Referenzziel-Analyse für Java-Bibliotheