation site. In this way, the special object can then propagate through the points-to analysis.

The second rule reads: when analyzing a `newInstance` call, if the receiver is a special object that was produced by a `forName` invocation, i, then the result of the `newInstance` will be another special object that will identify the `forName` call.

The final rule ties the logic together: if a cast to type `T` is found, where the cast variable points to a special object, $\widehat{o_{i,*}}$, then retrieve the object's `forName` invocation site, `i`, and infer that this invocation site returns a class object of type `T'`, where `T'` is a subtype of `T`. Using casts as type hints is exactly what we resorted to in Chapter 2, as well, to deal with untyped heap allocations. The logic here is fairly similar, but instead of patching prior `malloc` instructions, it is used to patch `forName` calls.

Other use-cases. As seen above, the back-propagation logic involves the result of several inter-procedural queries (e.g., points-to information at possibly distant call sites). In fact, there are use-based back-propagation patterns with even longer chains of reasoning. In the case below, the cast of `o2` informs the return value of `forName`, three reflection calls back!

```
Class c1 = Class.forName(className);
...      // c2 aliases c1
Constructor[] cons1 = c2.getConstructors(types);
...      // cons2 aliases cons1
Object o1 = cons2[i].newInstance(args);
...      // o2 aliases o1
e = (Event) o2;
```

Interestingly, the back-propagation analysis can exploit not just cast information but also strings (including partial strings, transparently, per our substring/string-flow analysis of Section 3.4.1). When retrieving a member from a reflectively discovered class, the string name supplied may contain enough information to disambiguate what this class may be. Consider the pattern:

```
Class c1 = Class.forName(className);
...      // c2 aliases c1
Field f = c2.getField(fieldName);
```

In this case, the value of the `fieldName` string can inform the analysis result for the earlier `forName` call. We apply this idea to the 4 API calls `Class.get[Declared]Method` and `Class.get[Declared]Field`.

Contrasting approaches. Our back-propagating reflection analysis (Section 3.4.2.1) has some close relatives in the literature. Livshits et al. [84, 85] also examined using future casts as hints for `forName` calls, as an alternative to regular string inference. Li et al. [80] generalize the Livshits approach to many more reflection calls. There are, however, important ways in which our techniques differ:

- Our analysis generalizes the pattern significantly. In our earlier example, from the beginning of this section, both the Li et al. and the Livshits et al. approaches require for the cast to not only occur in the same method as the `newInstance` call but also to post-dominate it! This restricts the pattern to an intra-procedural and fairly specific setting, reducing its generality:

```
Class c1 = Class.forName(className);
...       // c2 aliases c1
e = (Event) c2.newInstance();
```

The result of such a restriction is that the potential for imprecision is diminished, yet the ability to achieve empirical soundness is also scaled back. There are several cases where the cast will not post-dominate the intermediate reflection call, yet could yield useful information. This is precisely what Livshits et al. encountered experimentally—a direct quote illustrates:

> *The high number of unresolved calls in the JDK is due to the fact that reflection use in libraries tends to be highly generic and it is common to have 'Class.newInstance wrappers'—methods that accept a class name as a string and return an object of that class, which is later cast to an appropriate type in the caller method. Since we rely on intraprocedural post-dominance, resolving these calls is beyond our scope.* [85]

- We generalize back-propagation to string information and not just cast information (i.e., we exploit the use of `get[Declared]{Method,Field}` calls to resolve earlier `forName` calls). This feature also benefits from other elements of our overall analysis, namely substring matching and substring flow analysis (Section 3.4.1). For instance, by having more information on what are the possible strings passed to a `getMethod` call, we are more likely to determine the return value of a `getClass`, on which the `getMethod` was called.

3.4.2.2 Inventing Objects

Our approach introduces an alternative use-based reflection analysis technique, which works as a *forward propagation* technique (in contrast to the earlier back-propagation). It consists of inventing objects of the appropriate type at the point of a cast operation that has received the result of a reflection call. Consider again our usual `forName`-`newInstance`-cast example:

```
1 Class c1 = Class.forName(className);
2 ...       // c2 aliases c1
3 Object o1 = c2.newInstance();
4 ...       // o2 aliases o1
5 e = (Event) o2;
```

A major issue with our earlier back-propagation technique is that its results may adversely affect precision. The information will flow back to the site of the `forName` call, and from there to multiple other program points—not just to the point of the cast operation (line 5), or even to the point of the `newInstance` operation (line 3) in the example.

The object invention technique offers the converse compromise. Whenever a special, unknown reflective object flows to the point of a cast, instead of informing the result of `forName`, the technique invents a new, regular object of the right type (`Event`, in this case)

$$\text{Class.newInstance}^{+}\ \frac{i:\ \mathtt{p} = \mathtt{c.newInstance}\,()}{\mathtt{p} \mapsto \widehat{o^{+}_{i,*}}} \qquad \text{Invent}\ \frac{j:\ \mathtt{p} = (\mathtt{T})\,\mathtt{q} \qquad \mathtt{q} \mapsto \widehat{o^{+}_{i,*}}}{\mathtt{p} \mapsto \widehat{o^{+}_{i,\mathtt{T}}}}$$

Figure 3.7: Extending reflection handling with object invention

that starts its existence at the cast site. The "invented" object does not necessarily abstract actual run-time objects. Instead, it exploits the fact that a points-to analysis is fundamentally a may-analysis: it is designed to possibly yield over-approximate results, in addition to those arising in real executions. Thus, an invented value does not impact the correctness of the analysis (since having extra values in points-to sets is acceptable), yet it will enable it to explore possibilities that might not exist without the invented value. These possibilities are, however, strongly hinted by the existence of a cast in the code, over an object derived from reflection operations.

The algorithm for object invention in the analysis is again recursive with the main points-to logic. We illustrate for the case of `Class.newInstance`, although similar logic applies to reflection calls such as `Constructor.newInstance`, as well as `Method.invoke` and `Field.get`.

As in the back-propagating analysis, we use special marker objects.

- $\widehat{o^{+}_{i,*}}$ An invented object of unknown type for a given `newInstance` invocation site, $i \in L$.
- $\widehat{o^{+}_{i,\mathtt{T}}}$ An invented object of type $\mathtt{T} \in T$ for a given `newInstance` invocation site, $i \in L$.

The algorithm is captured in the two rules of Figure 3.7. The first one states that the variable assigned the result of a `newInstance` invocation points to a special object marking that it was produced by a reflection call. The marker object can then propagate through the points-to analysis.

The key part of the algorithm is to then invent an object at a cast site. This happens in the second rule: if a variable, `q`, is cast to a type `T` and points to a marker object that was produced by a `newInstance` call, then the variable, `p`, storing the result of the cast, points to a newly invented object, with the right type, `T`.

Note that in terms of empirical soundness the object invention approach is weaker, in most cases,[3] than the back-propagation analysis: if a type is inferred to be produced by an earlier `forName` call, it will flow down to the point of the cast, removing the need for object invention. (Conversely, inventing objects at the cast site will not catch all cases covered by back-propagation, since the special object of the back-propagation analysis may never flow to a cast.) Nevertheless, back-propagation is often less scalable. Thus, the benefit of object invention is that it allows to selectively turn off back-propagation while

[3] Although it is not weaker when (due to other unsoundness, e.g., dynamic loading or unrelated reflection) the value returned by a `forName` call is not detected to flow to the appropriate `newInstance`.

still taking advantage of information from a cast.

3.4.3 Balancing for Scalability

Consider again our inter-procedural back-propagating analysis technique relative to prior, intra-procedural techniques. Our approach explicitly aims for empirical soundness (i.e., to infer all potential results of a reflection call). At the same time, however, the technique may suffer in precision, since the result of a reflection call is deduced from far-away information, which may be highly over-approximate. Conversely, our object invention technique is more precise (since the invented object only starts existing at the point of the cast) but may suffer in terms of soundness. Thus, it can be used to supplement back-propagation when the latter is applied selectively.

To balance the soundness/precision tradeoff of the back-propagating analysis, we employ precision thresholds. Namely, back-propagation is applied only when it is reasonably precise in terms of type information. For instance, if a cast is found, it is used to back-propagate reflective information only when there are up to a constant, c, class types that can satisfy the cast (i.e., at most c subtypes of the cast type). Intuitively, a cast of the form “`(Event)`” is much more informative when `Event` is a class with only a few subclasses, rather than when `Event` is an interface that many tens of classes implement. Similarly, if string information (e.g., a method name) is used to determine what class object could have been returned by a `Class.forName`, the back-propagation takes place only when the string name matches methods of at most d different types. This threshold approach minimizes the potential for noise back-propagating and polluting all subsequent program paths that depend on the original reflection call.

A second technique for employing back-propagation without sacrificing precision and scalability adjusts the flow of special abstract objects that we introduced with our extensions. Although we want such objects to flow inter-procedurally, we can disallow their tracking through the heap (i.e., through objects or arrays), allowing only their flow through local variables. This is consistent with expected inter-procedural usage patterns of reflection results: although such results will likely be returned from methods (cf. the quote from [85] in Section 3.4.2.1), they are less likely to be stored in heap objects.

We employ both of the above techniques by default in our analysis (with $c = d = 5$). The user can configure their application through input options.

3.5 Evaluation

We implemented our techniques in the Doop framework [20], together with numerous improvements (i.e., complete API support) to Doop’s reflection handling. Following the Elf study [80], we perform the default joint points-to and call-graph analysis of Doop, which is an Andersen-style context-insensitive analysis, with full support for complex Java language features, such as class initialization, exceptions, etc. Our techniques are orthogonal to the context-sensitivity used, and can be applied to all analyses in the Doop framework. In general, nothing in our modeling of reflection limits either context- or flow-sensitivity.

The evaluation of our techniques aims to answer three research questions:

RQ1: *Can these techniques improve the soundness of a points-to analysis?*

RQ2: *Do the presented techniques have reasonable running times?*

RQ3: *Does an increase in soundness incur a significant loss in precision?*

Experimental Setup. 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. We have used a JVMTI agent to construct a dynamic call-graph for each analyzed program, by instrumenting its execution.

We analyze 10 benchmark programs from the DaCapo 9.12-Bach suite [17], with their default inputs (for the purposes of the dynamic analysis). Other benchmarks could not be executed or analyzed: *tradebeans*/*tradesoap* from 9.12-Bach do not run with our instrumentation agent, hence no dynamic call-graphs can be extracted for comparison. This is a known, independently documented, issue (see `http://sourceforge.net/p/dacapobench/bugs/70/`). We have been unable to meaningfully analyze *fop* and *tomcat*—significant entry points were missed. This suggests either a packaging error at determining what makes up the application and library code of each benchmark (manual repackaging is necessary since no application-library boundaries are provided by the DaCapo suite), or the extensive use of dynamic loading, which needs further special handling.

We use Oracle JDK 1.7.0_25 for the analysis. (For comparison, consider that the quote from [40] in the first section of this chapter, refers to the smaller JDK 1.6.)

Empirical soundness metric. We quantify the empirical unsoundness of the static analysis in terms of missing call-graph edges, compared to the dynamic call-graph. Call-graph construction is one of the best-known clients of points-to analysis [2, 3, 80] and has the added benefit of quantifying how much code the analysis truly reaches. We compare the call-graph edges found by our static analysis to a dynamic call-graph—a comparison also found in other recent work [126]. For a sound static analysis, no edge should occur dynamically but not predicted statically. However, this is not the case in practice, due to the unsound handling of dynamic features, as discussed in Section 3.1.

Results. Figure 3.8 plots the results of our experiments, combining both analysis time and empirical unsoundness (in call-graph edges). Missing bars labeled “n/a” correspond to analyses that did not terminate in 90mins (5400sec). Each chart plots the missing

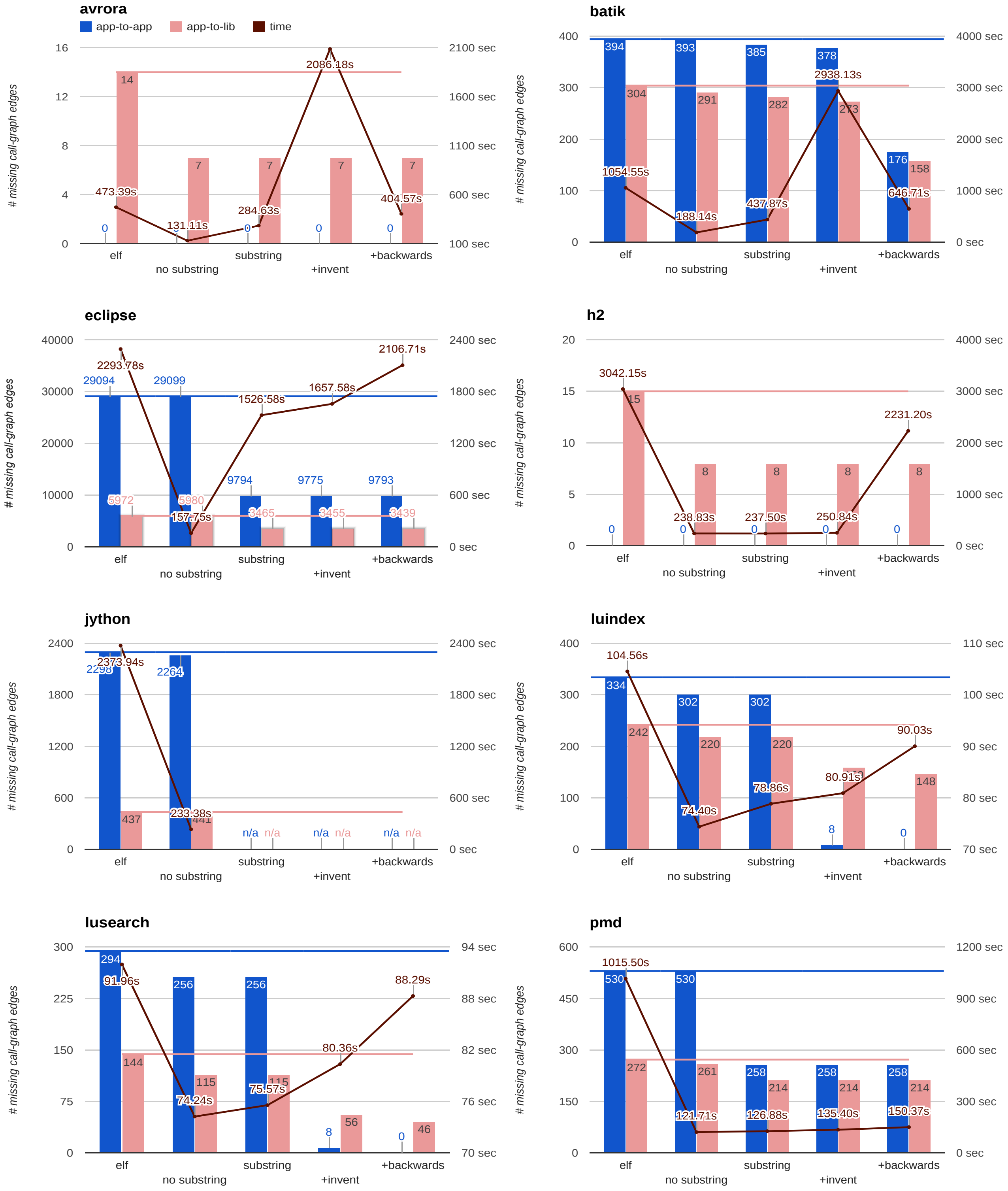


Figure 3.8: Unsoundness metrics (two bars: missing call-graph edges app-to-app and app-to-lib) and analysis time (line) over the DaCapo benchmarks. *Lower is better for all.* For missing bars ("n/a"), the analysis did not terminate in 90mins.

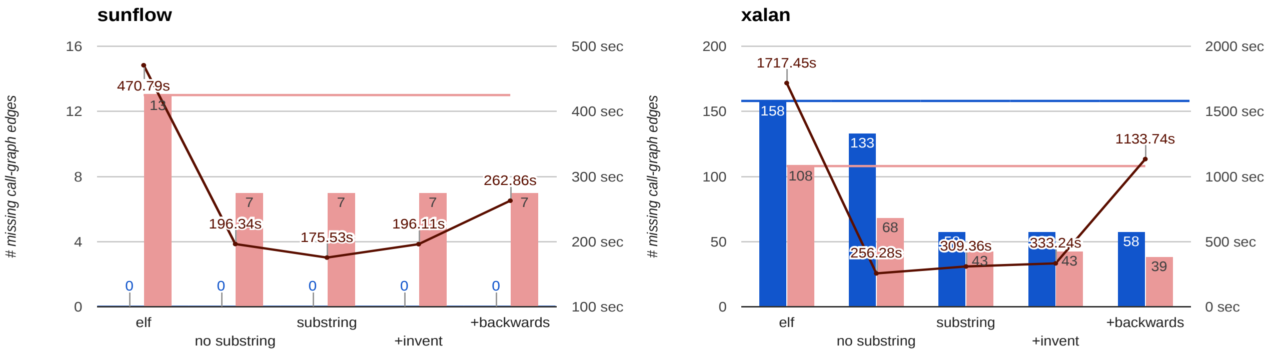


Figure 3.8: Unsoundness metrics (cont.)

Total Edges		Settings				
Benchmark	*dynamic*	elf	no substring	substring	+invent	+backwards
avrora	4165	19355	19379	20591	26586	20677
batik	8329	31602	31708	35314	47303	37013
eclipse	40026	10191	9032	115967	116635	117576
h2	4901	38252	35538	38107	38162	43952
jython	13583	19709	20537	n/a	n/a	n/a
luindex	3027	4547	4676	4682	5773	6115
lusearch	1845	4209	4352	4362	5266	5587
pmd	4874	8544	8592	9533	9557	9577
sunflow	2215	4223	4251	4285	4319	4407
xalan	6128	35918	35221	45160	45343	63746

Figure 3.9: Total static and dynamic call-graph edges for the DaCapo 9.12-Bach benchmarks. These include only *application-to-application* and *application-to-library* edges.

dynamic call-graph edges that are not discovered by the corresponding static analysis. We use separate bars for the *application-to-application* and *application-to-library* edges.

We consider only call-graph edges originating from application code, since library classes contain a fair amount of non-analyzable native methods.[4] We also filter out some missing edges (i.e., consider them implicitly covered), which involve the following methods:

- *Class Initializers.* Doop only models *which* subset of classes get initialized (without any information about where the initializer gets called from). We filter out edges to class initializer methods (i.e., `<clinit>`), if static analysis indicates that the class has been initialized.

[4] Call-graph edges from the library are still fully statically analyzed, thus our experiments demonstrate scalability relative to large libraries. We just do not report library-originating edges (though they are computed within the time reported) since these only cloud the picture, due to native code. There is no easy way to compare library-to-library results to dynamic edges without manual filtering, which raises validity questions.

- *Native.* Native code cannot be analyzed. However, some library reflection calls are wrappers for native methods (e.g., `forName()` and `forName0()`). Edges to these methods are, thus, completely extraneous due to our special modeling of their effect.
- *Class Loader.* Method `loadClass()` is invoked by the VM when a class needs to be loaded and `checkPackageAccess()` is invoked right after loading.
- *Synthetic.* Edges involving dynamically generated classes are impossible to obtain by reflection analysis alone, so we eliminate such instances.

We show five techniques:

1. *Elf.* This is the Elf reflection analysis [80], which also attempts to improve reflection analysis for Java.

2. *No substring.* Our reflection analysis, with engineering enhancements over the original Doop framework, but no analysis of partial strings or their flow.

3. *Substring.* The analysis integrates the substring and substring flow analysis of Section 3.4.1.

4. *+Invent.* This analysis integrates substring analysis as well as the object invention technique of Section 3.4.2.2.

5. *+Backwards.*[5] This analysis integrates substring analysis as well as the back-propagation technique of Section 3.4.2.1.

It is important to note that, by design, our techniques do not enhance the precision of an analysis, only its empirical soundness. Thus, the techniques only find *more* edges: they cover more of the program. This improvement appears as a reduction in the figures ("lower is better") only because the number plotted is the *difference* in the missing edges compared to the dynamic analysis.

Our research questions can now be answered:

RQ1: Do our techniques impact soundness? As can be seen, our techniques substantially increase the soundness of the analysis. In most benchmarks, more than half (to nearly all) of the missing *application-to-application* edges are recovered by at least one technique. The *application-to-library* missing edges also decreased, although not as much. In fact, the *eclipse* benchmark was hardly being analyzed in the past, since most of the dynamic call-graph was missing.

RQ2: Do the techniques have reasonable running times? Furthermore, although our approach emphasizes empirical soundness, it does not sacrifice scalability. All four of our settings are faster than Elf for almost all benchmarks. Aside from *jython*, for which only the Elf and *no substring* techniques are able to terminate before timeout,

[5] The *+Backwards* and *+Invent* techniques are both additions to the substring analysis, but neither includes the other.

in all other cases *substring* and at least one of *+invent* or *+backwards* outperformed Elf, while in 7-of-10 benchmarks *all* our techniques outperformed Elf. This is due to achieving scalability using the threshold techniques of Section 3.4.3 instead of by sacrificing some empirical soundness, as Elf does. (A major design feature of Elf is that it explicitly avoids inferring reflection call targets when it cannot fully disambiguate them.)

RQ3: Do the techniques sacrifice precision? For completeness, we also show a sanity-checking metric over our analyses. Empirical soundness could increase by computing a vastly imprecise call-graph. This is not the case for our techniques. Figure 3.9 lists the total static and dynamic edges being computed. On average, *+backwards* computes the most static edges (about 4.5 times the number of dynamic edges). On the lower end of the spectrum lies *no substring*, with a minimum of 3.4 times the number of dynamic edges being computed.

In pragmatic terms, a user of our analysis should use flags to pick the technique that yields more soundness without sacrificing scalability, for the given input program. This is a familiar approach—e.g., it also applies to picking the exact flavor and depth of context-sensitivity.

As a final note, the improved soundness due to the techniques presented in this chapter supports our thesis statement: *the abstract model of memory is, indeed, improved by recovering implicit structural information via inference, primarily by tracking the use of types in the program.* The use-based techniques leverage the way objects (returned by reflective calls) are later used in the program: i.e., what types they are cast to, what fields they access, and so on. By inspecting the use of reflective objects, these techniques are able to infer and partly recover the objects' structure, which is not evident at the site of their allocation, since the declared types involved in reflective operations are too generic to accurately describe them. Recovering such implicit structural information leads to a better memory model; one that more faithfully abstracts the memory that will be allocated by any actual execution.

3.6 Summary

In this chapter, we considered the problem of recovering structural information for static analysis of Java. Structural information can be lost due to inadequate reasoning of prior approaches about common reflection patterns. Such reasoning often fails to identify the true types of many memory allocations and leads to unsoundness. We introduced the notion of empirical soundness, a metric that quantifies how much of the actual dynamic behavior the static analysis covers.

Section 3.1 discusses the need for better reflection handling in static analysis of Java programs and the complications it poses to pointer analysis, specifically. We give a simple model of a standard points-to analysis for Java, in Section 3.2. In Section 3.3, we add an inter-related reflection analysis to this model, and then present the extensions that constitute our approach in Section 3.4. Intuitively, the traditional points-to part of the joint analysis (Section 3.2) is responsible for computing how heap objects flow intra- and inter-procedurally through the program, while the added rules (of Sections 3.3 and 3.4) contribute only the reflection handling. Finally, we evaluate our approach on the DaCapo 9.12-Bach suite, in Section 3.5, in terms of empirical soundness, by comparing against dynamic call-graphs.

4. CLASS HIERARCHY COMPLEMENTATION: SOUNDLY COMPLETING A PARTIAL TYPE GRAPH

> What in God's holy name are you blathering about?
>
> *The Big Lebowski*

In the previous chapter, we examined the problems caused by Java's reflection and presented a reflection analysis, integrated into a standard points-to analysis, that recovers structural information for reflective objects. In this chapter, we will continue to examine the problem of lost memory structure in Java, yet in a completely different context: that of *partial Java programs*.

As stated in Chapter 1, analyzing partial programs is crucial for Java, whose dynamic class loading allows JAR files to depend on an abundance of external libraries, even though only a subset of them will be required at runtime—in any possible execution. The primary challenge in analyzing partial Java programs concerns the implied type constraints in existing code, and their repercussions on the type hierarchy.

This leads to the more generic problem of class hierarchy complementation: given a partially known hierarchy of classes together with subtyping constraints ("A has to be a transitive subtype of B") complete the hierarchy so that it satisfies all constraints.

The problem has immediate practical application to the analysis of partial programs—e.g., it arises in the process of providing a sound handling of "phantom classes" in the Soot program analysis framework. We provide algorithms to solve the hierarchy complementation problem in the single inheritance and multiple inheritance settings. We also show that the problem in a language such as Java, with single inheritance but multiple subtyping and distinguished class vs. interface types, can be decomposed into separate single- and multiple-subtyping instances. We implement our algorithms in a tool, JPhantom, which complements partial Java bytecode programs so that the result is guaranteed to satisfy the Java verifier requirements. In a sense, JPhantom aims to recover structural information for phantom classes, via inference, by tracking their use in existing code. JPhantom is highly scalable and runs in mere seconds even for large input applications and complex constraints (with a maximum of 14s for a 19MB binary).

4.1 Program Complementation and Partial Type Hierarchies

Whole-program static analysis is essential for clients that require high-precision and a deeper understanding of program behavior. Modern applications of program analysis, such as large scale refactoring tools [35], race and deadlock detectors [97], and security vulnerability detectors [47, 88], are virtually inconceivable without whole-program analysis.

For whole-program analysis to become truly practical, however, it needs to overcome several real-world challenges. One of the somewhat surprising real-world observations is that whole-program analysis requires the availability of much more than the "whole program". The analysis needs an overapproximation of what constitutes the program. Furthermore, this overapproximation is not merely what the analysis computes to be the "whole program" after it has completed executing. Instead, the overapproximation needs to be as conservative as required by any intermediate step of the analysis, which has not yet been able to tell, for instance, that some method is never called.

Consider the example of trying to analyze a program P that uses a third-party library L. Program P will likely only need small parts of L. However, other, entirely separate, parts of L may make use of a second library, L'. It is typically not possible to analyze P with a whole program analysis framework without also supplying the code not just for L but also for L', which is an unreasonable burden. In modern languages and runtime systems, L' is usually not necessary in order to either compile P or run it under any input. The problem is exacerbated in the current era of large-scale library reuse. In fact, it is often the case that the user is not even aware of the existence of L' until trying to analyze P.

Unsurprisingly, the issue has arisen before, in different guises. The FAQ document[1] of the well-known Soot framework for Java analysis [130, 131] contains the question:

> *How do I modify the code in order to enable soot to continue loading a class even if it doesn't find some of it[s] references? Can I create a dummy soot class so it can continue with the load? How?*

This frequently asked question does not lead to a solution. The answer provided is:

> *You can try -use-phantom-refs but often that does not work because not all analyses can cope with such references. The best way to cope with the problem is to find the missing code and provide it to Soot.*

The "phantom refs" facility of Soot, referenced in the above answer, attempts to model missing classes (*phantom classes*) by providing dummy implementations of their methods referenced in the program under analysis. However, there is no guarantee that the modeling is in any way sound, i.e., that it satisfies the well-formedness requirements that the rest of the program imposes on the phantom class.

Our research consists precisely of addressing the above need in full generality. *Given a set of Java class and interface definitions, in bytecode form, we compute a "program complement", i.e., skeletal versions of any referenced missing classes and interfaces so that the combined result constitutes verifiable Java bytecode.* Our solution to this practical problem has two parts:

· A *program analysis* part, requiring analysis of bytecode and techniques similar to those employed by the Java verifier and Java decompilers. This analysis computes constraints

[1] http://www.sable.mcgill.ca/soot/faq.html

involving the missing types. For instance, if a variable of a certain type C is direct-assigned to a variable of a type S, then C must be a subtype of S.

· An *algorithmic* part, solving a novel typing problem, which we call the *class hierarchy complementation*, or simply *hierarchy complementation*, problem. The problem consists of computing a type hierarchy that satisfies a set of subtyping constraints *without* changing the direct parents of known types.

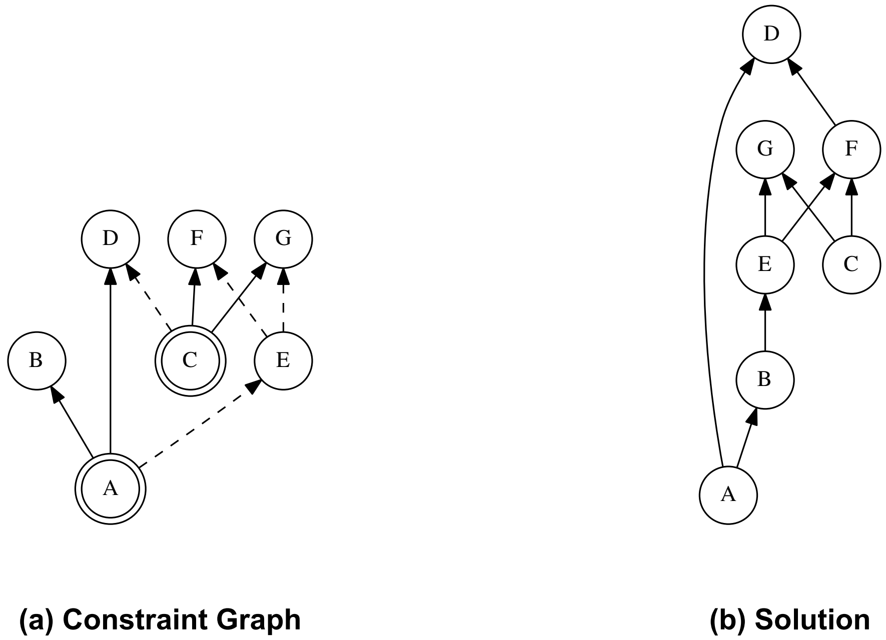
The algorithmic part of our solution, i.e., solving the hierarchy complementation problem, constitutes the main novelty of our approach. The problem appears to be fundamental, and even of a certain interest in purely graph-theoretic terms. For a representative special case, consider an object-oriented language with multiple inheritance (or, equivalently, an interface-only hierarchy in Java or C#).[2] A partial hierarchy, augmented with constraints, can be represented as a graph, as shown in Figure 4.1a. The known part of the hierarchy is shown as double circles and solid edges. Unknown (i.e., missing) classes are shown as single circles. Dashed edges represent subtyping constraints, i.e., indirect subtyping relations that have to hold in the resulting hierarchy. In graph-theoretic terms, a dashed edge means that there is a path in the solution between the two endpoints. For instance, the dashed edge from C to D in Figure 4.1a means that the unknown part of the class hierarchy has a path from C to D. This path cannot be a direct edge from C to D, however: C is a known class, so the set of its supertypes is fixed.

In order to solve the above problem instance, we need to compute a directed acyclic graph (DAG) over the same nodes,[3] so that it preserves all known nodes and edges, and adds edges *only to unknown nodes* so that all dashed-edge constraints are satisfied. That is, the solution will not contain dashed edges (indirect subtyping relationships), but every dashed edge in the input will have a matching directed path in the solution graph. Figure 4.1b shows one such possible solution. As can be seen, solving the constraints (or determining that they are unsatisfiable) is not trivial. In this example, any solution has to include an edge from B to E, for reasons that are not immediately apparent. Accordingly, if we change the input of Figure 4.1a to include an edge from E to B, then the constraints are not satisfiable—any attempted solution introduces a cycle. The essence of the algorithmic difficulty of the problem (compared to, say, a simple topological sort) is that we cannot add extra direct parents to known classes A and C—any subtyping constraints over these types have to be satisfied via existing parent types. This corresponds directly to our high-level program requirement: we want to compute definitions for the missing types only, without changing existing code.

For a language with single inheritance, the problem is similar, with one difference: the solution needs to be a tree instead of a DAG. (Of course, the input in Figure 4.1a already

[2] We avoid the terms "subclassing" or "inheritance" as synonyms for "direct subtyping" to prevent confusion with other connotations of these terms. In our context, we only care about the concept of subtyping, i.e., of a (monomorphic) type as a special case of another. Subtyping can be direct (e.g., when a Java class is declared to "extend" another or "implement" an interface) or indirect, i.e., transitive. We do, however, use the compound terms "single inheritance" and "multiple inheritance" as they are more common in the classification of languages than "single subtyping" and "multiple subtyping".

[3] Inventing extra nodes does not contribute to a solution in this problem.



(a) Constraint Graph **(b) Solution**

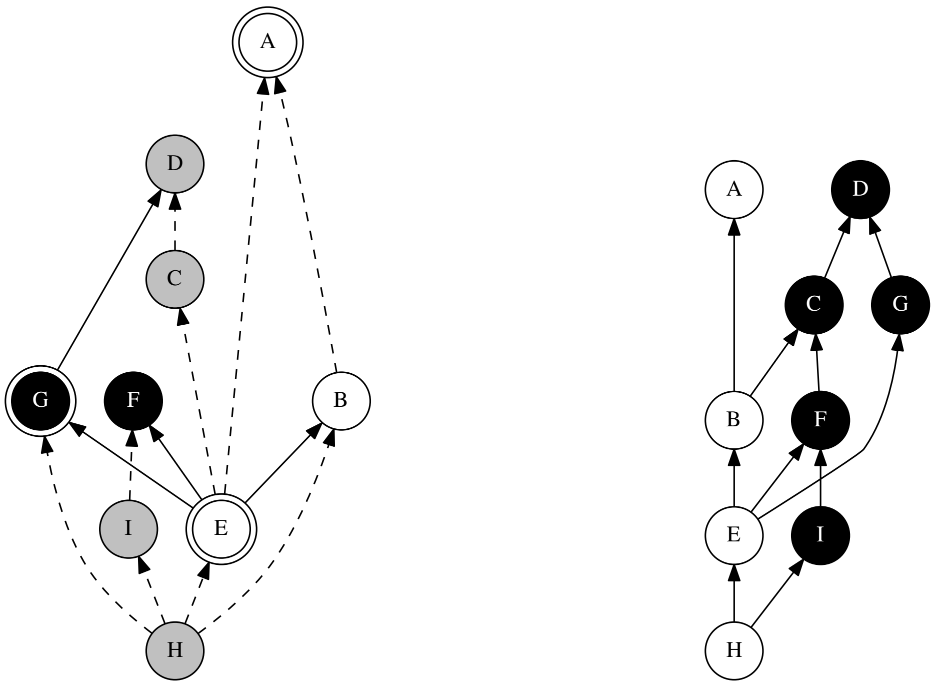
Figure 4.1: Example of constraints in a multiple inheritance setting. Double-circles signify known classes, single circles signify unknown classes. Solid edges ("known edges") signify direct subtyping, dashed edges signify transitive subtyping.

violates the tree property since it contains known nodes with multiple known parents.) We offer an algorithm that solves the problem by either detecting unsatisfiability or always ordering the nodes in a tree that respects all constraints.

The practical version of the hierarchy complementation problem is more complex. Mainstream OO languages often distinguish between classes and interfaces and only allow single direct subtyping among classes and multiple direct subtyping from a class/interface to an interface—a combination often called "single-inheritance, multiple subtyping". In this case, the graph representation of the problem is less intuitive. Consider Figure 4.2a that gives a problem instance. (A possible solution for these constraints is in Figure 4.2b, but is given purely for reference, as it is not central to our discussion.) There are now several node types: classes, interfaces (both known and unknown), as well as undetermined nodes. There are also more implicit constraints on them: classes can only have an edge to one other class, interfaces can only have edges to other interfaces. The latter constraint, for instance, forces D to be an interface and H to be a class. Thus, we see that the full version of the problem requires additional reasoning. We show that such reasoning can be performed as a pre-processing step. The problem can be subsequently broken up into two separate instances of the aforementioned single- and multiple-inheritance versions of hierarchy complementation.

In brief, the contributions of our work are as follows:

- We introduce a new typing problem, motivated by real-world needs for whole program



(a) Constraint Graph **(b) Solution**

Figure 4.2: Example of full-Java constraint graph. Double circles denote known classes/interfaces, whose outgoing edges in the solution are already determined (solid input edges). White nodes are classes, black nodes are interfaces, grey nodes are unknown types that are initially undetermined (i.e., the input does not explicitly identify them as classes or interfaces, although constraint reasoning may do so later).

analysis. To our knowledge, the hierarchy complementation problem has not been studied before, in any context.

– We produce algorithms that solve the problem in three different settings: single inheritance, multiple inheritance, and mixture of the two, as in Java or C#.

– We implement our algorithms in JPhantom: a practical tool for Java program complementation that addresses the soundness shortcomings of previous Java “phantom class” approaches. We show that JPhantom scales well and takes only a few seconds to process even large benchmarks with complex constraints—e.g., less than 6sec for a 3.2MB binary that induces more than 100 constraints.

– We discuss the problem of hierarchy complementation in more general settings. The simplicity of our approach is a result of only assuming (for the input) and satisfying (for the output) the fairly weak Java bytecode requirements. We show that the problem becomes harder at the level of the type system for the source language.

4.2 Motivation and Practical Setting

We next discuss the practical setting that gives rise to the hierarchy complementation problem.

Our interest in hierarchy complementation arose from efforts to complement existing Java bytecode in a way that satisfies the soundness guarantees of the Java verifier. Consider a small fragment of known Java bytecode and the constraints it induces over unknown types. (We present bytecode in a slightly condensed form, to make clear what method names or type names are referenced in every instruction.) In this code, classes `A` and `B` are available, while types `X`, `Y`, and `Z` are phantom, i.e., their definition is missing.

```
public void foo(X, Y)
0: aload_2      // load on stack 2nd argument (of type Y)
1: aload_1      // load on stack 1st argument (of type X)
2: invokevirtual X.bar:(LA;)LZ; // method 'Z bar(A)' in X
3: invokevirtual B.baz:()V;     // method 'void baz()' in B
 ...
```

The instructions of this fragment induce several constraints for our phantom types. For instance:

- `X` has to be a class (and not an interface) since it contains a method called via the `invokevirtual` bytecode instruction.
- `X` has to support a method `bar` accepting an argument of type `A` and returning a value of type `Z`.
- `Y` has to be a subtype of `A`, since an actual argument of declared type `Y` is passed to `bar`, which has a formal parameter of type `A`. This constraint also means that if `A` is known to be a class (and not an interface) then `Y` is also a class.
- `Z` has to be a subtype of `B`, since a method of `B` is invoked on an object of declared type `Z` (returned on top of the stack by the earlier invocation).

The goal of our JPhantom tool is to satisfy all such constraints and generate definitions of phantom types `X`, `Y`, and `Z` that are compatible with the bytecode that is available to the tool (i.e., exists in known classes). Compatibility with existing bytecode is defined as satisfying the requirements of the Java verifier, which concern type well-formedness.

Note that such definitions will contain essential parts of missing structural information for the phantom types: method and field members, as well as supertypes. Any subsequent static analysis that will operate on the types produced by JPhantom will create abstract objects that are much closer, in structure, to reality.

Of these constraints, the hardest to satisfy are those involving subtyping. Constraints on members (e.g., `X` has to contain a “`Z bar(A)`”) are easy to satisfy by just adding type-correct dummy members to the generated classes. This means that the problem in the core of JPhantom is solving the class hierarchy complementation problem, as presented in the introduction and defined rigorously in later sections. The binding of the problem to practical circumstances deserves some discussion, however.

First, note that, in our setting of the problem, we explicitly disallow modification of known code, e.g., in order to remove dependencies, or to add a supertype or a member to it. Such modifications would have a cascading effect and make it hard to argue about what properties are really preserved. Additionally, we do not assume any restrictions on the input, other than the well-formedness condition of being legal Java bytecode (according to the verifier). Strictly speaking, our well-formedness condition for the input is defined as follows: *a legal input is bytecode that can be complemented (by only adding extra class and interface definitions) so that it passes the Java verifier*. Note that this well-formedness condition does not depend on the program complement that our approach produces: an input is legal if there is *some* complement for it, not necessarily the one that JPhantom computes.

A final interesting point concerns the practical impact of the JPhantom soundness condition. For most program analyses, omitting parts of the code introduces unsoundness, if we make no other assumptions about the program or the omitted part. E.g., it is impossible to always soundly compute points-to information, or may-happen-in-parallel information when part of the program is missing. Therefore, guaranteed soundness for all clients is inherently unachievable for *any* partial program analysis approach. The practical reality is that there is a large need for facilities for handling partial programs. For instance, the Soot phantom class machinery has been one of the most common sources of discussion and questions on the Soot support lists, and it has been a central part of several Soot revisions.[4] The only "correctness condition" that Soot phantom class support is trying to achieve, however, is the low-level "the analyzer should not crash".

Given the practical interest for the solution of a worst-case unsolvable problem, we believe that our soundness guarantee makes a valuable contribution: it is much better to analyze a partial program in a way such that the Java verifier requirements (for type-level well-formedness) are satisfied than to ignore any correctness considerations, as past approaches do.

4.3 Hierarchy Complementation for Multiple Inheritance

We begin with a modeling of the hierarchy complementation problem in the setting of multiple inheritance. This means that every class in our output can have multiple parents.

We can model our problem as a graph problem. Our input is a directed graph $G = (V, E)$, with two disjoint sets of nodes $V = V_{known} \dot{\cup} V_{phantom}$ and two disjoint sets of edges $E = E_{direct} \dot{\cup} E_{path}$, where $E_{direct} \subseteq V_{known} \times V$ (i.e., direct edges have to originate from known nodes—the converse is not true, as known nodes can be inferred to

[4] Even the most recent Soot release, 2.5.0, lists improved support for phantom classes and excluding methods from an analysis as one of the major changes in the release notes.

subtype unknown ones due to assignment instructions in the bytecode). The set of nodes V is a set of types, while the set of edges E corresponds to our subtyping constraints. That is, an edge (v_s, v_t) encodes the constraint $v_s <: v_t$. The E_{direct} subset encodes the direct-subtype constraints. The output of our algorithm should be a *DAG* (with edges from children to their parents), $G_D = (V, E')$, such that:

1. $\forall v_s \in V_{known} : (v_s, v_t) \in E' \Leftrightarrow (v_s, v_t) \in E_{direct}$ (i.e., all direct edges from known nodes are preserved and no new ones are added to such nodes)

2. $(v_s, v_t) \in E_{path} \Rightarrow$ there is a path from v_s to v_t in G_D

Note that our only limiting constraint here is that we cannot have cycles in the resulting hierarchy. Moreover, since each type may have multiple supertypes in this setting, a directed acyclic graph is fitting as our intended output.

In contrast to the general case, the problem is trivial if we have a phantom-only input, i.e., if we ignore V_{known} and E_{direct}. It suffices to employ a cycle-detection algorithm, and—if no cycles are present—return the input constraint graph as our solution: all path edges can become direct subtyping edges. If our input graph contains a cycle, then our problem is unsolvable. If not, our solution would probably contain some redundant edges (i.e., edges connecting nodes that are already connected by another path) that we could prune to minimize our output. In either case, our solution would be valid w.r.t. our constraints.

The problem becomes much more interesting when we take V_{known} into account. The source of the difficulty is the combination of cycle detection with nodes whose outgoing edge set cannot be extended. Consider first the pattern of Figure 4.3.

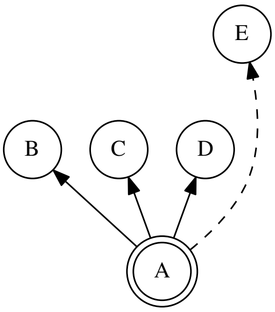


Figure 4.3: In any solution of these constraints, either B or C or D have to be ordered below E, since no new outgoing edges can be added to A and the path constraint to E needs to be satisfied.

This pattern is a basic instance of interesting reasoning in the case of multiple inheritance. We have $A \in V_{known}$ such that $(A, B), (A, C), (A, D) \in E_{direct}$ and $(A, E) \in E_{path}$. We cannot, however, satisfy the path ordering constraint by adding edges to the known node A. Therefore the output must have one of B, C, D ordered below E. We

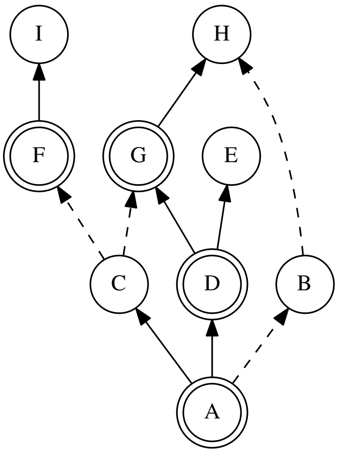


Figure 4.4: The phantom projection set of A is $\{C, E, H\}$. In order to satisfy path-edge (A, B) we can either add a path-edge (C, B), (E, B), or (H, B). The last one creates a cycle.

refer to the set of $\{B, C, D\}$ as the *projection set* of node A, which is a more generally useful concept.

Definition 4.1. *Projection Set.* A node $t \in V_{phantom}$ belongs to the *projection set* of a node $s \in V_{known}$ iff t is reachable from s through a path of *direct* edges.

$$\textit{proj}(s) \equiv \{t \in V_{phantom} : (s,t) \in {E_{direct}}^{+}\}$$

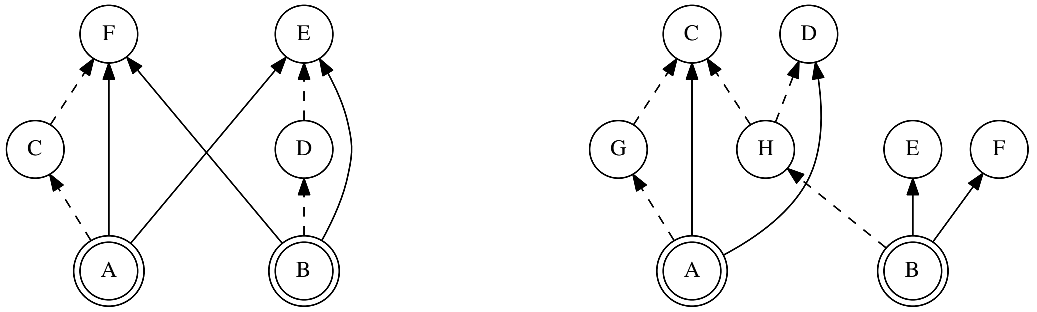
with the $+$ symbol denoting transitive closure.

That is, for each known node we can follow its outgoing direct-edges recursively, ending each path when we reach a phantom node. For instance, in Figure 4.4, the phantom projection set for node A is $\{C, E, H\}$.

Referring again to Figure 4.4, we can see that if H is chosen from the projection set of A in order to satisfy the path-edge (A, B), and therefore edge (H, B) is added, then this would immediately create a cycle because of the existing (B, H) edge. Our algorithm should prevent such a cycle by making the correct choice from the relevant projection set.

Combining this projection set choice with cycle detection leads to interesting search outcomes. Figure 4.5a shows an example of unsatisfiable input. The path edge (B, D) makes either E or F be subtypes of D, and similarly the path edge (A, C) makes either E or F be subtypes of C. Nevertheless, any choice leads to cycles. In contrast, Figure 4.5b shows an input for which a solution is possible, and which we use to illustrate our algorithm.

Algorithm 4.1 solves in polynomial time (an easy bound is $O(|V| \cdot |E|)$) any instance of the hierarchy complementation problem in the multiple inheritance setting. The main part



(a) Unsatisfiable input

(b) Satisfiable input

Figure 4.5: Multiple Inheritance Examples

of the algorithm is function stratify(), which computes a stratification with the property that any constraint edge is facing upwards (i.e., from a lower to a higher stratum). Moreover, this stratification ensures that, for any path-edge (s, t) originating from a known node, there will exist a phantom node p in the projection set of s that is placed lower than t. Given this stratification, it is easy to compute the final solution (as in function solve()). To satisfy any such path-edge (s, t), we add a direct-edge from p to t. This respects our invariant of all edges facing upwards, thus ensuring that no cycles will be present in our solution.

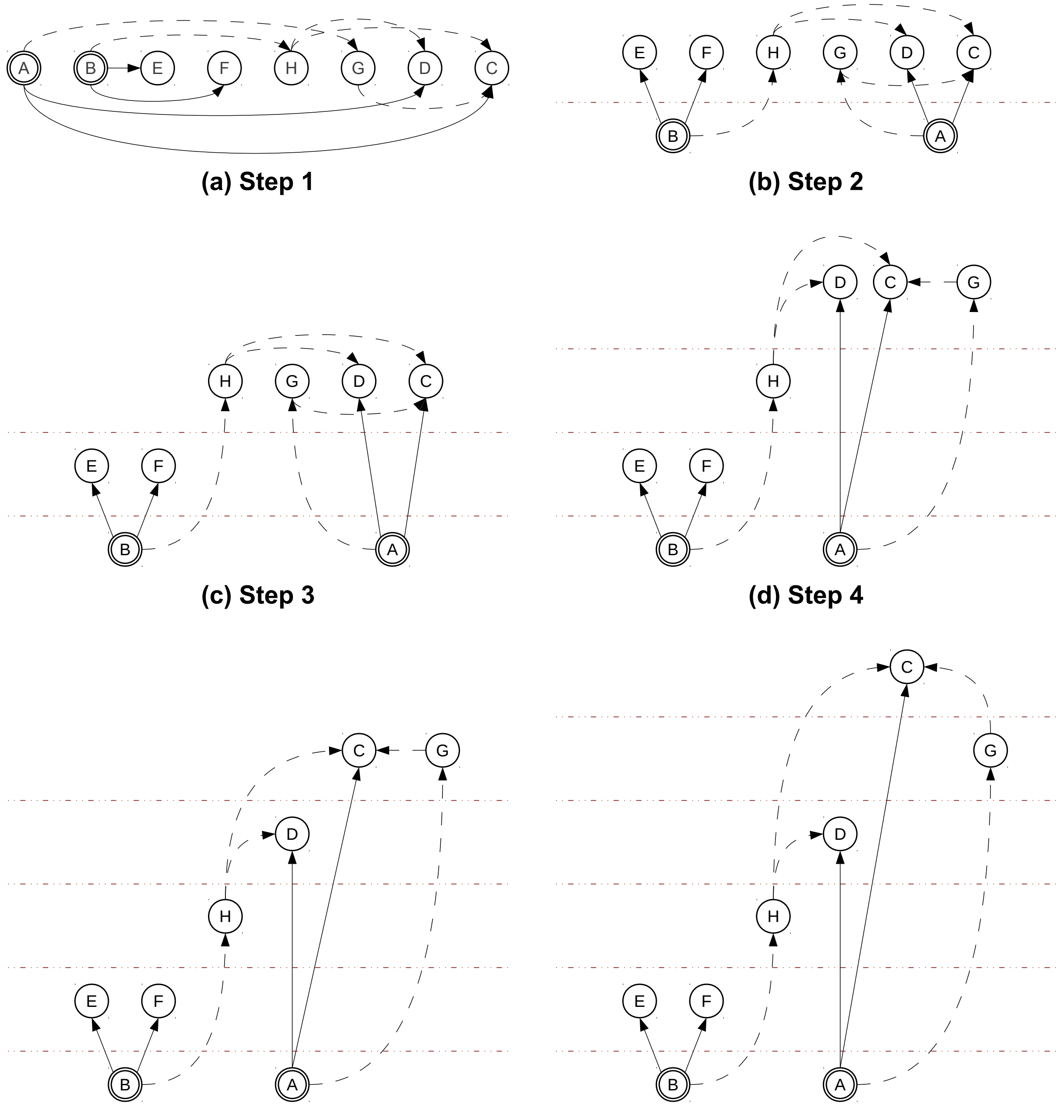
Function stratify() starts from a single stratum, and then computes on each iteration a new stratification, S_{i+1}, by building on the stratification of the previous step, S_i, and advancing some nodes to a higher stratum in order to satisfy constraints. This process is repeated until we converge to the final stratification, which will respect all of our constraints (line 21). If no new node converges at some step (i.e., all nodes that reached a certain stratum advance to the next), then we can be certain that we are dealing with unsatisfiable input, and terminate, thus avoiding infinite recursion (line 23). The nodes to be advanced at each step are determined at line 18, which captures the essence of the algorithm. The new stratum of a node t will be either (i) its current stratum, (ii) the stratum right above the source of an edge (s, t), or (iii) the one right above the *lowest* projection node of the source of a path-edge (s, t) originating from a known node—whichever is higher. These conditions raise the stratum of a node to the minimum required to satisfy the natural constraints of the problem, per our above discussion: edges in the solution should be from lower to higher strata.

Figure 4.6 presents an illustration of the algorithm's application to the example of Figure 4.5b. The sets $\{C, D\}$ and $\{E, F\}$ are the projection sets of nodes A and B respectively. At the first step, all nodes will be placed in the lowest stratum. Note that, at this point, all nodes could be placed in topological order: Figure 4.6a is perfectly valid as the output of a topological sort. However, this is not a solution by our standards, since node A cannot satisfy the edge to G because both of its projection nodes, D and C, are placed

Algorithm 4.1 Multiple-inheritance solver

```
function solve($G = (V, E)$)
    $S \leftarrow$ stratify($G$)
    $U \leftarrow \{(s,t) \in E_{path} : s \in V_{known}\}$
    $E_S \leftarrow E \quad U$
    for all $(s,t) \in U$ do
        let $p \in$ proj$(s) : S[p] < S[t]$                         ▷ such $p$ always exists
        $E_S \leftarrow E_S \cup \{(p,t)\}$
    end for
    return $E_S$
end function
function stratify($G = (V, E)$)
    $U \leftarrow \{(s,t) \in E_{path} : s \in V_{known}\}$
    for all $t \in V$ do
        $S_0[t] \leftarrow 0$
    end for
    for $i = 0 \rightarrow |V| - 1$ do
        for all $t \in V$ do
            $S_{i+1}[t] \leftarrow \max \left\{ \begin{array}{c} S_i[t] \\ \max\limits_{(s,t) \in E} \{1 + S_i[s]\} \\ \max\limits_{(s,t) \in U} \{1 + \min\limits_{p \in \textit{proj}(s)} \{S_i[p]\}\} \end{array} \right\}$
        end for
        if $\forall v \in V : S_{i+1}[v] = S_i[v]$ then
            return $S_i$                                             ▷ reached a fixpoint
        else if $\forall v \in V : S_{i+1}[v] = S_i[v] \Rightarrow S_i[v] = S_{i-1}[v]$ then
            break                                        ▷ no progress made on this step
        end if
    end for
    return error                                         ▷ unsolvable constraint graph
end function
```

after G. Adding an edge from either one would be subject to creating cycles. At the next step, our algorithm advances every node except A and B, since all are edge targets. At step 3, things become more interesting. Nodes D, C have to be advanced by the same criterion, since node H contains edges to both, and they all reside in the same stratum at step 2. However, nodes H and G have to be advanced for a different reason, since they are targets of path-edges originating from known nodes, namely A and B, whose projections ($\{D, C\}$ and $\{E, F\}$ respectively) were on the second stratum during the previous step. At step 4, this condition ceases to exist for node H, since nodes E, F have "stabilized" at a lower stratum. This in turn causes node D to stabilize at step 5. At step 6, G can also stay put, since it is in a higher stratum than the lowest projection of A, namely D. No nodes are advanced at step 7 (which is omitted in Figure 4.6), thus signifying that our stratification has successfully converged to its final form. It is therefore



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

(e) Step 5

(f) Step 6

Figure 4.6: An example of the stratification produced by the multiple-inheritance solver for Example 4.5b.

simple to compute a solution, by adding edges (H, D), (H, C), (G, C), (D, G) and either (F, H) or (E, H) to the direct-edges (A, C), (A, D), (B, E), (B, F). This set of edges will constitute our final solution.

It is also easy to see that our algorithm would soundly detect that the example of Figure 4.5a is unsatisfiable. At the first step, only known nodes A, B would remain in the lowest stratum, but on the next iteration all remaining nodes would advance again, thus triggering the condition of failure (line 23), since an iteration passed with no progress made.

A detailed proof of the correctness of our algorithm can be found in Appendix A.1.

4.4 Hierarchy Complementation for Single Inheritance

The problem for a single inheritance setting has a very similar statement as in the earlier case of multiple inheritance, but markedly different reasoning intricacies and solution approaches, due to a newly arising constraint: every class in this setting can only have a single parent.

Formally, our problem is modeled in much the same way as before. Our input is again a directed graph $G = (V, E)$, with two disjoint sets of nodes $V = V_{known} \dot{\cup} V_{phantom}$ and two disjoint sets of edges $E = E_{direct} \dot{\cup} E_{path}$, where $E_{direct} \subseteq V_{known} \times V$. The difference is that the output of our algorithm should be a directed *tree* (instead of a DAG), $G_T = (V, E')$, such that the same conditions as in the earlier case are satisfied:

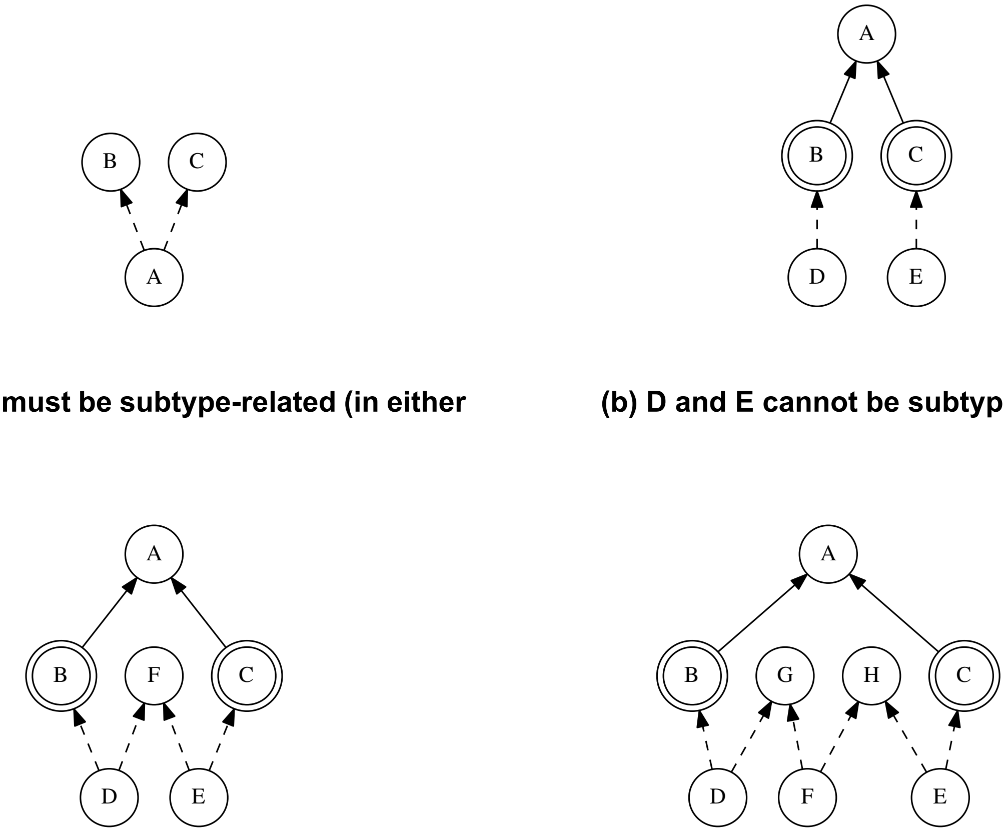
1. $\forall v_s \in V_{known} : (v_s, v_t) \in E' \Leftrightarrow (v_s, v_t) \in E_{direct}$
2. $(v_s, v_t) \in E_{path} \Rightarrow$ there is a path from v_s to v_t in G_T

Without loss of generality, we assume that there exists a "root" node $n_r \in V_{known}$ that is a common supertype for all of our types. If no such type exists, we can create an artificial one, by adding extra constraint edges. In this way, we can be certain that computing a graph with a single outgoing edge for all nodes (but one) will form a tree instead of a forest.

The problem is quite hard in its general setting. There are several patterns that necessitate a complex search in the space of possibilities. Figures 4.7a-4.7d show some basic patterns that induce complex constraints. All nodes reachable from a single one need to be linearly ordered (Figure 4.7a shows the simplest case). This requires computing an ordering (i.e., guessing a permutation) of these nodes. Other constraints can render some of the permutations invalid. The basic pattern behind such restrictions is that of Figure 4.7b: there are hierarchies that cannot be related. Combining the two patterns suggests that there needs to be a search in the space of permutations for a valid one: Figures 4.7c and 4.7d show some simple cases.

Composing such constraints into more complex hierarchies gives an idea of the difficulty of the search involved. Figure 4.8 shows an example where it is hard to see without complex reasoning which of the E, F, G nodes have to be placed above A and which cannot.

Clearly the problem can be modeled as a constraint satisfaction problem instance, where $V_{phantom}$ is our set of variables and V is the domain of values (representing the variable's direct supertype). The path-edges and the absence of cycles constitute our constraints. This requires an exponential search in the worst case. Indeed, our implementation performs precisely such an exhaustive search, but with a heuristic choice of nodes so that the search tries to satisfy the constraints introduced by the patterns in Figures 4.7a and



(a) B and C must be subtype-related (in either direction).

(b) D and E cannot be subtype-related.

(c) A has to be a subtype of F.

(d) A has to be a subtype of either G or H.

Figure 4.7: Single Inheritance Basic Patterns

4.7b—i.e., the pattern of Figure 4.7a is identified, all induced permutations are tried, and the pattern of Figure 4.7b is used to prune them eagerly, instead of waiting to detect failure later.

Most importantly, our approach provides special handling for a simple but practically quite common case. In this special case, there is a polynomial algorithm for solving the problem and exhaustive search is avoided.

Simplified setting: No direct-edges to phantom nodes. It is easy to solve the problem in the case that there are no direct edges from known nodes to phantom nodes. Since we are in a single-inheritance setting, this means that no class in the known part of the program has a superclass in the complement that we are trying to produce. In this case, we have that $E_{direct} \subseteq V_{known} \times V_{known}$. The extra condition allows us to employ a fast polynomial time algorithm. This interesting case of our problem is very common in practice. Intuitively, the ease of dealing with this case stems from avoiding the search in the space of permutations when the input contains patterns such as those in Figure 4.7c:

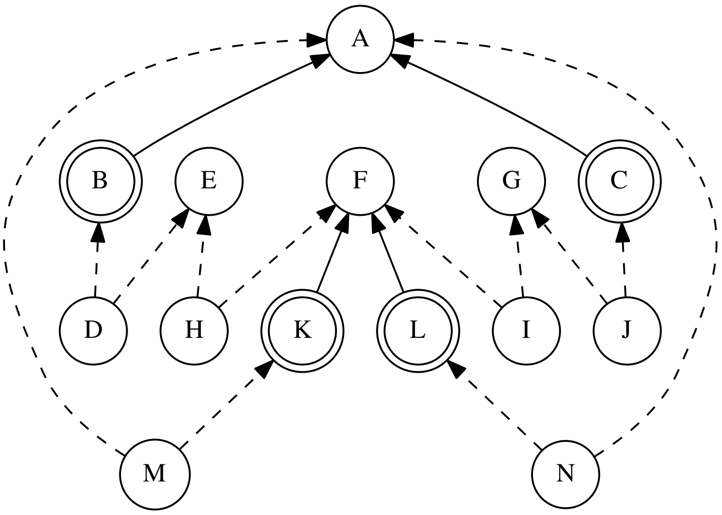


Figure 4.8: Harder composite example of single-inheritance constraints. The (undirected) path from B to C through E, F, G implies that $(A <: E) \vee (A <: F) \vee (A <: G)$. However, since F is the first common known supertype of M and N, and A just a supertype of both, $F <: A$, and thus $(A <: E) \vee (A <: G)$.

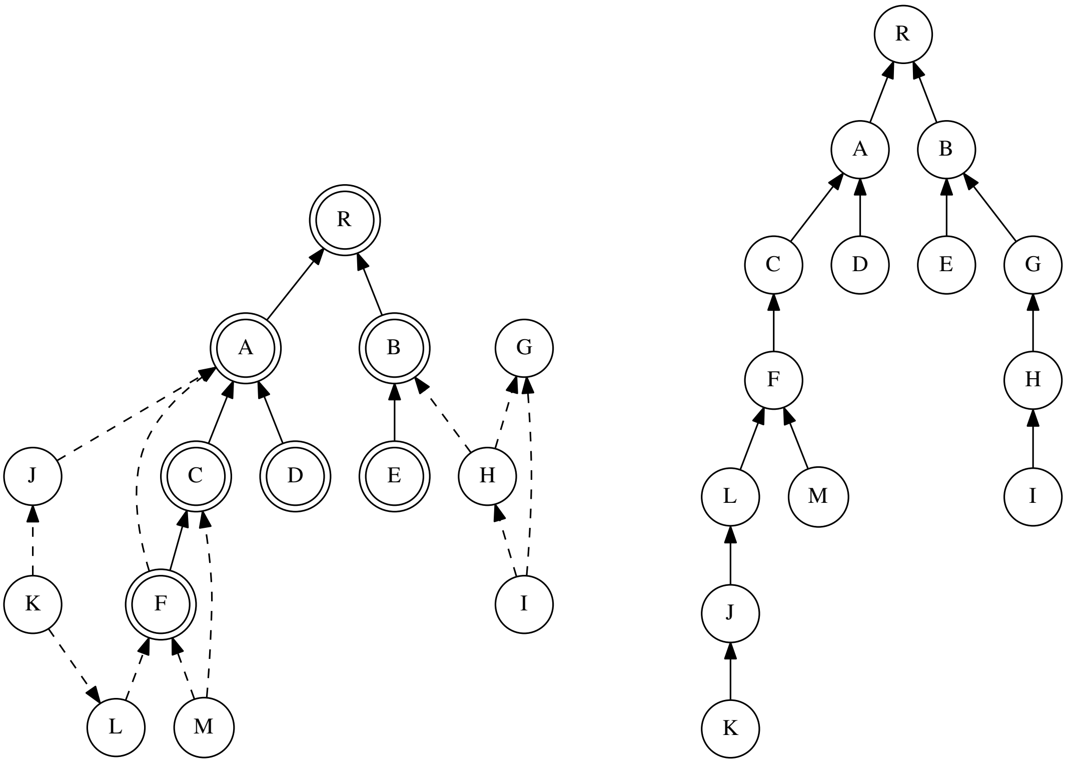
if two permutations have elements in common (e.g., the permutation of B and F, and that of F and C in Figure 4.7c) they cannot include nodes that are guaranteed to be subtype-unrelated (such as B and C in this example) and all unknown nodes have to be below the known ones in any solution.

Algorithm 4.2 first removes path-edges originating from known-nodes, after verifying that the corresponding paths indeed exist. It then uses union/find data structures to compute connected components of phantom nodes, while treating path-edges as *undirected* edges: anything connected through such edges can safely end up in a single linear ordering. Then, for each phantom undirected connected component, it computes the lowest known-node to serve as the first-common-supertype of all of this component's phantom nodes. Note that when two known-nodes are reachable by two phantom nodes of the same connected component (in the phantom subgraph), then one of them ought to be a supertype of the other, or else no solution can exist in a single inheritance setting. This condition is captured in line 24. After the first common (known) supertype for every connected component has been computed, a mere topological sort, i.e. placing all relevant nodes in a total order, is enough to satisfy all of this component's constraints. This may introduce many superfluous edges in the solution: these edges are not actually required by our constraints (since a topological order is a total order). In practice, we produce a partial order by using a variant of topological sort that generates a tree instead of a list as its result, but a full topological sort also satisfies the correctness requirements of the algorithm. (We return to the topic of why we actually want a weaker ordering in Section 4.5.)

Algorithm 4.2 Single-inheritance solver for strictly known direct-supertypes

```
function solve($G = (V, E)$)
    let $R$ be the "root" node of $V$
    let $S$ be the tree of known nodes $(V_{known}, E_{direct})$
    for all $(s, t) \in E_{path} : s \in V_{known}$ do
        if $\nexists$ path $s \rightsquigarrow t$ in $S$ then
            return error (unsatisfiable constraint)
        end if
        $E_{path} \leftarrow E_{path} \quad \{(s, t)\}$                    ▷ remove already satisfied edge
    end for
    for all $v \in V_{phantom}$ do
        makeSet($v$)                                    ▷ create single-element disjoint sets
    end for
    for all $(s, t) \in E_{path} : t \in V_{phantom}$ do
        union($s$, $t$)                        ▷ merge two connected (phantom) components
    end for                               ▷ result: undirected connected components (UCCs)
    for all $v \in V_{phantom}$ do
        $k \leftarrow$ find($v$)
        top$[k] \leftarrow R$                                                ▷ initially "root"
    end for                             ▷ init UCC's lowest common known superclass (LCS)
    for all $(s, t) \in E_{path} : t \in V_{known}$ do                    ▷ must be $s \in V_{phantom}$
        $k \leftarrow$ find($s$)
        if $\exists$ path $t \rightsquigarrow$ top$[k]$ in $S$ then
            top$[k] \leftarrow t$                           ▷ lower superclass found, update LCS
        else if $\nexists$ path top$[k] \rightsquigarrow t$ in $S$ then
            return error (unsatisfiable constraint)
        end if
    end for
    for all $k \mapsto v$ in top do                                ▷ for each UCC and its LCS
        $U \leftarrow \{(s, t) \in E_{path} : t \in V_{phantom} \wedge \text{find}(s) = k\}$
                                      ▷ directed subgraph of original over nodes of this UCC
        $L \leftarrow$ a topological order of $U$                                ▷ linearize subgraph
        $hd \leftarrow$ the top node of $L$
        $S \leftarrow S \cup L \cup \{(hd, v)\}$
    end for
    return $S$
end function
```

In the example of Figure 4.9, Algorithm 4.2 first checks and removes the (F, A) path-edge. Then the phantom nodes are divided in the following phantom connected components: $\{G, H, I\}$, $\{J, K, L\}$, and $\{M\}$. The first common known supertype for each component is B, F, and F respectively. Each component is then linearized, which generates the following complete orders that are appended to the output: $I <: H <: G <: B$, $K <: J <: L <: F$, and $M <: F$.



(a) Constraint Graph **(b) Solution**

Figure 4.9: Algorithm 4.2 - Example.

4.5 Single Inheritance, Multiple Subtyping: Classes and Interfaces

It is easy to combine the single- and multiple-inheritance approaches of the last two sections in the context of a language that has single inheritance but multiple subtyping. It is a common case for strongly-typed languages to allow multiple inheritance only for a subset of types. Java and C# interfaces [44, 54], and Scala traits [100] are such examples.

In order to support such a separation, we have to introduce a new dimension to our problem that can be simulated as a graph coloring variant. Each node in V can be assigned a color denoting its inheritance type. A *black* node can have many direct supertypes (i.e., multiple inheritance), while a *white* node can only have one (i.e., single inheritance). We will use the terms “white node” (resp. “black node”) and “class” (resp. “interface”) interchangeably.

Note that, initially, our input may not fully determine the final color for each of its types. Thus, we have to introduce a new color (*grey*) to refer to the subset of nodes whose color is yet undetermined. In the end, our solution should soundly determine a safe color (black or white) for each of the (grey) input nodes, so that no constraints of the verifier will be violated.

Therefore, our solution in this new setting is a synthesis of a single inheritance and a multiple inheritance solution. That is, the output of our algorithm should be a *DAG* that satisfies the same conditions as those in the multiple inheritance setting, $G_S = (V, E')$, and a function $f_c : V \rightarrow \{black, white\}$, such that the restriction of G_S to $\{v \in V : f_c(v) = white\}$ (i.e., white nodes) is a tree.

To safely decompose our problem into two different subproblems (one for single and one for multiple inheritance), we assign colors to all nodes as a preprocessing step. There are two kinds of constraints that lead to restricting the colors of a node. First, we have local constraints: we may get a node color from the initial input—i.e., an observed bytecode instruction (such as `invokeinterface`) may directly restrict the color of a phantom type. (More constraints of this form are discussed in Section 4.6.) Second, we may get transitive constraints, due to restrictions on subtyping. Interfaces can only subtype interfaces (except for the `Object` class in Java). This leads to two types of transitive constraints: If a black node s has a path to node t, then t must also be black (interfaces can only extend interfaces). Symmetrically, if a node s has a path to a white node t, then s must also be white (classes can only be extended by other classes).

Furthermore, phantom nodes with no color constraints can be safely assumed to be interfaces (black), for maximum flexibility in solving other constraints. It is always easier to satisfy a given set of constraints in a multiple inheritance setting instead of a single inheritance setting, since the conditions of single inheritance are stricter (a tree is a DAG).

As a result of the above observations, we can color all nodes by applying local or transitive constraints to the original input before solving a single and a multiple inheritance hierarchy complementation problem separately. That is, we can follow every possible path from any node whose color has already been set and mark the nodes we find along the way accordingly. The color of our source node determines the direction of movement (i.e., from white source nodes, we have to go backwards). When this process is over, we can assign the color black to all remaining undetermined (in terms of color) nodes. An example of this process can be seen in our earlier Figure 4.2. Once we have assigned a *black-or-white* color to every node, we can split our constraint graph into two subgraphs by isolating white-to-white edges (and feeding them to a single inheritance solver). After we have determined our class hierarchy, we can proceed with satisfying the rest of the edges using multiple inheritance rules.

The key to this approach is that the single inheritance solver does not need the output of the multiple inheritance solver to compute a solution, and vice versa. All we need to ensure (for the multiple inheritance solver) is that we take into account class supertypes that are reachable through direct edges of a known class when determining the class's *projection set*. Thus, the class/interface decomposition indeed produces two independent subproblems that can be solved separately. The composition of the two solutions will certainly not create any cycles, if its two subparts do not contain any. If that was not the case, then there would be a cycle that contained at least one class and one interface, which is impossible since no interface can be a subtype of a class (other than `Object`) in

Java.

As for our arbitrary choice of defaulting undetermined nodes to interfaces, suppose that a solution exists if a subset U of those undetermined nodes were treated as classes. We could then transform this solution to another one where these nodes were interfaces instead. The single inheritance solution could be produced by replacing each node in U with its parent (in the former single inheritance solution), w.r.t. its incoming edges, and then removing it, until no nodes in U were present. This process would still satisfy all constraints on the remaining class nodes. A multiple inheritance solution also exists. Consider the union of the former multiple plus single inheritance solution. The result is a DAG that respects all of the multiple inheritance setting constraints. Again, we can erase any edges to class-determined nodes (i.e., all class nodes that are not in U) in a way that all subtype relations involving the rest of the nodes remain unaltered, i.e., by iteratively replacing an edge to a class-determined node with edges to all of its direct supertypes, until no edges to class-determined nodes are left. This process would yield a valid multiple inheritance solution that can be safely combined with the single inheritance one. Therefore, marking undetermined nodes as interfaces does not affect the outcome of our algorithm, i.e., no solution will be found if and only if no solution existed.

4.6 Implementation and Practical Evaluation

We next discuss practical aspects of our implementation. First, we consider the *program analysis* part of our work, which solves the problem of producing complements of a partial Java program by appealing to the solver of the class hierarchy complementation problem. Subsequently, we present experiments applying our JPhantom tool to real programs.

4.6.1 JPhantom Implementation

JPhantom is a practical and scalable tool for program complementation, based on the algorithms we have presented in this chapter.[5] JPhantom uses the ASM library [21] to read and trasform Java bytecode. Given a jar file that contains phantom references, it produces a new jar file with dummy implementations for each phantom class. The resulting jar file satisfies all formal constraints of the JVM Specification [83]. We give a brief explanation of the different stages of computation for the analysis of an input jar file by JPhantom.

JPhantom execution consists of the following steps. It (1) performs a first pass over the jar contents in order to recreate the existing class hierarchy (type signatures only) and store the field and method declarations of the contained classes, then (2) makes a second pass to extract all phantom references and store the full class representations. A third pass (3) extracts all relevant type constraints, before (4) they are fed to JPhantom's hierarchy complementation solver, which computes a valid solution, if such a solution is possible. At this point, we can proceed to (5) bytecode generation, where we create new class files

[5] JPhantom is available online at `https://github.com/gbalats/jphantom`.

Opcode	*Types*	*Stack Types*	*Constraints*
`AASTORE`		$a : E[]\ \ i : int\ \ v : V$	$V <: E$
`ARETURN`		*obj* : S	$S <: R_m$
`ASTORE`	T	*obj* : S	$S <: T$
`ATHROW`		*obj* : S	$S <:$ "`java.lang.Throwable`"
`GETFIELD`	$T.F$	*obj* : S	*isClass*$(T) \wedge S <: T$
`PUTFIELD`	$T.F$	*obj* : $S\ \ v : U$	*isClass*$(T) \wedge S <: T \wedge U <: F$
`PUTSTATIC`	$T.F$	$v : U$	*isClass*$(T) \wedge U <: F$
`INVOKEINTERFACE`	$T.(\overline{A})R$	*arg*$_0 : S_0$ *arg*$_1 : S_1$ ...	*isIface*$(T) \wedge S_0 <: T$
`INVOKEVIRTUAL`	$T.(\overline{A})R$	*arg*$_0 : S_0$ *arg*$_1 : S_1$ ...	*isClass*$(T) \wedge S_0 <: T$
`INVOKESPECIAL`	$T.(\overline{A})R$	*arg*$_0 : S_0$ *arg*$_1 : S_1$ ...	*name* $=$ "`<init>`" $\Rightarrow$ *isClass*$(T) \wedge S_0 <: T$
`INVOKESTATIC`	$T.(\overline{A})R$	*arg*$_1 : S_1$ ...	*isClass*(T)
`INVOKE*`	$T.(\overline{A})R$	$($*arg*$_0 : S_0)$ *arg*$_1 : S_1$ ...	$S_i <: A_i, \forall i = 1, \ldots$

Figure 4.10: *Generated Bytecode Constraints.* At this point, our analyzer has already computed the (sets of) types for every stack and local variable at every point of execution (bytecode in method). For simplicity, we assume that each set of reference types contains a single element (3rd column). Each bytecode may involve some declared types (2nd column) by references in the constant pool or by entries in the local variable table (if such exists). Also, let R_m be the containing method's return type.

for our missing (phantom) types. Finally, we compute method bodies to add to each type. For instance, when the solver determines that a phantom-class type X must implement an interface type Y, all missing methods of Y should be added to X, so that the resulting bytecode is valid. After all such methods have been computed, they are added in the last (6) step of execution.

Phantom references include references to missing classes, as well as references to missing fields and methods. Note that both phantom and existing classes may have references to missing members, since there are cases of existing classes calling a method or referencing a field declared in one of their phantom supertypes. JPhantom detects all such references and adds the relevant missing declarations to its output. If a member is missing from a phantom class, we add it directly to that class as part of JPhantom's output. Otherwise, if a member is missing from an existing class, we add it to an appropriate phantom supertype in its projection set instead. We encode these declarations as additional constraints over the missing classes, generated in the second step of JPhantom's execution. It suffices to use the existing class hierarchy and declared members (step 1), to perform member lookup for the purpose of determining if a member is missing and where it should be added.

The most interesting aspects of the above steps have to do with analyzing the bytecode to produce the constraints (step 3) used as input to the hierarchy complementation algorithm. In order to extract type constraints, we have to simulate a symbolic execution of

Java bytecode by following every possible execution path, while computing the types of stack and local variables. This is necessary because, in general, bytecodes receive some untyped arguments whose types we need to infer, in order to extract our constraints. This process is analogous to Pass 3 [83, Section 4.9.2] of the bytecode verification process.

When computing such type information for stack and local variables, there are points where we have to merge two different paths of execution. That is, the two paths may map the same variable to different types, in which case we have to merge two different types into a new one. Typically, when merging two types A, B the resulting type is the first common superclass of A and B. In Java, there always exists such a common superclass since every reference type (interfaces included) is a subtype of `java.lang.Object`.

In our case, however, since we do not have the complete type hierarchy at the time of constraint extraction, we cannot compute the first common superclass for any two nodes. This is why we apply the alternative technique of storing *sets of reference types*, as presented in alternative verifier designs [127]. I.e., our bytecode analyzer stores not a single type, but a set of types for each variable at every point of execution. Figure 4.10 lists the constraints that may be generated by the analyzer for certain bytecodes. Since our analyzer generates constraints due to widening reference conversions, it is easy to see that storing a set of reference types fits our needs well. Consider the following case:

```
class Test {
  A foo(B b, C c) {
    return (b == null) ?
      c : b;
  }
}
```

```
A foo(B, C);
  Code:
   0:       aload_1
   1:       ifnonnull 8
   4:       aload_2
   5:       goto 9
   8:       aload_1
   9:       areturn
```

Our analyzer will compute that the stack contains a single item with type $\{B, C\}$, before position 9, which is the outcome of merging the two different execution paths. Let us also assume that A and B are phantom classes. This toy example demonstrates why we have chosen to store sets of types, since we cannot compute the first common superclass of B, C. After our tool has completed the analysis of method `foo()`, it will generate (because of the `areturn` instruction) the constraint $B <: A \wedge C <: A$.

4.6.2 JPhantom in Practice

We next detail a typical usage scenario of JPhantom, together with the complications that would arise in its absence.

Consider performing a static analysis of a large Java program. For instance, the Doop framework [20, 66] integrates points-to analysis with call-graph construction, computation of heap object points-to information, and various client analyses (escape analysis, virtual call elimination, class cast elimination). Doop uses Soot as a front-end and post-processes the facts generated by Soot. When faced with an incomplete program, the user of the analysis is faced with various issues. To illustrate and quantify them we created a synthetic incomplete program, *antlr-minus*, by artificially subtracting parts of the antlr parser generator jar. (We also use *antlr-minus* as a performance benchmark in the next section.)

A user that tries to analyze *antlr-minus* will encounter the following issues:

– *Crash in Soot.* Earlier versions of Soot, e.g., Soot 2.3.0, will often crash when trying to analyze a program that contains phantom references. Soot provides the `-allow-phantom` flag, as a command-line option that the user can set to inform Soot that its input contains phantom references, and that Soot should try to handle them instead of terminating with an error. However, for several Soot versions the flag is not sufficient to prevent Soot from crashing in some cases.

– *Need to handle phantom references in the client of Soot.* Although the latest version of Soot (2.5) has increased its tolerance of phantom references to the point where it no longer crashes, this only prevents against crashes in Soot itself and does not yield any meaningful handling of phantom references. The problem is propagated to the client of Soot. The client analysis (any external tool that uses Soot) now needs to have special-case code for handling phantom classes, in whichever way makes sense to the client. There is no evident general-purpose solution to fixing the Soot output for *any* client without adding code to deal with phantom references, essentially duplicating what JPhantom does already. In our case, if the Doop front-end that reads Soot information tried to just handle phantom references as regular references, it would crash (as we have confirmed experimentally), since it needs to encode for every variable its full type information (e.g., member methods). (The Doop front-end does not crash in practice because it handles phantom references specially, by merely ignoring them, as we discuss next.) In contrast, JPhantom allows any tool completely unaware of phantom references, such as the Doop front-end, to be able to run without unexpected behavior, as long as its input is first transformed by JPhantom.

– *Incompleteness when analyzing with* Doop*.* The Doop front-end is coded so that it avoids crashes but only at the cost of completely ignoring any reference to a phantom class. A method that takes phantom types as arguments is just skipped. This handling has been the default for Doop since its original version. Unfortunately, this leads to incompleteness in the resulting analysis performed by Doop.

  Figure 4.11 presents a Venn diagram over the sets of reachable methods as computed by Doop for three different inputs: (i) the original *antlr* jar, (ii) our synthetic benchmark, *antlr-minus*, and (iii) the output of JPhantom after analyzing *antlr-minus*, that is, a transformed version of the *antlr-minus* jar with no phantom references. The original jar yields $52,357$ reachable methods, out of which only $42,337$ are detected in the presence of phantom references (*antlr-minus*), without using JPhantom. Additionally, phantom references introduce 500 false positives that correspond to non-existing methods.[6] After employing JPhantom to alleviate the effect of phantom references, Doop manages

[6] It may seem surprising that eliminating code can introduce new (falsely) reachable methods. The rea-

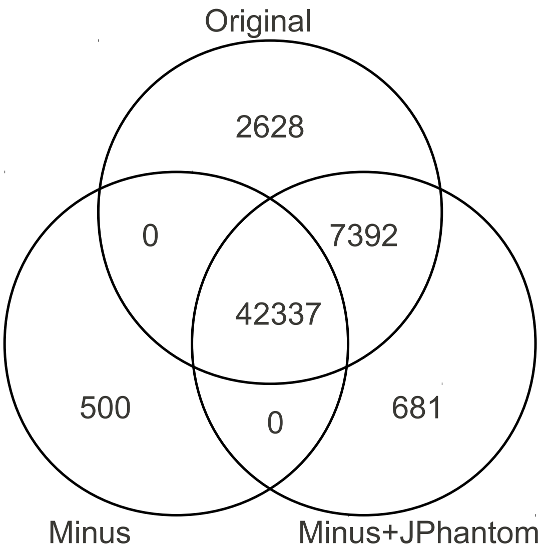


Figure 4.11: A Venn diagram that shows how three different sets of reachable methods relate to each other. These three sets—(i) *Original*, (ii) *Minus*, and (iii) *Minus+JPhantom*—correspond to the outcomes of analyzing (i) the *antlr* jar (original), (ii) the *antlr-minus* jar (subset of the original jar), and (iii) the *antlr-minus* jar after being transformed by JPhantom, respectively. The sets are not drawn to scale: the size of each subset is indicated only by the number in it.

to find $7,392$ of the $10,020$ missing reachable methods, resulting in 73.77% recall (over the missing methods alone, or 95% over all methods). The false positives of directly analyzing *antlr-minus* disappear but 681 new ones emerge, yielding a precision of 98.65%. Even so, this allows us to discover almost 3 out of every 4 missing reachable methods, which originally constituted 19.14% of the total reachable methods, dropping this percentage to just 5.02%.

It is notable that this high recall is achieved although recall could, in principle, be arbitrarily low. JPhantom is trying to guess the structure of missing code with as much information as remains in existing code—but this could be a tiny fraction of the missing information. The missing code could be hiding a huge portion of the application, and expose only a handful of phantom types on the unknown/known code boundary.

This finding supports our thesis statement: static analysis is improved (in terms of completeness[7]), by recovering missing structural information of phantom classes, via inference, by tracking the use of types in the program.

In summary, JPhantom avoids problems with crashes when encountering phantom references as well as incompleteness when phantom references are merely ignored. In practice, it is effective in discovering large parts of the interface for missing methods and

son is that a non-existent method m in class C may be reported reachable, based on method signature information on the call-site alone, whereas in the original code the true reachable method m was defined in a, now missing, superclass S of C, and not in C.

[7] Improving completeness essentially means improving empirical soundness, which is the metric we used in Chapter 3, where we compared the statically computed call-graphs against dynamic call-graphs of actual executions.

the produced complement respects the requirements of the Java VM verifier, i.e., the most fundamental Java well-formedness rules for types.

4.6.3 Performance Experiments

We use a 64-bit machine with a quad-core Intel i7 2.8GHz CPU. The machine has 8GB of RAM. We ran our experiments using JDK 1.7 (64-bit).

Our benchmarks consist of (1) *antlr*, a parser generator, (2) *antlrworks*, the GUI Development Environment for antlr, (3) *c3p0*, a library that provides extensions to traditional JDBC drivers, (4) *jruby* and (5) *jython*, implementations of Ruby and Python programming languages respectively that run on top of the JVM, (6) logback-classic and (7) logback-core, two modules of the *logback* logging framework, (8) *pmd*, a Java source code analyzer, (9) *postgres*, the PosgreSQL JDBC driver, (10) *sablecc*, a compiler generator, and (11) *antlr-minus*, a synthetic benchmark described in the previous section. Every benchmark is just a jar file that serves as JPhantom's input, which then detects all phantom references and generates the complemented jar.

We encountered most of these benchmarks while working on static program analysis with the Doop framework [20, 66]. (As already mentioned, Doop has been the vehicle for our reflection analysis of the previous chapter, as well.) For many of the benchmarks it was, upon original encounter, an unexpected discovery that they could not be analyzed due to dependencies to unknown classes in other libraries.

Figure 4.12 presents input features and the running time of JPhantom for each of our benchmarks. The first column is the name of the benchmark. The second column is the size of the output, i.e., the complemented jar, divided into the original size of the input (benchmark) and the size of the complement itself (i.e., the size of the generated phantom classes). The third and fourth columns are the number of phantom classes and constraints detected respectively. The last column is the running time of JPhantom, including the time to analyze the input, compute a type hierarchy that respects all of the constraints detected, create the phantom classes with the required members and supertypes, augment the input jar and flush its contents to disk.

Even the largest benchmark (jruby) takes seconds to complete. Moreover, the size of the input is highly correlated with the running time of JPhantom and much less correlated with the number of constraints. This suggests that most of the time is spent on reading and analyzing the input, rather than on the type hierarchy solver. The only slight exception is the logback-classic benchmark, which requires about 1.8 seconds to complete despite its small size. This is due to the large number of phantom classes and constraints this benchmark produces, which is to be expected since it is built on top of logback-core (which is not supplied as part of the input). This practice of creating such a strong dependency is probably justified by logback's design. The framework implements the SLF4J (Simple Logging Facade for Java) protocol, which acts as a common interface for a variety of logging frameworks, and hides the actual framework (called *binding*) to be used underneath. From both logback-classic and antlr-minus we can see that JPhantom scales well as the

Input jar	*Size*	*Phantom*	*Constraints*	*Time*
antlr	3.3M + 0.7K	1	2	$4.82s$
antlrworks	3.5M + 2.2K	5	7	$6.11s$
c3p0	597K + 1.8K	4	2	$2.05s$
jruby	19M + 5.9K	16	20	$13.70s$
jython	2.5M + 4.0K	8	9	$3.26s$
logback-classic	247K + 55K	148	212	$1.76s$
logback-core	358K + 7.9K	22	22	$1.61s$
pmd	1.2M + 11K	28	36	$2.62s$
postgres	499K + 0	0	0	$1.95s$
sablecc	306K + 2.3K	5	8	$1.59s$
antlr-minus	3.2M + 17K	37	103	$5.82s$

Figure 4.12: Results of experiments.

number of constraints increases.

To see the constraints and their solution for a benchmark instance, consider the list below, which is the actual execution output of a JPhantom run on the jruby benchmark:

```
Phantom Classes Detected:                                                    [constraint]

org.apache.tools.ant.BuildException                                       must be a class
org.apache.tools.ant.Task                                                 must be a class
org.apache.tools.ant.Project
org.apache.bsf.util.BSFFunctions                                          must be a class
org.apache.bsf.util.BSFEngineImpl                                         must be a class
org.apache.bsf.BSFException                                               must be a class
org.apache.bsf.BSFManager                                                 must be a class
org.apache.bsf.BSFDeclaredBean                                            must be a class
org.apache.bsf.BSFEngine
org.osgi.framework.Bundle                                            must be an interface
org.osgi.framework.BundleReference
org.osgi.framework.FrameworkUtil                                          must be a class
org.osgi.framework.BundleException
org.osgi.framework.BundleContext                                     must be an interface
java.dyn.Coroutine                                                        must be a class
java.dyn.CoroutineBase

Constraints:

org.apache.bsf.BSFException <: Throwable
org.apache.tools.ant.BuildException <: Throwable
org.osgi.framework.BundleException <: Throwable
org.jruby.embed.bsf.JRubyEngine <: org.apache.bsf.util.BSFEngineImpl
org.jruby.embed.bsf.JRubyEngine <: org.apache.bsf.BSFEngine
org.jruby.ant.RakeTaskBase <: org.apache.tools.ant.Task
org.jruby.javasupport.bsf.JRubyEngine <: org.apache.bsf.BSFEngine
org.jruby.ext.fiber.CoroutineFiber$1 <: java.dyn.Coroutine
org.jruby.javasupport.bsf.JRubyEngine <: org.apache.bsf.util.BSFEngineImpl

Class Hierarchy
* class java.lang.Object
  * class org.apache.bsf.BSFManager
  * class org.osgi.framework.FrameworkUtil
  * class Throwable (implements java.io.Serializable)
    * class org.osgi.framework.BundleException
    * class org.apache.tools.ant.BuildException
    * class org.apache.bsf.BSFException
  * class org.apache.bsf.BSFDeclaredBean
  * class org.apache.bsf.util.BSFFunctions
  * class org.apache.tools.ant.Task
    * class org.jruby.ant.RakeTaskBase
  * class java.dyn.Coroutine
    * class org.jruby.ext.fiber.CoroutineFiber$1
  * class org.apache.bsf.util.BSFEngineImpl (implements
     org.apache.bsf.BSFEngine)
    * class org.jruby.javasupport.bsf.JRubyEngine
    * class org.jruby.embed.bsf.JRubyEngine

Interface Hierarchy
* interface org.osgi.framework.Bundle
* interface org.apache.bsf.BSFEngine
* interface java.io.Serializable
* interface org.osgi.framework.BundleContext
```

It is evident that the final hierarchy respects all of the reported constraints. Some interesting points are that: (i) `org.apache.bsf.BSFEngine` defaults to interface since no constraint determines whether it is actually an interface or a class, (ii) `org.osgi.framework.BundleException` is inferred to be a class since it is a subtype of the class `Throwable`, and (iii) two known classes, `org.jruby.javasupport.bsf.JRubyEngine` and `org.jruby.embed.bsf.JRubyEngine`, used as subtypes of interface `org.apache.bsf.BSFEngine`, add the latter to the supertypes of their phantom projection, `org.apache.bsf.util.BSFEngineImpl`.

4.7 Discussion

We next discuss the problem of hierarchy complementation speculatively, in settings different from ours. The general problem is that of complementing programs so that they respect static well-formedness requirements. Thus, the problem applies to language-level type systems, static analyses (e.g., “complement this program so that it passes this analysis, defined a priori”) and other settings more general than our Java bytecode domain. Indeed, much of our ability to solve the problem effectively has to do with the simple type checking performed by the Java bytecode verifier. The verifier effectively checks monomorphic types, i.e., that a reference to an object is statically guaranteed to refer to memory with the expected layout.

If we were to transpose the problem to the domain of Java source code, the constraints to be derived are richer and more complex than the ones we encountered. The Java language-level type system has intricate requirements relative to overriding, casts, exceptions, and more. By way of example, we discuss some of these complications below.

- Exception handling at the Java language level immediately introduces very powerful constraints for types. The Java language requires that a method that overrides another may throw an exception only if it was already declared to be thrown. Consider two methods:

  ```
  class S {
    void foo() throws A, B {...}
  }

  class C  extends S {
    void foo() throws X, Y, Z {...}
  }
  ```

  The requirement in this case is hard to reason about without an exhaustive search. It can be stated as: "for C.foo to be a valid overriding of S.foo, X, Y and Z have to be subtypes of either A or B." Consider how this rich constraint would affect our ability to solve the hierarchy complementation problem at the source level. Imagine that S is a known class while C, X, Y, and Z are phantom classes. If the language allowed us to infer through observation of other code that C is a subtype of S and that it provides a method "void foo() throws X, Y, Z" then in order to generate a complement we would need to satisfy the following: $\mathtt{C} <: \mathtt{S} \Rightarrow \forall t \in \{\mathtt{X,Y,Z}\} : (t <: \mathtt{A} \vee t <: \mathtt{B})$.

  In contrast, the bytecode verifier only ensures a much simpler constraint: that a type declared to be thrown by a method is a subtype of Throwable.

- A similar kind of constraint at the Java language level is also produced by the overriding rule for return types. Java (5 and above) allows overriding methods to have a covariant return type. That is, the overriding method can declare to return a subtype of the overridden method's return type. Much as in the case of exceptions, this induces complex constraints, especially when combined with search to examine whether a type can be a subtype of another. Consider the following case:

  ```
  interface S {
    R foo();
  }
  // R,X,Y phantom types
  // we know X contains method "Y foo()"
  ```

  For phantom types R and X, if some other constraint (e.g., of the kind induced in the case of multiple inheritance in Section 4.3) can be satisfied by making X a sybtype of S, then we get the additional constraint: "if X becomes a subtype of S then Y must be a subtype of R". This is again a very expressive constraint kind and, consequently, hard to reason about. For instance, the above constraint allows us to determine that two phantom types cannot be subtype-related. If two types declare methods with identical argument types but guaranteed-incompatible return types (e.g., `void` and `Object`), then the types are guaranteed to not be ordered by the subtyping relation, in either direction.

At the bytecode level, subtyping together with signature conformance does not imply other subtyping relationships, in the above manner. By merely having a method with the same argument types, we are not guaranteed that it overrides the respective superclass method. Instead, overloading is perfectly legal among methods that differ only in their return types. The bytecode method call resolution procedure does not rely on name/type lookup but on direct identifiers of methods.

- Casts yield no constraints at the bytecode level although they do at the source level. The reason is that the bytecode elides all unnecessary casts (i.e., upcasts). For instance, at the source level, upon seeing in code that passes the type checker a statement of the form "`(X) new C()`" we can be certain that (assuming `X` and `C` are both classes) the classes `X` and `C` are subtype-related: the cast can be either an upcast or a downcast, otherwise it would fail statically. At the bytecode level, however, a corresponding "`checkcast X`" instruction, when the object at the top of the argument stack is of static type `C`, allows no inference. The corresponding source code could well have been "`(X)((Object) c)`", with the intervening upcast elided during compilation to bytecode.

- Constraints can be induced not just by varying the requirements for the output but also by varying the assumptions for the input. In our setting, we only assumed that the input is legal Java bytecode when complemented with some extra definitions. This is distinctly different from assuming that the input has been produced by the translation of Java source code. (Bytecode could well have been produced via compilers for other high-level languages or via bytecode generators.) For instance, the Java language maintains types for all local variables. At the source level, if we call methods on the outcome of a conditional expression, we are guaranteed to be able to assign a type to it. Consider:

```
A a;
B b;
x = (foo()? a : b);
x.meth();  // I::meth()
x.meth2(); // J::meth2()
```

  In Java source, the above code means that there exists some type `X` (the type given to variable `x`) such that `X` is a subtype of `I` and `X` is a subtype of `J`, while also `A` and `B` are subtypes of `X`. An equivalent conditional in bytecode form does not need to assign a type to `X`. The constraints will be merely: `A` and `B` are subtypes of both `I` and `J`, without allowing us to infer the existence of such an unknown type `X`. Our constraint solving process is significantly simplified by the fact that we never need to infer the existence of more types.

The above is just a sampling of complications that arise if the hierarchy complementation problem is transposed to other domains, requiring the satisfaction of different static requirements. The effectiveness and efficiency of our approach is largely due to the simplicity of the Java bytecode verification requirements. However, other domains give rise to challenging problems, with a wealth of different constraints, possibly appropriate for future work.

4.8 Summary

In this chapter, we introduced the class hierarchy complementation problem. The problem consists of finding definitions to complement an existing partial class hierarchy together with extra subtyping constraints, so that the resulting hierarchy satisfies all constraints. In the context of Java bytecode and the constraints of the bytecode verifier, our problem is the core challenge of complementing partial programs soundly, i.e., so that they pass the checks of the verifier when loaded together with the generated complement.

We discuss the need for analyzing partial programs and introduce hierarchy complementation in Section 4.1, and show how the problem arises in practice in Section 4.2. We offer algorithms for the hierarchy complementation problem in both the multiple and the single inheritance setting (in Sections 4.3 and 4.4, respectively). In Section 4.5, we show how to decompose the problem in the mixed single-inheritance multiple-subtyping setting of Java, into separate single- and multiple-subtyping instances, and link it to practice with our JPhantom bytecode complementation tool (in Section 4.6). Finally, we discuss how the problem of hierarchy complementation changes when we transpose it to the domain of Java source code (instead of bytecode), in Section 4.7.

5. RELATED WORK

Obviously, you're not a golfer.

The Dude

This chapter includes related work of previous chapters (in Sections 5.1 – 5.3) and then considers more generic directions in static analysis literature (in Section 5.4). Specifically, (a) Section 5.1 contains related work for Chapter 2; (b) Section 5.2 contains related work for Chapter 3; and (c) Section 5.3 for Chapter 4.

5.1 Field-Sensitive C/C++ Pointer Analysis

Regarding our structure-sensitive pointer analysis, we discussed some closely related work throughout Chapter 2 (most importantly [104, 105]). More generally, most C and C++ analyses in the past have focused on scalability, at the expense of precision. Several (e.g., [52, 75, 138]) do not model more than a small fraction of the functionality of modern intermediate languages.

One important addition is the DSA work of Lattner et al. [73], which was the original points-to analysis in LLVM. The analysis is no longer maintained, so comparing experimentally is not possible. In terms of a qualitative comparison, the DSA analysis is a sophisticated but ad hoc mix of techniques, some of which add precision, while others sacrifice it for scalability. For instance, the analysis is field-sensitive using byte offsets, at both the source and the target of points-to edges. However, when a single abstract object is found to be used with two different types, the analysis reverts to collapsing all its fields. (Our analysis would instead create two abstract objects for the two different types.) Furthermore, the DSA analysis is unification-based (a Steensgaard analysis), keeping coarser abstract object sets and points-to sets than our inclusion-based analysis. Finally, the DSA analysis uses deep context-sensitivity, yet discards it inside a strongly connected component of methods.

The field-sensitive inclusion-based analysis of Avots et al. [10] uses type information to improve its precision. As in this work, they explicitly track the types of objects and their fields, and filter out field accesses whose base object has an incompatible type (which may arise due to analysis imprecision). However, their approach is array-insensitive and does not employ any kind of type back-propagation to create more (fine-grained) abstract objects for polymorphic allocation sites. Instead, they consider objects used with multiple types as possible type violations. Finally, they extend type compatibility with a form of structural equivalence to mark types with identical physical layouts as compatible. The implementation of `cclyzer` applies a more general form of type compatibility, presented in Section 2.5.

Miné [94] presents a highly precise analysis, expressed in the abstract interpretation framework, that translates any field and array accesses to pointer arithmetic. By relying on an external numerical interval analysis, this technique is able to handle arbitrary integer computations, and, thus, any kind of pointer arithmetic. However, the precision comes with scalability and applicability limitations: the technique can only analyze programs without dynamic memory allocation or recursion.

There are similarly other C/C++-based analyses that claim field sensitivity [50, 51], but it is unclear at what granularity this is implemented. Existing descriptions in the literature do not match the precision of our structure-sensitive approach, which maintains maximal structure information (with typed abstract objects and full distinction of subobjects), at both sources and targets of points-to relationships. Nystrom et al. [99] have a fine-grained heap abstraction that corresponds to standard use of "heap cloning" (a.k.a. "context-sensitive heap").

5.2 Static Analysis and Reflection

The traditional handling of reflection in static analysis has been through integration of user input or dynamic information. The Tamiflex tool [18] exemplifies the state of the art. The tool observes the reflective calls in an actual execution of the program and rewrites the original code to produce a version without reflection calls. Instead, all original reflection calls become calls that perform identically to the observed execution. This is a practical approach, but results in a blend of dynamic and static analysis. It is unrealistic to expect that uses of reflection will always yield the same results in different dynamic executions—or there would be little reason to have the reflection (as opposed to static code) in the first place. Our approach attempts to restore the benefits of static analysis, with reasonable empirical soundness.

An alternative approach is that of Hirzel et al. [57, 58], where an online pointer analysis is used to deal with reflection and dynamic loading by monitoring their run-time occurrence, recording their results, and running the analysis again, incrementally. This approach is quite interesting when applicable. However, it is not applicable in many static analysis settings. Maintaining and running a precise static analysis during program run time is often not realistic (e.g., for expensive context-sensitive analyses). Furthermore, the approach does not offer the off-line soundness guarantees one may expect from static analysis: it is not possible to ask questions regarding all methods that may ever be called via reflection, only the ones that have been called so far.

Interesting work on static treatments of reflection is often in the context of dynamic languages, where resolving reflective invocations is a necessity. Furr et al. [42] offer an analysis of how dynamic features are used in the Ruby language. Their observations are similar to ours: dynamic features (reflection in our case) are often used either with sets of constant arguments (in order to avoid writing verbose, formulaic code), or with known prefixes/suffixes (e.g., to re-locate within the file system).

Madsen et al. [88] employ a use-based analysis technique in the context of Javascript.

When objects are retrieved from unknown code (typically libraries) the analysis infers the object's properties from the way it is used in the client. In principle, this is a similar approach to our use-based techniques of Section 3.4.2 (both object invention and back-propagation) although the technical specifics differ. The conceptual precursor to both approaches is the work on reflection by Livshits et al. [84, 85], which has been extensively discussed and contrasted throughout Chapter 3 (see Sections 3.3, and 3.4.2).

Advanced techniques for string analysis have been presented by Christensen et al. [24]. They analyze complex string expressions and abstract them via a context-free grammar that is then widened to a regular language. The regular approximations produced by this approach are richer than the prefix and suffix matching of our substring analysis, and can thus better approximate the possible values of arbitrary string expressions. Reflection is one of their examples but they only apply it to small benchmarks.

Ali and Lhoták [2, 3] offer comparisons of dynamic and static call-graph edge metrics. They discover hundreds of missing edges in several of the DaCapo 2006-10-MR2 benchmarks. However, their experiments do not integrate the vastly improved support for reflection (e.g., modeling of `Object.getClass`) offered by Elf or our current work.

Stancu et al. [126] empirically compare profiling data with a points-to static analysis. However, they target only the most reflection-light benchmarks of the DaCapo 9.12-Bach suite (*avrora, luindex, and lusearch*), and patch the code to avoid reflection entirely.

5.3 Class Hierarchy Complementation and Static Analysis on Partial Programs

The hierarchy complementation problem, which we presented in Chapter 4, is in principle new, although indirectly related to various other pieces of work in the literature.

From a theory standpoint, our problem is an attempt to more fully determine the structure of a partially ordered set. There is no exact counterpart of our algorithms in the literature. However, there has been work on sorting a poset, i.e., completely determining the partial order [33]. The challenge in such algorithmic research, however, is to perform the sorting with a minimal number of queries. None of the interesting devices of our algorithms are present. Specifically, the device of the single inheritance case (if a node can reach two others, they have to be ordered relative to each other) does not apply, and neither does the interesting constraint of the multiple inheritance case (we cannot add direct supertypes to a known node).

Complementing a program so that the result respects static properties is analogous to analyzing only parts of a program but giving guarantees on the result. There are few examples of program analyses of this nature. Notably, Ali and Lhoták introduce a technique [2] for analyzing an application separately from a library, while keeping enough information (from the library analysis) to guarantee that the application-level call-graph is correct. Furthermore, the Averroes system [3] uses the assumption that the missing code is independently developed in order to produce a worst-case skeletal library. That is, Averroes takes an existing library and strips away the implementation, keeping only the interface between the library and the application. The implementation is replaced with code (at the bytecode level) that performs worst-case actions on the arguments passed into the library, for the purposes of call-graph construction (i.e., the generated code calls all methods the eliminated original code could ever call). Averroes is related to JPhantom but at rather opposite ends of the spectrum: Averroes produces worst-case skeletal implementations, while JPhantom produces minimal, best-case implementations that still respect well-formedness at the type level. At the same time, Averroes assumes the library interface is available and just tries to avoid analyzing library implementations, while JPhantom applies precisely when the library is completely missing. Thus, JPhantom truly applies to the case of partial programs, whereas Averroes analyzes a partial program but under the assumption that the whole program was available to begin with. It would be interesting to treat a partial program first with JPhantom and then apply Averroes to the JPhantom-produced program complement to obtain the worst-case behavior of a plausible interface for the missing code.

Analyzing partial programs at the level of source code has been studied by Dagenais and Hendren [31]. Apart from the elaborate type constraints that may arise from missing source code, as discussed in Section 4.7, another important challenge is syntactic ambiguities (e.g., *is it a package or a class name?*). The goal of Dagenais and Hendren is to be able to deal with both typing problems and syntactic ambiguities in any given partial Java program and produce a typed IR (although it may contain placeholders representing unknown types and packages). Their approach is primarily based on heuristics that make arbitrary choices on various occasions (with no form of backtracking) that may eventually lead to wrong results. Even though they, too, identify similar subtype constraints and record missing class members to be added in the typed IR, they do not define any similar type hierarchy complementation problem or come up with algorithms to systematically construct a valid solution in such a setting.

Our hierarchy complementation problem bears a superficial resemblance to the *principal typings* problem [4–6]. The principal typings problem consists of computing types for a Java module in complete isolation from every other module it references. I.e., principal typings aim to achieve a more aggressive form of separate compilation, by computing the minimal type information on other classes that a class needs in order to typecheck and compile. Thus, the motivation is fairly similar to ours, but the technical problem is quite different. First, in our setting we already have the result of compilation in the form of bytecode, and bytecode only. Second, our emphasis is on satisfying constraints instead of capturing them as a rich type. Finally, our constraints are of a very different nature from any in the principal typings literature. As discussed in Section 4.7, the input and output language assumptions crucially determine the essence of an incomplete program analysis problem.

5.4 Other Directions in Static Analysis

In this section, we will extend our focus and examine more generic directions and different methodologies in static analysis literature.

CFL reachability formulation. In Chapters 2 and 3, we formulated pointer analysis algorithms in inference rules that can be straightforwardly expressed in Datalog. Employing a restricted language not only offers guarantees of termination and complexity bounds, but also permits more aggressive optimization of the language features.

Along these lines, pointer analysis and other related analyses have been formulated as a *context-free language (CFL) reachability* problem. The idea is that we may encode an input program as a labeled graph, and a specific analysis corresponds to the definition of a context-free grammar, G. The relation being computed holds for two nodes of the graph if and only if there exists a path from one node to the other, such that the concatenation of the labels of edges along the path belongs in the language $L(G)$ defined by G.

Specifically, the input graph normally consists of nodes representing program elements such as variables, types, methods, statements, and so on. Edges represent relations between those program elements. For instance, an edge $(s,\, t)$ may represent that there exists an assignment from variable s to variable t. Moreover, edges may encode field accesses (load/store), method invocations, pointer dereferences, etc, and hence may even connect different *kinds* of program elements. The exact choice of domains depends on the specific analysis being run and the problem it addresses. Since we want to express many input relations, we need many types of edges, represented as labels (e.g., we can label a field access edge by some symbol denoting field access plus the name of the field). For a given analysis, a context-free grammar G encodes the desired computed relations (e.g., which pointers are memory aliases) as non-terminal symbols, and supplies production rules that express how they relate to the simpler relations represented by graph edges (terminals). The CFL reachability answer is then commonly computed by employing a dynamic programming algorithm.

The first application of such a framework in program analysis was designed to solve various interprocedural dataflow-analysis problems [108], but CFL reachability has since been used in a wide range of problems, such as: (i) the computation of points-to relations [107], (ii) the (demand-driven) computation of may-alias pairs for a C-like language [138], (iii) Andersen-style pointer analysis for Java [125].

Any CFL reachability problem can be converted to a Datalog program [107], but the inverse is not true (i.e., CFL reachability corresponds to a restricted class of Datalog programs, the so-called “chain programs”). Thus, the primary advantage of CFL reachability is that it permits more efficient implementations.

A *chain program* consists of rules of the form:

$$p(X,Y) \leftarrow q_0(X,Z_1),\, q_1(Z_1,Z_2),\, \ldots,\, q_k(Z_k,Y).$$

We can express a CFL reachability problem in Datalog by using such a chain rule to represent the following production of grammar G:

$$p \rightarrow q_0\, q_1\, \cdots\, q_k$$

where a Datalog fact $e(m, n)$ represents an edge (m, n) labeled e in the graph.

An even more restrictive variant, *Dyck-CFL reachability*, can be obtained by restricting the underlying CFL to a Dyck language, i.e., one that generates balanced-parentheses expressions. Although restrictive, this approach still suffices for some simple pointer analysis algorithms, while allowing very aggressive optimization, often with impressive performance gains [135].

Constraint graph approaches and optimizations. Several optimization techniques have appeared in the pointer analysis literature, based on the concept of the *constraint graph*: a graph whose nodes denote pointer expressions and whose edges denote flow between these pointer expressions.

Such an edge may arise either: (i) *before* points-to computation—due to an explicit assignment instruction "$q = p$", for example, that the input program contains—or (ii) *during* points-to computation, by also taking field accesses (i.e., load/store instructions) into account in order to infer additional flow. These techniques have typically targeted the C language.

A variety of constraint graph optimization techniques have been presented that can be applied either *offline* (i.e., before points-to computation) [49, 112], or *online* (i.e., during points-to computation) [38]. Hybrid approaches also exist [52]. The essence of most of these techniques lies in identifying nodes with guaranteed-equivalent points-to sets and collapsing them into a single representative node. Such equivalence classes may arise, for instance, when nodes participate in a cycle of the constraint graph [38, 53], or even if they share a common dominator [98]. The *set-based pre-processing* technique [122] generalizes such approaches by also allowing the removal of *edges* (and not just merging of nodes) from the constraint graph. It also restricts the application of optimizations to an intra-procedural setting, so that they can be applied in conjunction with nearly any pointer analysis algorithm, together with on-the-fly call-graph construction.

Shape Analysis. So far, allocation sites is the primary source that drives abstract object creation, in the techniques we have presented for pointer analysis. Despite our deviation from the standard allocation-site abstraction, where a single abstract object will be created per allocation site, even our own techniques, described in Chapter 2, will use the allocation site as the basis of abstraction (but may create multiple objects per site, nonetheless). A different approach altogether is that of the techniques termed as *shape analysis* [39, 74, 89, 114–116]. The primary goal of standard shape analysis is to be able to infer the *shapes* of objects in memory. E.g., to be able to detect if some objects form a list or a tree, if some list may contain cycles, if a subtree or a portion of a list may be shared (i.e., be reachable from multiple objects), and so on.[1]

To achieve such a feat, shape analysis associates a list of properties to each abstract memory object (e.g., *pointed by variable v*, *transitively reachable by variable r*, and so on) and uses Kleene's three-valued logic to differentiate between must and may information. For instance, if the "*pointed by variable* v" property of an abstract object $\widehat{obj}$ has the value 1 at some memory state, it means that variable v *must* point to this object (at the given state), whereas the value $\frac{1}{2}$ would represent our familiar *may* notion of points-to. Hence, shape analysis performs an amalgam of must and may analysis simultaneously.

At each memory state the analysis has computed, it tries to collapse $\frac{1}{2}$ values of properties to either 0 or 1, via the so called *focus operation*. Inconsistent states are then discarded at the *coerce operation*. Thus, the analysis dynamically tries to eliminate uncertainty by *focusing* on the values of some core predicates (and statement-specific formulas), at the expense of possible memory state explosion—the abstract interpretation of each program statement tends to create multiple output states for each one of its input states. As for abstract objects, they are defined only by the values they have for some basic properties (called *abstraction properties*) of the particular analysis. Therefore, the upper bound for the number of abstract objects in memory is exponential to the number of abstraction predicates defined. (The latter will almost certainly include a predicate per each program variable.)

By applying these techniques, and choosing the right abstraction predicates, the analysis will be able to carve out the *shapes* of objects in memory. For instance, given an input memory state where variable v points to the head of some list l with at least two elements, the abstract interpretation of instruction "`v = v->next`" will create multiple output memory states where either: (i) v points to a new head of a list with a non-empty tail (corresponding to the case where l contained at least three elements), or (ii) v points to the single element of a list with no tail at all (corresponding to the case where l contained exactly two elements). In both output memory states, the analysis will compute values of 1 for the predicate "*pointed by variable v*"; hence, it will know exactly where v points to and maintain precision, at the expense of increasing the possible memory states by 1.

Sagiv, Reps, and Wilhelm initiated the field of parametric shape analysis via 3-valued logic [114–116], and Lev-Ami and Sagiv presented the TVLA framework for shape analysis [74]. Since then, there has been work on various extensions such as a more economic heap abstraction [89], better support for recursive programs [111] and programs with highly nested data structures [15], and the incorporation of a value analysis into the shape analysis algorithm [39], among others.

[1] The term shape analysis is quite generic (i.e., any analysis designed to infer the shapes of objects) and has been examined in many different contexts [43]. Here, we focus on *parametric shape analysis via 3-valued logic*, as one of the most notable methodologies in that area.

Separation Logic. Points-to analysis provides a model of the heap (or memory, in general, for a language such as C). Other approaches for heap analysis that can be used to prove pointer safety are based on the field of *separation logic*. Separation logic, in turn, can be viewed as an extension of Hoare logic [41, 59, 102, 110]. Hoare logic is a formal system for reasoning about the correctness of programs, by encoding the programming language's semantics in *Hoare tiples*. A Hoare triple has the form $\{P\}\,C\,\{Q\}$ and describes how the execution of a command changes the state of the computation. Specifically, it states that whenever the assertion P holds, before executing command C, then assertion Q will hold afterwards (if C terminates). The $P,\ Q$ assertions can express conditions on program variables, written by using standard mathematical notations together with logical operators (or, in general, some form of calculus like *first-order logic*).

Hoare logic provides two ways to generate verification conditions: (i) either forwards, by starting from a precondition and generating formulas to prove a postcondition (ii) or backwards, by starting from a postcondition and trying to prove a precondition. Either way, in the general case, it cannot provide fully automated reasoning; building a proof may require human guidance.

Separation logic [61, 102, 109, 110] extends Hoare logic by introducing additional operators in the syntax of assertions, that facilitate local reasoning. Namely, the *separating conjunction* $P * Q$ asserts that P and Q hold for separate portions of memory, and thus can be used on program-proof rules to provide modular reasoning about programs. Additional operators include the separating implication, $P \mathrel{-\!\!*} Q$, which asserts that if the current heap is extended with a disjoint part in which P holds, then Q will hold in the extended heap, and more. Note that, these operators do not increase the "completeness" of Hoare logic—what can be proven in separation logic can also be proven in Hoare logic. Rather, they merely simplify the specifications and proofs.

Calcagno et al. present a compositional shape analysis [22] to be used in (lightweight) program verification that builds on these concepts of separation logic (instead of the TVLA approach of Sagiv, Reps, and Wilhelm). In classical logic, *abduction* stands for the inference of "missing" assumptions M, such that, given another assumption A and a goal G, one can prove G by synthesizing A and M:

$$A \wedge M \;\vdash\; G$$

A similar problem can be phrased by using separating conjunction instead of classical conjunction, which also partitions the premises:

$$A \,*\, M \;\vdash\; G$$

This finally leads to the more general problem of *bi-abduction*, if we allow for leftover portions of state (the frame):

$$A \,*\, \textit{?anti-frame} \;\vdash\; G \,*\, \textit{?frame}$$

The notion of bi-abduction is used as the basis of a new compositional interprocedural shape analysis algorithm, which synthesizes pre- and post-conditions for each program function, by performing a symbolic execution that presumes existing specifications. When symbolically executing a method call with a given specification, in the form of a pre- and a post-condition (which correspond to A and G), the analysis infers a frame and an anti-frame. Such inferred frames and anti-frames will eventually lead to the computation of the final pre- and post-conditions for the enclosing function, when all of its instructions are processed.

Abstraction Strategies. In Chapter 2, we used an abstraction policy for memory objects that may produce more than one abstract objects per allocation site. The exact strategy by which memory objects are represented by a static analysis is a crucial design choice.

Using the allocation site as a means of abstraction dates at least back to the work of Jones and Muchnick [63], who use graphs to summarize heap data structures. An allocation site (i.e., cons instruction) is handled by creating a new graph node that may represent multiple concrete allocations for a given execution. The same principle for abstracting cons cells is used later in the work of Chase et al. [23].

Previously, we also talked about shape analysis [114–116] as another means of abstracting memory that goes far beyond allocation sites, by introducing the concept of arbitrary abstraction predicates. The recency-abstraction approach of Balakrishnan and Reps [11] lies somewhere in between. For each allocation site, their analysis creates two abstract objects: (i) the first one represents the *most-recently-allocated* object (for the given site), while (ii) the second one summarizes all the other (least recent) objects for the same site. Note that the first object represents just a *single* concrete object and, hence, can be used to perform strong updates, which is an essential technique to improve the precision and scalability of a flow-sensitive analysis. In contrast, shape analysis would instead record a summary abstraction predicate for each abstract object that captures whether it represents more than one concrete objects. This would naturally lead to the specialization of non-summary nodes that exhibit interesting shape properties (such as the head of some list structure) according to the rest of the abstraction criteria. Kanvar and Khedker present a thorough taxonomy of various other strategies in heap abstractions [65].

Context Sensitivity. A pointer analysis attempts to compute an abstract yet reasonably precise model of the program's memory for all possible executions. Recall the basic query that such a model needs to answer, from Chapter 2:

> What can a pointer variable point to, i.e., what can its value be when dereferenced during program execution?

Of course, if the same question is phrased in the Java setting—in Java, variable references point to heap allocations—instead of C/C++, it becomes:

> What heap objects can each reference variable point to, during program execution?

As discussed in the previous paragraph, there are many different approaches on how to abstract memory objects. Simpler approaches, like the allocation-site abstraction, lead to more efficiently computable memory models, whereas on the opposite end we can have models with complex abstractions, such as shape analysis, that do not scale well with program size.

A generic approach that increases the granularity of the analysis abstractions and yields greater precision is that of context sensitivity. The idea is to qualify variables and abstract objects with context information. Thus, a points-to edge $(v, c_1, \hat{o}, c_2)$ now captures that variable v, under context c_1, points to abstract object $\hat{o}$, allocated under context c_2. Essentially, this means that we have refined our abstractions: the same variable v is represented by multiple variable-context tuples, (v, c_1), each one corresponding to some different execution path, and having an independent points-to set. Similarly, the same allocation site (given that we have used the allocation-site abstraction as a basis for our object abstraction) of $\hat{o}$ is now represented by multiple object-context tuples, $(\hat{o}, c_2)$, each one corresponding to a different execution context under which the allocation was made.

There are many flavors of context sensitivity in existing literature, which use different strategies for creating contexts. To name a few:

- *Call-site sensitivity* (or *k*-CFA in the context of functional languages, even though there are important differences in applying it to functional versus object-oriented settings [90]) that uses method call sites as context elements [118, 119]
- *Object sensitivity*, where object allocation sites are used as context elements [91, 92]
- *Type sensitivity*, which is similar to object sensitivity but instead collapses the objects to their types to achieve better scalability [123]
- *Hybrid context-sensitivity* that selectively combines call-site and object sensitivity [66].

Each approach has its relative merits and unique performance characteristics; none is clearly superior to all others. The exact choice of context has to consider the specific characteristics of the program being analyzed. Generally, it is a matter of experimentation but automated machine-learning techniques can also be used to adapt the degree of context sensitivity [46, 81, 82, 101]. Other approaches such as counter-example-guided abstraction refinements, use the feedback from failed analysis runs to pick a new abstraction which is less likely to fail [48, 136, 137].

6. CONCLUSIONS AND FUTURE WORK

That rug really tied the room together.

The Dude

In the final chapter, we shall assess our initial thesis and conclude, while also considering interesting directions for future work.

Our thesis states that there exists *implicit structural information* in the program, about the memory it will allocate, which we can recover via inference and use it to improve the quality of a static analysis (by coming up with a better abstract memory model). In Chapter 2, we examined the problem of pointer analysis in C/C++, as a typical example of a low-level language with direct memory access. We identified particular causes of analysis imprecision, in untyped allocations (via generic `malloc`-like allocation routines) and in the language's capability of allowing pointers to point *inside* an object (i.e., to point to one of its fields or array elements), which further complicates the modeling of field sensitivity.

We presented a structure-sensitive pointer analysis that may achieve better precision by changing its abstraction strategy: when the same allocation site is used to create objects of many different types, the analysis comes up with many different strongly-typed abstract objects to represent the same instruction. Also, each different field and array element of each such strongly-typed object (of a complex type) will be represented by a unique abstract subobject itself, having its own separate points-to set, while also maintaining strong type information. The final outcome is a much finer-grained allocation abstraction (than that of the typical C/C++ points-to analysis) that is guided primarily by the flow of types. The key to this technique is that tracking the flow of types that will determine what objects will be created is not determined before the analysis has run, but simultaneously ("on-the-fly"), thus yielding better results due to the recursive nature of the computation. We describe the analysis in precise terms and show that its approach succeeds in maintaining precision when analyzing realistic programs.

This is a prime example of how complex inference (in this case, the relations computed by the pointer analysis itself) that tracks the use of types in the program can be used to recover structural information, namely, the various structures that may be used to access or modify any untyped heap allocation in memory. The recovered structural information, in turn, improves the quality of the analysis, in terms of precision, as shown in the evaluation section of Chapter 2. In our experience, the techniques we described are essential for analyzing C/C++ programs at the same level of precision as programs in higher-level languages.

In Chapter 3, we shift our focus to higher-level languages like Java, with no pointer arithmetic or direct memory access capabilities. However, even in such a setting, our thesis applies: we can improve the quality of a pointer analysis (now, in terms of empirical soundness) by recovering structural information for objects involved in reflective operations. By using reflection, Java programs can encompass dynamic behavior, and as such, are difficult to statically analyze.

The challenges with reflection idioms are quite similar to those of analyzing C/C++ that we have already encountered:

- the same (reflective) allocation site can be used to allocate objects of many different types
- no local type information exists for reflective objects; instead, the types are determined by run-time strings that are passed on as arguments to the reflective operations involved
- apart from normal object allocations, there are also reflective calls that return special objects representing program classes, fields, and methods; a reflective call on a method object is much like a C/C++ call via a function pointer.

To tackle the limitations of existing pointer analyses, induced by reflection, we presented a static reflection analysis (which also runs simultaneously with the core points-to analysis) that employs similar techniques to those of Chapter 2: by inspecting the use of reflective objects (i.e., what types they are cast to, what fields they access, and so on), the analysis is able to recover essential parts of the objects' structure. Our techniques build on top of state-of-the-art handling of reflection in Java, by elegantly extending declarative reasoning over reflection calls and inter-procedural object flow. Our main emphasis has been in achieving higher empirical soundness, i.e., in having the static analysis truly model observed dynamic behaviors.

By comparing the statically computed call-graphs against dynamically computed ones from actual executions, we find that these techniques, indeed, improve empirical soundness (i.e., the static analysis is now able to cover more of the actual dynamic behavior of the program than before). Although full soundness is infeasible for a realistic analysis, it is possible to produce general techniques that enhance the ability to analyze reflection calls.

As already noted, reflection is but one of the possible causes of lost memory structure, when statically analyzing Java programs. In Chapter 4, we examine how to recover lost structural information for partial Java programs.

Regarding such need, dynamic class loading allows programs to depend on an abundance of external libraries, without actually requiring all of them to execute. Transitive dependencies (i.e., the dependencies of the libraries directly used by the program) make things even worse, from the perspective of a static analyzer, since such dependencies are much more numerous and less likely to be actually used. Hence, whole programs might be both prohibitively difficult to obtain and unnecessary at the same time; if some parts of the program were missing, they would make no difference in any actual execution, had these parts been truly redundant. In such cases, missing these parts should also make no difference in analyzing the program, which explains why there is a strong incentive for static analysis tools to be able to analyze partial programs. Despite such incentive, static analysis tools are rarely well-equipped or error-tolerant enough to be able to cope with partial programs. When they do so, they risk a great deterioration of their results.

To this end, we have presented a generic complementation approach, in Chapter 4, that transforms a partial program to a whole one, while also seeking to provide definitions for its missing parts so that the "complement" satisfies all static and dynamic typing requirements induced by the code under analysis. This requires satisfying constraints relative to methods and fields of the missing classes, as well as subtyping constraints and constraints on whether a missing type has to be a class or interface. To identify such constraints, we analyze the program and track uses of any phantom types therein (i.e., types being used but not defined in the code that is available), so that we can recover their lost structure. Inferring missing members of phantom types is straightforward. The primary challenge, however, lies in computing the missing parts of the type hierarchy, in a way that satisfies all the implied subtyping constraints.

We have defined the type hierarchy complementation problem as follows: given a partially known hierarchy of classes together with subtyping constraints ("A has to be a transitive subtype of B") complete the hierarchy so that it satisfies all constraints. We formulate the problem systematically and offer algorithms to solve it in various inheritance settings. The result is the articulation of a novel typing problem in the OO context.

The entire complementation approach, including the algorithms to solve the type hierarchy complementation problem are implemented in JPhantom, a program complementation tool of practical interest for Java bytecode. We evaluated JPhantom on both synthetic and real-world benchmarks to show that we produce practical complements of significant size in a few seconds and, in this way, allow the analysis of previously un-analyzable partial programs.

Lastly, we used the synthetic benchmark (based on the *antlr* parser generator) to evaluate the difference between analyzing a partial program as is, and analyzing it after being complemented by JPhantom. The comparison demonstrated that the latter is much closer to analyzing the original whole program: analyzing the partial program without complementation fails to find a great number of reachable methods (which will not be analyzed at all). Therefore, recovering the structural information of phantom types, via our program complementation technique, has improved the static analysis on the partial program, thus reinforcing our thesis.

We believe that the hierarchy complementation problem is fundamental and is likely to arise in different settings in the future, hopefully aided by our modeling of the problem and some of its solution avenues.

To summarize, we advocate that there are many opportunities of recovering implicit structural information about memory that can improve the static analysis of programs, but require complex inference that takes advantage of indirect uses of types. We have examined three different scenarios to test and evaluate our thesis, regarding generic C/C++ programs, and Java programs that either use reflection or are missing parts of their code. In all cases, we where able to improve static analysis, by recovering memory structure that was not previously evident.

6.1 Future Work

Finally, we will discuss some interesting future directions to tackle existing limitations of our approaches.

6.1.1 Flow-Sensitivity and Strong Updates

To begin with, our structure-sensitive points-to analysis for C/C++ lacks flow sensitivity, which is a useful feature for C-like languages. Flow sensitivity takes the order of program instructions into account, to increase analysis precision. The results it computes are now specialized to a specific program point, at which they hold. Thus, different program points will be associated with different points-to results, by the end of an analysis run.

We have already discussed, in Chapter 2, the SSA transformation built in the LLVM bitcode format. SSA can be viewed as a limited form of flow sensitivity, since it ensures that a variable is only assigned once; a variable with multiple assignments will be split into multiple different variables, one per assignment. However, SSA is far from adequate, especially in LLVM bitcode, where address-taken variables bypass SSA, by being transformed to pointers whose contents can be changed by multiple store instructions. This is not specific to analysis of C/C++ or LLVM bitcode. In a Java pointer analysis, like the one we have briefly presented in Chapter 3, despite SSA, fields of objects can be, again, updated multiple times.

To take advantage of flow sensitivity and be able to substantially affect precision, an analysis has to perform *strong updates*. A store instruction strongly updates the results of an over-approximate pointer analysis, when it overwrites the previous points-to contents for the given address. A weak update only *extends* points-to contents without removing any prior results. However, it is not generally sound for a flow-sensitive pointer analysis to perform a strong update without additional reasoning. The reason is that an abstract object summarizes multiple concrete objects; to strongly update it, we have to ensure that the update applies to all concrete objects that it represents.

We previously discussed how the recency-abstraction of Balakrishnan and Reps [11] allows us to perform strong updates. Another approach is that of Lhoták and Chung, who perform strong updates only on singleton points-to sets [75]. However, even with strong updates, the cost of flow-sensitivity could be prohibitive. Improving the efficiency of flow-sensitive analyses has been studied extensively in the literature [34, 77, 79, 128]. Khedker et al. perform a context-sensitive and flow-sensitive analysis but limit the points-to information they compute based on a liveness analysis about pointer ranges [67]. Ye et al. limit flow-sensitivity by partitioning the program into regions and maintaining flow-sensitivity only between the regions but not inside [134]. Incorporating such approaches and possibly devising other methods that enable strong updates in limited cases would be a key step in further improving the precision and overall value of our structure-sensitive analysis.

6.1.2 Integer and String Value Analysis

Our structure sensitive analysis for C/C++ takes array indices into account, to introduce a form of array-sensitivity by which it can statically differentiate distinct array elements. Array indices are not the only case where our analysis could benefit from integer values. The number of bytes specified in an allocation instruction (e.g., via the single parameter of a `malloc()` call) could help us filter some invalid object-type associations.

Such cases, however, are only the tip of the iceberg in how a static analysis can find use in reasoning about the domains of integer values. Our analysis could incorporate more sophisticated approaches in tracking integer-valued quantities and numerical properties of program variables. For a language like C, which allows pointers to be treated as integers, even aliasing information could be tied to the computation of integer value domains. The standard use case of numeric domains, however, is for answering queries about integer overflows and array bounds checking. One of the most precise approaches in the computation of numeric abstract domains is analysis based on the polyhedral domain [30]. Cheaper variants include the Octagons abstract domain [95], the Pentagons domain [87], and difference-bound matrices (or Zones) [93].

For similar reasons, we can employ more sophisticated string analyses to upgrade our reflection handling of Chapter 3, such as that of Christensen et al. [24], which we discussed in the previous chapter.

6.1.3 Context Sensitivity

We have already discussed context sensitivity in the previous section, and the various strategies in specifying what program elements will constitute the context.

For the analysis of LLVM bitcode—or any other format that has been lowered to a C-like language for that purpose, where the OO features have been translated away to more basic constructs—it is not evident when and how to use object sensitivity in a generic and parameterized way. An obvious generalization would be to leverage the values of every parameter (and not just the receiver object) for the purposes of building context at method calls, as in the cartesian product algorithm [1]. Such an approach, however, would certainly impede the scalability of the analysis. Generalizing past approaches on context-sensitivity to work well both in the presence and absence of object-oriented features, e.g., by identifying what parameters could be used as objects, per method call, is a direction worth investigating.

ABBREVIATIONS - ACRONYMS

ABI	Application Binary Interface
API	Application Programming Interface
CFL	Context-Free Language
DAG	Directed Acyclic Graph
DLL	Dynamic-link Library
GEP	LLVM Bitcode's `getelementptr` instruction
GUI	Graphical User Interface
DSA	Data Structure Analysis
IDE	Integrated Development Environment
IR	Intermediate Representation
JAR	Java Archive
JDK	Java Development Kit
JIT	Just-in-time
JVM	Java Virtual Machine
JVMTI	JVM Tool Interface
OO	Object-oriented
RISC	Reduced Instruction Set Computing
SSA	Static Single Assignment
SQL	Structured Query Language
TVLA	Three-Valued Logic Analysis Engine
WYSINWYX	What You See Is Not What You Execute
XSLT	Extensible Stylesheet Language Transformations

APPENDIX A. APPENDIX TO CHAPTER 4

A.1 Multiple Inheritance Correctness Proof

We shall call the path-edges originating from known-nodes kp-*edges*. We will also use the symbols $S_0, S_1, \ldots, S_\infty$ to denote the various stratifications computed at each step of Algorithm 4.1. Note that our algorithm will actually produce a finite number of stratifications (at most $|V|$) but we can disregard both the upper limit of the main loop and the early-failure condition (line 23) for proving correctness. Instead we focus on the main computation (line 18) and the infinite sequence of stratifications that would be produced if it was allowed to run forever (even after reaching a fixpoint).

Lemma 1. *For all $v \in V$, the sequence $\{S_0[v], S_1[v], \ldots\}$ is non-decreasing.*

Proof. Direct consequence of line 18 of the algorithm. □

Lemma 2. *For all $i \in \mathbb{N}$, $0 \leq S_i[v] \leq i, \forall v \in V$.*

Proof. Induction on step i.

1. *Base*: For $i = 0$, we have that $S_i[v] = S_0[v] = 0, \forall v \in V$.
2. *Inductive Step*: Assume that $0 \leq S_n[v] \leq n, \forall v \in V$ for some value of n. We must show that $0 \leq S_{n+1}[v] \leq n + 1, \forall v \in V$. Let k be a node in V. Either $S_{n+1}[k] = S_n[k]$, and therefore $0 \leq S_{n+1}[k] \leq n$, or there will exist a node s, s.t. $S_{n+1}[k] = S_n[s] + 1$, in which case $1 \leq S_{n+1}[k] \leq n + 1$.

□

Definition A.1. A node $v \in V$ is *i-stabilized* if and only if $S_i[v] = S_{i+1}[v]$ and either $i = 0$ or $S_{i-1}[v] < S_i[v]$.

Theorem 1. *(Once a node's stratum does not change, it will not change again.) If $S_i[v] = S_{i+1}[v]$ for some node $v \in V$ and a value $i \in \mathbb{N}$, then $S_j[v] = S_i[v], \forall j \in \mathbb{N}$ such that $j \geq i$.*

Proof. Induction on step i.

1. *Base*: For $i = 0$, let v be a node in V, such that $S_0[v] = S_1[v]$. From Lemma 2, we have that $S_0[v] = S_1[v] = 0$, which can happen if and only if v has no incoming edges (otherwise an edge would cause the node to move to a higher stratum on iteration 1). It is therefore impossible for v to change in the following iterations since it has no constraining edges.

2. *Inductive Step*: Assume that the theorem holds for all $i < n$ for some value of $n \in \mathbb{N}$. Let $t \in V$ be a node, such that $S_n[t] = S_{n+1}[t]$. We will show that t's stratum will not change in the future. It suffices to prove that, for each of t's constraining edges, there will be a node s that has already been stabilized at a lower stratum than t, and can be used to satisfy the constraint at this point. Therefore, the constraint will remain satisfied in future iterations due to s, which will remain in the same stratum from now on (induction hypothesis). For ordinary *path*-edges, node s is no other than the source of the edge itself, while for *kp*-edges, it is the lower phantom projection of the edge's source at step n that we may use instead. Let us consider ordinary path-edges first, in more detail. From Lemma 2, we have that $0 \leq S_n[t] \leq n$, and thus $0 \leq S_{n+1}[t] \leq n$. Let $(s, t) \in E$ be an incoming edge of t. We have that $0 \leq S_n[s] < S_{n+1}[t] \leq n$ which entails that $0 \leq S_n[s] \leq n - 1$. Therefore, according to Lemma 2, we have that $0 \leq S_i[s] \leq n-1, \forall i \in \{0, \ldots, n\}$. By the pigeonhole principle, and due to Lemma 1, there must surely exist two consecutive values $i, i+1$, s.t. $S_i[s] = S_{i+1}[s]$ and $i < n$. From the induction hypothesis, we know that s will therefore not change and its constraint on t will be irrelevant in future iterations. We proceed similarly, for a *kp*-edge (s, t) (where we use the lowest phantom projection of s at this point, instead of s itself). Thus, t will not change in the future, since every constraint of t will remain satisfied after this iteration.

□

Corollary 1. *For all $v \in V$ and $n \in \mathbb{N}^+$, $S_{n-1}[v] \neq S_n[v] \Rightarrow S_n[v] = n$.*

Theorem 2. *The stratification sequence $S_0, S_1, \ldots$ will diverge (i.e., not reach a fixpoint) if and only if at some computation step, n, no new nodes stabilize and not all nodes have already stabilized—that is, $\exists n \in \mathbb{N}^+$, such that: for some $v \in V$, $S_{n+1}[v] \neq S_n[v]$ and for all $v \in V$, $S_{n+1}[v] = S_n[v] \Rightarrow S_n[v] = S_{n-1}[v]$.*

Proof.

1. (*If*) Let n be a computation step, such that $(\forall v \in V)\ (S_{n-1}[v] \neq S_n[v] \Rightarrow S_n[v] \neq S_{n+1}[v])$. We can disregard any node u such that $S_{n-1}[u] = S_n[u]$, and observe that for each remaining node, there must exist at least a constraining edge that cannot be satisfied with a node that has already been "stabilized". That said, due to Corollary 1, each remaining node $v \in V$, s.t. $S_{n-1}[v] \neq S_n[v]$, will be placed at a higher (by 1) stratum at this point, i.e., $S_i[v] = i, \forall i \in \{0, \ldots, n+1\}$. Since the relative positions of all the remaining nodes will be the same at step $n+1$ as they had been at step n, there is no way for a node to be stabilized at this last iteration. In other words, there is a cyclic dependency between the remaining nodes that will remain unaltered, thus eliminating the possibility of reaching a fixpoint.

2. (*Only If*) Due to Theorem 1, we know that we need at most $|V|$ computation steps, in order to reach a fixpoint, if at each computation step there exists at least a new node that gets stabilized. In other words, we need a finite number of steps to reach

a fixpoint, if each step results in some progress. Thus, failure to reach a fixpoint requires an iteration where no progress has been made, i.e., no new nodes get stabilized.

□

Therefore, the optimization in Algorithm 4.1 of detecting this exact condition (line 23) and terminating would be triggered if and only if no fixpoint would be reached whatsoever, had the algorithm continued its execution.

Soundness. We need to show that, if our algorithm computes a solution, this solution will be sound. Firstly, our algorithm maintains the invariant that $\forall (s,t) \in E$, node s will eventually—i.e., once we reach a fixpoint—be placed somewhere lower than node t (otherwise this condition would trigger yet another iteration). Therefore, our solution will contain no cycles since all of its edges will be facing *upwards*, i.e., from a lower to a higher stratum. Furthermore, it is evident that, for each *kp*-edge (s,t), there will always exist a node $p \in$ *proj*(s), such that p will be placed at a lower stratum than t in our final solution. Thus, we can add the edge (p,t) without introducing any cycles if none existed so far. This process will therefore generate a valid solution. □

Completeness. We need to show that, if a solution exists for a given constraint graph, then our algorithm will also be able to compute a solution, or equivalently (according to Theorem 2) that the stratification sequence being computed will reach a fixpoint. Consider such a (posited but unknown) solution. For such a solution we may generate a stratification (since the solution may contain no cycles), such that each of its edges is facing upwards and no empty strata exist, that is, $\forall (s,t) \in E_{sol} : S_{sol}[s] < S_{sol}[t]$, where E_{sol} are the edges that form the solution, and S_{sol} is its stratification. We first show an important lemma.

Lemma 3. *Let S_{sol} denote the stratification of a valid solution of the problem instance. We have that: $\forall i \in \mathbb{N}, \forall v \in V : S_i[v] \leq S_{sol}[v]$.*

Proof. Suppose that there is a step $k \in \mathbb{N}$, such that it contains at least one node $u \in V$ with $S_k[u] > S_{sol}[u]$, and without loss of generality, suppose that k is the smallest such integer, i.e., that before that point our stratification was upper bounded by that of the unknown solution: $\forall j \in \{n \in \mathbb{N} \mid 0 \leq n < k\}, \forall v \in V : S_j[v] \leq S_{sol}[v]$. For u to be placed at a higher stratum by our algorithm there must exist an edge $(s,u) \in E$ such that either (i) (s,u) was a constraining ordinary path-edge: $S_k[u] = S_{k-1}[s] + 1$, or (ii) (s,u) was a *kp*-edge and $\forall p \in$ *proj*(s) : $S_k[u] = S_{k-1}[p] + 1$. In the first case, we have that: $S_{sol}[u] \leq S_{k-1}[s] \leq S_{sol}[s]$, and since (s,u) must also be present in the solution, this violates the single-direction edge property. In the second case, $S_{sol}[u] \leq S_{k-1}[p] \leq S_{sol}[p], \forall p \in$ *proj*(s), which leads to another contradiction, since there must exist a node $p \in$ *proj*(s), such that a path exists from p to u in the solution, which can only happen if p was placed at a strictly lower stratum than u. Thus, since all possible cases lead to a contradiction, we have proved our initial proposition: our algorithm always assigns to every node a stratum that is lower than, or equal to, that of any true solution of the problem instance. □

Let $s_{sol} = \sum_{v \in V} S_{sol}[v]$ and $s_i = \sum_{v \in V} S_i[v], \forall i \in \mathbb{N}$. It follows that $s_i \leq s_{sol}, \forall i \in \mathbb{N}$, for any such possible solution. Additionally, because of Lemma 1 and our two theorems, we know that each step but the last will strictly increase the sum of all strata values over all nodes. E.g., if our algorithm ended its execution at step n, then we would have: $s_0 < s_1 < ... < s_{n-1} = s_n$.

Suppose there is a solution but our algorithm fails to compute one (i.e., no fixpoint will ever be reached). Since s_i strictly increases at each step of our algorithm, and the only way to return is by finding a valid solution, we know that there will exist a step n, such that $s_n > s_{sol}$. However, this contradicts our proven proposition that $s_i \leq s_{sol}, \forall i \in \mathbb{N}$. Therefore, we conclude that if valid a solution exists, our algorithm will also be able to compute one.

□

Theorem 3 (Principality). *Any solution produced by Algorithm 4.1 will have a minimum number of strata. That is, for any possible solution of the problem instance, with t denoting the solution's total strata, we have that $n \leq t$, where n is the total number of strata produced by Algorithm 4.1.*

Proof. Let S_{sol} denote the stratification of a valid solution of the problem instance, and t its total number of strata. Without loss of generality we assume that strata are denoted as consecutive integers starting from 0, beginning from the lowest stratum. Thus, $\forall v \in V : S_{sol}[v] < t$.

Since a solution exists and *completeness* has been proved, we know that Algorithm 4.1 will also be able to terminate successfully at some step $n \in \mathbb{N}$, yielding its own solution. Let n_s be the total number of strata, and $x \in V$ be a node at the highest stratum of the solution computed by our algorithm. That is, $\forall v \in V : S_n[v] \leq S_n[x]$. Node x will also be the node that was changed last, which is at step $n - 1$, i.e., $S_n[x] = S_{n-1}[x] \neq S_{n-2}[x]$. Therefore, from Corollary 1, we have: $S_n[x] = S_{n-1}[x] = n - 1$. Since strata are consecutive integers starting from 0, we have that: $n_s = S_n[x] + 1 = n$.

According to Lemma 3, we have: $n = n_s = S_n[x] + 1 \leq S_{sol}[x] + 1 \leq t < t + 1$. Thus, we have shown that our algorithm minimizes the total number of strata it produces. □

REFERENCES

[1] Ole Agesen. “The cartesian product algorithm: simple and precise type inference of parametric polymorphism”. In: *Proc. of the 9th European Conf. on Object-Oriented Programming*. ECOOP ’95. Århus, Denmark: Springer, 1995, pp. 2–26. isbn: 3-540-60160-0.

[2] Karim Ali and Ondrej Lhoták. “Application-only call graph construction”. In: *Proc. of the 26th European Conf. on Object-Oriented Programming*. ECOOP ’12. Beijing, China: Springer, 2012, pp. 688–712. isbn: 978-3-642-31056-0. doi: `10.1007/978-3-642-31057-7_30`.

[3] Karim Ali and Ondrej Lhoták. “Averroes: whole-program analysis without the whole program”. In: *Proc. of the 27th European Conf. on Object-Oriented Programming*. ECOOP ’13. Montpellier, France: Springer, 2013, pp. 378–400. isbn: 978-3-642-39037-1. doi: `10.1007/978-3-642-39038-8_16`.

[4] Davide Ancona and Elena Zucca. “Principal typings for Java-like languages”. In: *Proc. of the 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’04. Venice, Italy: ACM, 2004, pp. 306–317. isbn: 1-58113-729-X. doi: `10.1145/964001.964027`.

[5] Davide Ancona et al. “Even more principal typings for Java-like languages”. In: *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP)*. 2004.

[6] Davide Ancona et al. “Polymorphic bytecode: compositional compilation for Java-like languages”. In: *Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: ACM, 2005, pp. 26–37. isbn: 1-58113-830-X. doi: `10.1145/1040305.1040308`.

[7] Lars O. Andersen. “Program analysis and specialization for the C programming language”. PhD thesis. DIKU, University of Copenhagen, 1994.

[8] *Apache Ant™*. `http://ant.apache.org/`.

[9] *Apache Maven software project management and comprehension tool*. `http://maven.apache.org/`.

[10] Dzintars Avots et al. “Improving software security with a C pointer analysis”. In: *Proc. of the 27th International Conf. on Software Engineering*. ICSE ’05. St. Louis, MO, USA: ACM, 2005, pp. 332–341. isbn: 1-58113-963-2. doi: `10.1145/1062455.1062520`.

[11] Gogul Balakrishnan and Thomas W. Reps. “Recency-abstraction for heap-allocated storage”. In: *Proc. of the 13th International Symp. on Static Analysis*. SAS ’06. Springer, 2006, pp. 221–239.

[12] Gogul Balakrishnan and Thomas W. Reps. “WYSINWYX: what you see is not what you execute”. In: *ACM Trans. on Programming Languanges and Systems* 32.6 (2010), 23:1–23:84. doi: `10.1145/1749608.1749612`.

[13] George Balatsouras and Yannis Smaragdakis. “Class hierarchy complementation: soundly completing a partial type graph”. In: *Proc. of the 28th annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 515–532. isbn: 978-1-4503-2374-1.

[14] George Balatsouras and Yannis Smaragdakis. “Structure-sensitive points-to analysis for C and C++”. In: *Proc. of the 23rd International Symp. on Static Analysis*. SAS ’16. Edinburgh, United Kingdom: Springer, 2016. isbn: 978-3-662-53412-0.

[15] Josh Berdine et al. “Shape analysis for composite data structures”. In: *Proc. of the 19th International Conf. on Computer Aided Verification*. Vol. 4590. CAV ’07. Berlin, Germany: Springer, 2007, pp. 178–192. isbn: 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_22.

[16] Marc Berndl et al. “Points-to analysis using BDDs”. In: *Proc. of the 2003 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’03. San Diego, California, USA: ACM, 2003, pp. 103–114. isbn: 1-58113-662-5. doi: 10.1145/781131.781144.

[17] Stephen M. Blackburn et al. “The DaCapo benchmarks: Java benchmarking development and analysis”. In: *Proc. of the 21st annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 169–190. isbn: 1-59593-348-4. doi: 10.1145/1167473.1167488.

[18] Eric Bodden et al. “Taming reflection: aiding static analysis in the presence of reflection and custom class loaders”. In: *Proc. of the 33rd International Conf. on Software Engineering*. ICSE ’11. Honolulu, HI, USA: ACM, 2011, pp. 241–250. isbn: 978-1-4503-0445-0. doi: 10.1145/1985793.1985827.

[19] 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 (June 1975 1975), pp. 234–245. issn: 0362-1340. doi: 10.1145/390016.808445.

[20] Martin Bravenboer and Yannis Smaragdakis. “Strictly declarative specification of sophisticated points-to analyses”. In: *Proc. of the 24th annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’09. Orlando, Florida, USA: ACM, 2009. isbn: 978-1-60558-766-0.

[21] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. “ASM: a code manipulation tool to implement adaptable systems”. In: *Adaptable and extensible component systems* 30 (2002).

[22] Cristiano Calcagno et al. “Compositional shape analysis by means of bi-abduction”. In: *Proc. of the 36th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’09. Savannah, GA, USA: ACM, 2009, pp. 289–300. isbn: 978-1-60558-379-2. doi: 10.1145/1480881.1480917.

[23] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. “Analysis of pointers and structures”. In: *Proc. of the 1990 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’90. White Plains, New York, USA: ACM, 1990, pp. 296–310. isbn: 0-89791-364-7. doi: 10.1145/93542.93585.

[24] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. “Precise analysis of string expressions”. In: *Proc. of the 10th International Symp. on Static Analysis*. SAS ’03. San Diego, CA, USA: Springer, 2003, pp. 1–18. isbn: 3-540-40325-6. doi: 10.1007/3-540-44898-5_1.

[25] Edmund M. Clarke and E. Allen Emerson. “Design and synthesis of synchronization skeletons using branching-time temporal logic”. In: *Logics of programs, workshop*. Ed. by Dexter Kozen. Vol. 131. LOP ’81. Yorktown Heights, New York, USA: Springer, 1981, pp. 52–71. isbn: 3-540-11212-X. doi: 10.1007/BFb0025774.

[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: A unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proc. of the 4th ACM Symp. on Principles of Programming Languages*. POPL ’77. Los Angeles, California, USA: ACM, 1977, pp. 238–252. isbn: 0-89791-692-1. doi: 10.1145/512950.512973.

[28] 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.

[29] 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.

[30] Patrick Cousot and Nicolas Halbwachs. “Automatic discovery of linear restraints among variables of a program”. In: *Proc. of the 5th ACM Symp. on Principles of Programming Languages*. POPL ’78. Tucson, Arizona, USA: ACM, 1978, pp. 84–96. doi: 10.1145/512760.512770.

[31] Barthélémy Dagenais and Laurie J. Hendren. “Enabling static analysis for partial Java programs”. In: *Proc. of the 23rd annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’08. Nashville, TN, USA: ACM, 2008, pp. 313–328. isbn: 978-1-60558-215-3. doi: 10.1145/1449764.1449790.

[32] Manuvir Das. “Unification-based pointer analysis with directional assignments”. In: *Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 35–46. isbn: 1-58113-199-2. doi: 10.1145/349299.349309.

[33] Constantinos Daskalakis et al. "Sorting and selection in posets". In: *Proc. of the 20th annual ACM-SIAM Symp. on Discrete Algorithms*. SODA '09. New York, New York: Society for Industrial and Applied Mathematics, 2009, pp. 392–401. url: http://dl.acm.org/citation.cfm?id=1496770.1496814.

[34] Arnab De and Deepak D'Souza. "Scalable flow-sensitive pointer analysis for Java with strong updates". In: *Proc. of the 26th European Conf. on Object-Oriented Programming*. ECOOP '12. Beijing, China: Springer, 2012, pp. 665–687. isbn: 978-3-642-31056-0. doi: 10.1007/978-3-642-31057-7_29.

[35] Danny Dig. "A refactoring approach to parallelism". In: *IEEE Software* 28.1 (2011), pp. 17–22. doi: 10.1109/MS.2011.1.

[36] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. "Context-sensitive interprocedural points-to analysis in the presence of function pointers". In: *Proc. of the 1994 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI '94. Orlando, Florida, USA: ACM, 1994, pp. 242–256. isbn: 0-89791-662-X.

[37] 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*. Ed. by J. W. de Bakker and Jan van Leeuwen. Vol. 85. ICALP '80. Noordwijkerhout, Netherlands: Springer, 1980, pp. 169–181. isbn: 3-540-10003-2. doi: 10.1007/3-540-10003-2_69.

[38] Manuel Fähndrich et al. "Partial online cycle elimination in inclusion constraint graphs". In: *Proc. of the 1998 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: ACM, 1998, pp. 85–96. isbn: 0-89791-987-4. doi: 10.1145/277650.277667.

[39] Pietro Ferrara, Raphael Fuchs, and Uri Juhasz. "TVAL+ : TVLA and value analyses together". In: *Proc. of the 2012 International Conf. on Software Engineering and Formal Methods*. SEFM '12. Thessaloniki, Greece: Springer, 2012, pp. 63–77. isbn: 978-3-642-33825-0. doi: 10.1007/978-3-642-33826-7_5.

[40] Stephen J. Fink et al. *WALA UserGuide: PointerAnalysis*. http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis.

[41] Robert W Floyd. "Assigning meanings to programs". In: *Proc. of Symp. in Applied Mathematics. Mathematical aspects of computer science*. (Providence, Rhode Island). Ed. by J. T. Schwartz. Vol. 19. American Mathematical Society, 1967, pp. 19–32.

[42] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. "Profile-guided static typing for dynamic scripting languages". In: *Proc. of the 24th annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA '09. Orlando, Florida, USA: ACM, 2009, pp. 283–300. isbn: 978-1-60558-766-0. doi: 10.1145/1640089.1640110.

[43] Rakesh Ghiya and Laurie J. Hendren. “Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C”. In: *Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’96. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 1–15. isbn: 0-89791-769-3. doi: 10.1145/237721.237724.

[44] James Gosling et al. *The Java™Language Specification, third edition*. Addison-Wesley Professional, 2005. isbn: 0321246780.

[45] *Gradle build automation system*. http://gradle.org/.

[46] Radu Grigore and Hongseok Yang. “Abstraction refinement guided by a learnt probabilistic model”. In: *Proc. of the 43rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’16. St. Petersburg, Florida, USA: ACM, 2016, pp. 485–498. isbn: 978-1-4503-3549-2. doi: 10.1145/2837614.2837663.

[47] 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.

[48] Bhargav S. Gulavani and Sriram K. Rajamani. “Counterexample driven refinement for abstract interpretation”. In: *Proc. of the 12th international conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Holger Hermanns and Jens Palsberg. Vol. 3920. TACAS ’06. Vienna, Austria: Springer, 2006, pp. 474–488. isbn: 3-540-33056-9. doi: 10.1007/11691372_34.

[49] Ben Hardekopf and Calvin Lin. “Exploiting pointer and location equivalence to optimize pointer analysis”. In: *Proc. of the 14th International Symp. on Static Analysis*. SAS ’07. Kongens Lyngby, Denmark: Springer, 2007, pp. 265–280. isbn: 978-3-540-74060-5.

[50] Ben Hardekopf and Calvin Lin. “Flow-sensitive pointer analysis for millions of lines of code”. In: *Proc. of the 9th International Symp. on Code Generation and Optimization*. CGO ’11. Chamonix, France: IEEE Computer Society, 2011, pp. 289–298. isbn: 978-1-61284-356-8. doi: 10.1109/CGO.2011.5764696.

[51] Ben Hardekopf and Calvin Lin. “Semi-sparse flow-sensitive pointer analysis”. In: *Proc. of the 36th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’09. Savannah, GA, USA: ACM, 2009, pp. 226–238. isbn: 978-1-60558-379-2. doi: 10.1145/1480881.1480911.

[52] Ben Hardekopf and Calvin Lin. “The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code”. In: *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: ACM, 2007, pp. 290–299. isbn: 978-1-59593-633-2.

[53] Nevin Heintze and Olivier Tardieu. “Ultra-fast aliasing analysis using CLA: a million lines of C code in a second”. In: *Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’01. Snowbird, Utah, USA: ACM, 2001, pp. 254–263. isbn: 1-58113-414-2. doi: 10.1145/378795.378855.

[54] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. isbn: 0321154916.

[55] Julien Henry, David Monniaux, and Matthieu Moy. “Succinct representations for abstract interpretation - combined analysis algorithms and experimental evaluation”. In: *Proc. of the 19th International Symp. on Static Analysis*. SAS ’12. Deauville, France: Springer, 2012, pp. 283–299. isbn: 978-3-642-33124-4. doi: `10.1007/978-3-642-33125-1_20`.

[56] Michael Hind et al. “Interprocedural pointer alias analysis”. In: *ACM Trans. on Programming Languages and Systems* 21.4 (1999), pp. 848–894. doi: `10.1145/325478.325519`.

[57] Martin Hirzel, Amer Diwan, and Michael Hind. “Pointer analysis in the presence of dynamic class loading”. In: *Proc. of the 18th European Conf. on Object-Oriented Programming*. ECOOP ’04. Oslo, Norway: Springer, 2004, pp. 96–122. isbn: 3-540-22159-X. doi: `10.1007/978-3-540-24851-4_5`.

[58] Martin Hirzel et al. “Fast online pointer analysis”. In: *ACM Trans. on Programming Languages and Systems* 29.2 (2007). doi: `10.1145/1216374.1216379`.

[59] C. A. R. Hoare. “An axiomatic basis for computer programming”. In: *Commun. ACM* 12.10 (1969), pp. 576–580. doi: `10.1145/363235.363259`.

[60] 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`.

[61] Samin S. Ishtiaq and Peter W. O’Hearn. “BI as an assertion language for mutable data structures”. In: *Proc. of the 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’01. London, UK: ACM, 2001, pp. 14–26. isbn: 1-58113-336-7.

[62] *Itanium C++ ABI*. `https://mentorembedded.github.io/cxx-abi/abi.html`.

[63] Neil D. Jones and Steven S. Muchnick. “Flow analysis and optimization of LISP-like structures”. In: *Proc. of the 6th ACM Symp. on Principles of Programming Languages*. POPL ’79. San Antonio, Texas, USA: ACM, 1979, pp. 244–256. doi: `10.1145/567752.567776`.

[64] John B. Kam and Jeffrey D. Ullman. “Monotone data flow analysis frameworks”. In: *Acta Informatica* 7 (1977), pp. 305–317. doi: `10.1007/BF00290339`.

[65] Vini Kanvar and Uday P. Khedker. “Heap abstractions for static analysis”. In: *CoRR* abs/1403.4910 (2014). url: `http://arxiv.org/abs/1403.4910`.

[66] George Kastrinis and Yannis Smaragdakis. “Hybrid context-sensitivity for points-to analysis”. In: *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’13. Seattle, WA, USA: ACM, 2013. isbn: 978-1-4503-2014-6.

[67] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. “Liveness-based pointer analysis”. In: *Proc. of the 19th International Symp. on Static Analysis*. SAS ’12. Deauville, France: Springer, 2012, pp. 265–282. isbn: 978-3-642-33124-4. doi: 10.1007/978-3-642-33125-1_19.

[68] Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis - theory and practice*. CRC Press, 2009. isbn: 978-0-8493-2880-0. url: http://www.crcpress.com/product/isbn/9780849328800.

[69] 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. doi: 10.1145/512927.512945.

[70] James C. King. “Symbolic execution and program testing”. In: *Commun. ACM* 19.7 (1976), pp. 385–394. doi: 10.1145/360248.360252.

[71] Thomas Kotzmann et al. “Design of the Java HotSpot™ client compiler for Java 6”. In: *ACM Trans. on Architecture and Code Optimization (TACO)* 5.1 (2008). doi: 10.1145/1369396.1370017.

[72] Chris Lattner and Vikram S. Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Proc. of the 2nd International Symp. on Code Generation and Optimization*. CGO ’04. San Jose, CA, USA: IEEE Computer Society, 2004, pp. 75–88. isbn: 0-7695-2102-9. doi: 10.1109/CGO.2004.1281665.

[73] Chris Lattner, Andrew Lenharth, and Vikram S. Adve. “Making context-sensitive points-to analysis with heap cloning practical for the real world”. In: *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: ACM, 2007, pp. 278–289. isbn: 978-1-59593-633-2. doi: 10.1145/1250734.1250766.

[74] Tal Lev-Ami and Shmuel Sagiv. “TVLA: A system for implementing static analyses”. In: *Proc. of the 7th International Symp. on Static Analysis*. SAS ’00. Santa Barbara, CA, USA: Springer, 2000, pp. 280–301. isbn: 3-540-67668-6. doi: 10.1007/978-3-540-45099-3_15.

[75] Ondřej Lhoták and Kwok-Chiang Andrew Chung. “Points-to analysis with efficient strong updates”. In: *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: ACM, 2011, pp. 3–16. isbn: 978-1-4503-0490-0.

[76] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. “KLOVER: A symbolic execution and automatic test generation tool for C++ programs”. In: *Proc. of the 23rd International Conf. on Computer Aided Verification*. Vol. 6806. CAV ’11. Snowbird, Utah, USA: Springer, 2011, pp. 609–615. isbn: 978-3-642-22109-5. doi: 10.1007/978-3-642-22110-1_49.

[77] Lian Li, Cristina Cifuentes, and Nathan Keynes. “Boosting the performance of flow-sensitive points-to analysis using value flow”. In: *Proc. of the 19th ACM SIGSOFT International Symp. on Foundations of Software Engineering*. Szeged, Hungary: ACM, 2011, pp. 343–353. isbn: 978-1-4503-0443-6. doi: 10.1145/2025113.2025160.

[78] Lian Li, Cristina Cifuentes, and Nathan Keynes. “Practical and effective symbolic analysis for buffer overflow detection”. In: *Proc. of the 18th ACM SIGSOFT International Symp. on Foundations of Software Engineering*. Santa Fe, New Mexico, USA: ACM, 2010, pp. 317–326. isbn: 978-1-60558-791-2. doi: 10.1145/1882291.1882338.

[79] Lian Li, Cristina Cifuentes, and Nathan Keynes. “Precise and scalable context-sensitive pointer analysis via value flow graph”. In: *Proc. of the 2013 International Symp. on Memory Management*. ISMM ’13. Seattle, Washington, USA: ACM, 2013, pp. 85–96. isbn: 978-1-4503-2100-6. doi: 10.1145/2464157.2466483.

[80] Yue Li et al. “Self-inferencing reflection resolution for Java”. In: *Proc. of the 28th European Conf. on Object-Oriented Programming*. ECOOP ’14. Uppsala, Sweden: Springer, 2014, pp. 27–53. isbn: 978-3-662-44201-2.

[81] Percy Liang and Mayur Naik. “Scaling abstraction refinement via pruning”. In: *Proc. of the 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: ACM, 2011, pp. 590–601. isbn: 978-1-4503-0663-8. doi: 10.1145/1993498.1993567.

[82] Percy Liang, Omer Tripp, and Mayur Naik. “Learning minimal abstractions”. In: *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: ACM, 2011, pp. 31–42. isbn: 978-1-4503-0490-0. doi: 10.1145/1926385.1926391.

[83] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. isbn: 0201432943.

[84] Benjamin Livshits. “Improving software security with precise static and runtime analysis”. PhD thesis. Stanford University, 2006.

[85] Benjamin Livshits, John Whaley, and Monica S. Lam. “Reflection analysis for Java”. In: *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*. APLAS ’05. Tsukuba, Japan: Springer, 2005, pp. 139–160. isbn: 3-540-29735-9. doi: 10.1007/11575467_11.

[86] Benjamin Livshits et al. “In defense of soundiness: a manifesto”. In: *Communications of the ACM* 58.2 (Jan. 2015), pp. 44–46. issn: 0001-0782. doi: 10.1145/2644805.

[87] Francesco Logozzo and Manuel Fähndrich. “Pentagons: a weakly relational abstract domain for the efficient validation of array accesses”. In: *Proc. of the 2008 ACM Symp. on Applied Computing*. Fortaleza, Ceara, Brazil: ACM, 2008, pp. 184–188. isbn: 978-1-59593-753-7. doi: 10.1145/1363686.1363736.

[88] Magnus Madsen, Benjamin Livshits, and Michael Fanning. “Practical static analysis of JavaScript applications in the presence of frameworks and libraries”. In: *Proc. of the ACM SIGSOFT International Symp. on the Foundations of Software Engineering*. FSE ’13. Saint Petersburg, Russian Federation: ACM, 2013, pp. 499–509. isbn: 978-1-4503-2237-9. doi: 10.1145/2491411.2491417.

[89] Roman Manevich et al. “Partially disjunctive heap abstraction”. In: *Proc. of the 11th International Symp. on Static Analysis*. SAS ’04. Verona, Italy: Springer, 2004, pp. 265–279. isbn: 3-540-22791-1. doi: 10.1007/978-3-540-27864-1_20.

[90] Matthew Might, Yannis Smaragdakis, and David Van Horn. “Resolving and exploiting the *k*-CFA paradox: illuminating functional vs. object-oriented program analysis”. In: *Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’10. Toronto, Ontario, Canada: ACM, 2010, pp. 305–315. isbn: 978-1-4503-0019-3. doi: 10.1145/1806596.1806631.

[91] 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: 10.1145/1044834.1044835.

[92] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. “Parameterized object sensitivity for points-to and side-effect analyses for Java”. In: *Proc. of the 2002 International Symp. on Software Testing and Analysis*. ISSTA ’02. Roma, Italy: ACM, 2002, pp. 1–11. isbn: 1-58113-562-9. doi: 10.1145/566172.566174.

[93] Antoine Miné. “A new numerical abstract domain based on difference-bound matrices”. In: *Proc. of the 2nd Symp. on Programs as Data Objects*. Vol. 2053. PADO ’01. Aarhus, Denmark: Springer, 2001, pp. 155–172. isbn: 3-540-42068-1. doi: 10.1007/3-540-44978-7_10.

[94] Antoine Miné. “Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics”. In: *Proc. of the 2006 ACM SIGPLAN/SIGBED conf. on Languages, Compilers, and Tools for Embedded Systems*. LCTES ’06. Ottawa, Ontario, Canada: ACM, 2006, pp. 54–63. isbn: 1-59593-362-X. doi: 10.1145/1134650.1134659.

[95] Antoine Miné. “The Octagon abstract domain”. In: *Higher-Order and Symbolic Computation* 19.1 (2006), pp. 31–100. doi: 10.1007/s10990-006-8609-1. url: http://dx.doi.org/10.1007/s10990-006-8609-1.

[96] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997. isbn: 1-55860-320-4.

[97] Mayur Naik, Alex Aiken, and John Whaley. “Effective static race detection for Java”. In: *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’06. Ottawa, Ontario, Canada: ACM, 2006, pp. 308–319. isbn: 1-59593-320-4. doi: 10.1145/1133981.1134018.

[98] Rupesh Nasre. “Exploiting the structure of the constraint graph for efficient points-to analysis”. In: *Proc. of the 2012 International Symp. on Memory Management*. ISMM ’12. Beijing, China: ACM, 2012, pp. 121–132. isbn: 978-1-4503-1350-6. doi: 10.1145/2258996.2259013.

[99] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. “Importance of heap specialization in pointer analysis”. In: *Proc. of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’04. Washington DC, USA: ACM, 2004, pp. 43–48. isbn: 1-58113-910-1. doi: 10.1145/996821.996836.

[100] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. IC/2004/64. EPFL Lausanne, Switzerland, 2004.

[101] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. “Learning a strategy for adapting a program analysis via bayesian optimisation”. In: *Proc. of the 30th annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’15. Pittsburgh, Pennsylvania, USA: ACM, 2015, pp. 572–588. isbn: 978-1-4503-3689-5. doi: 10.1145/2814270.2814309.

[102] 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. doi: 10.1007/3-540-44802-0_1.

[103] 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*. Ed. by Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto. ASE ’10. Antwerp, Belgium: ACM, 2010, pp. 179–180. isbn: 978-1-4503-0116-9. doi: 10.1145/1858996.1859035.

[104] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. “Efficient field-sensitive pointer analysis for C”. In: *Proc. of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’04. Washington DC, USA: ACM, 2004, pp. 37–42. isbn: 1-58113-910-1. doi: 10.1145/996821.996835.

[105] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. “Efficient field-sensitive pointer analysis of C”. In: *ACM Trans. on Programming Languages and Systems* 30.1 (2007). doi: 10.1145/1290520.1290524.

[106] Jean-Pierre Queille and Joseph Sifakis. “Specification and verification of concurrent systems in CESAR”. In: *Proc. of the 5th International Symp. on Programming*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Vol. 137. Torino, Italy: Springer, 1982, pp. 337–351. isbn: 3-540-11494-7. doi: 10.1007/3-540-11494-7_22.

[107] Thomas W. Reps. “Program analysis via graph reachability”. In: *Information & Software Technology* 40 (1998), pp. 701–726. doi: 10.1016/S0950-5849(98)00093-7.

[108] 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. doi: 10.1145/199448.199462.

[109] John C. Reynolds. "Intuitionistic reasoning about shared mutable data structure". In: *Millennial perspectives in computer science*. Ed. by Jim Davies, Bill Roscoe, and Jim Woodcock. Houndsmill, Hampshire: Palgrave, 2000, pp. 303–321.

[110] 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: IEEE Computer Society, 2002, pp. 55–74. isbn: 0-7695-1483-9. doi: 10.1109/LICS.2002.1029817.

[111] Noam Rinetzky and Shmuel Sagiv. "Interprocedural shape analysis for recursive programs". In: *Proc. of the 10th International Conf. on Compiler Construction*. CC '01. Genova, Italy: Springer, 2001, pp. 133–149. isbn: 3-540-41861-X. doi: 10.1007/3-540-45306-7_10.

[112] Atanas Rountev and Satish Chandra. "Off-line variable substitution for scaling points-to analysis". In: *Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 47–56. isbn: 1-58113-199-2. doi: 10.1145/349299.349310.

[113] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. "Points-to analysis for Java using annotated constraints". In: *Proc. of the 16th annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA '01. Tampa, Florida, USA: ACM, 2001, pp. 43–55. isbn: 1-58113-335-9. doi: 10.1145/504282.504286.

[114] Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Parametric shape analysis via 3-valued logic". In: *ACM Trans. on Programming Languages and Systems* 24.3 (May 2002), pp. 217–298. issn: 0164-0925. doi: 10.1145/514188.514190.

[115] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Parametric shape analysis via 3-valued logic". In: *Proc. of the 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: ACM, 1999, pp. 105–118. isbn: 1-58113-095-3. doi: 10.1145/292540.292552.

[116] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Solving shape-analysis problems in languages with destructive updating". In: *ACM Trans. on Programming Languages and Systems* 20.1 (1998), pp. 1–50. doi: 10.1145/271510.271517.

[117] Micha Sharir and Amir Pnueli. "Two approaches to interprocedural data flow analysis". In: *Program flow analysis: theory and applications*. Ed. by Steven S. Muchnick and Neil D. Jones. Englewood Cliffs, NJ: Prentice-Hall, 1981. Chap. 7, pp. 189–234. isbn: 0137296819.

[118] Micha Sharir and Amir Pnueli. “Two approaches to interprocedural data flow analysis”. In: *Program flow analysis: theory and applications*. Ed. by Steven S. Muchnick and Neil D. Jones. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981. Chap. 7, pp. 189–233. isbn: 0137296819.

[119] Olin Shivers. “Control-flow analysis of higher-order languages”. PhD thesis. Carnegie Mellon University, 1991.

[120] 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. url: http://dx.doi.org/10.1561/2500000014.

[121] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. “More sound static handling of Java reflection”. In: *Proc. of the 13th Asian Symp. on Programming Languages and Systems*. APLAS ’15. Pohang, South Korea: Springer, 2015. isbn: 978-3-319-26528-5.

[122] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. “Set-based preprocessing for points-to analysis”. In: *Proc. of the 28th annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 253–270. isbn: 978-1-4503-2374-1. doi: 10.1145/2509136.2509524.

[123] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. “Pick your contexts well: understanding object-sensitivity”. In: *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: ACM, 2011, pp. 17–30. isbn: 978-1-4503-0490-0.

[124] Yannis Smaragdakis et al. “More sound static handling of Java reflection”. In: *Proc. of the 13th Asian Symp. on Programming Languages and Systems*. APLAS ’15. Pohang, South Korea: Springer, 2015, pp. 485–503. isbn: 978-3-319-26528-5. doi: 10.1007/978-3-319-26529-2_26.

[125] Manu Sridharan et al. “Demand-driven points-to analysis for Java”. In: *Proc. of the 20th annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’05. San Diego, CA, USA: ACM, 2005, pp. 59–76. isbn: 1-59593-031-0. doi: 10.1145/1094811.1094817.

[126] Codruţ Stancu et al. “Comparing points-to static analysis with runtime recorded profiling data”. In: *Proc. of the 2014 International Conf. on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools*. PPPJ ’14. Cracow, Poland: ACM, 2014, pp. 157–168. isbn: 978-1-4503-2926-2. doi: 10.1145/2647508.2647524.

[127] Robert F. Stärk and Joachim Schmid. *The problem of bytecode verification in current implementations of the JVM*. Tech. rep. ETH Zürich, 2000.

[128] Yulei Sui and Jingling Xue. “On-demand strong update analysis via value-flow refinement”. In: *Proc. of the 24th ACM SIGSOFT International Symp. on Foundations of Software Engineering*. Seattle, Washington, USA: ACM, 2016, pp. 460–473. isbn: 978-1-4503-4218-6. doi: 10.1145/2950290.2950296.

[129] Yulei Sui and Jingling Xue. “SVF: interprocedural static value-flow analysis in LLVM”. In: *Proc. of the 25th International Conf. on Compiler Construction*. CC ’16. Barcelona, Spain: ACM, 2016, pp. 265–266. isbn: 978-1-4503-4241-4. doi: 10.1145/2892208.2892235.

[130] Raja Vallée-Rai et al. “Optimizing Java bytecode using the Soot framework: is it feasible?” In: *Proc. of the 9th International Conf. on Compiler Construction*. CC ’00. Berlin, Germany: Springer, 2000, pp. 18–34. isbn: 3-540-67263-X.

[131] 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.

[132] John Whaley and Martin C. Rinard. “Compositional pointer and escape analysis for Java programs”. In: *Proc. of the 14th annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’99. Denver, Colorado, USA: ACM, 1999, pp. 187–206. isbn: 1-58113-238-7. doi: 10.1145/320384.320400.

[133] John Whaley et al. “Using Datalog with binary decision diagrams for program analysis”. In: *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*. APLAS ’05. Tsukuba, Japan: Springer, 2005, pp. 97–118. isbn: 3-540-29735-9. doi: 10.1007/11575467_8.

[134] Sen Ye, Yulei Sui, and Jingling Xue. “Region-based selective flow-sensitive pointer analysis”. In: *Proc. of the 21st International Symp. on Static Analysis*. SAS ’14. Munich, Germany: Springer, 2014, pp. 319–336. isbn: 978-3-319-10935-0. doi: 10.1007/978-3-319-10936-7_20.

[135] Qirun Zhang et al. “Fast algorithms for Dyck-CFL-reachability with applications to alias analysis”. In: *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’13. Seattle, WA, USA: ACM, 2013, pp. 435–446. isbn: 978-1-4503-2014-6. doi: 10.1145/2462156.2462159.

[136] Xin Zhang, Mayur Naik, and Hongseok Yang. “Finding optimum abstractions in parametric dataflow analysis”. In: *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’13. Seattle, WA, USA: ACM, 2013, pp. 365–376. isbn: 978-1-4503-2014-6. doi: 10.1145/2462156.2462185.

[137] Xin Zhang et al. “On abstraction refinement for program analyses in Datalog”. In: *Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: ACM, 2014, pp. 239–248. isbn: 978-1-4503-2784-8. doi: 10.1145/2594291.2594327.

[138] Xin Zheng and Radu Rugina. “Demand-driven alias analysis for C”. In: *Proc. of the 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA: ACM, 2008, pp. 197–208. isbn: 978-1-59593-689-9. doi: 10.1145/1328438.1328464.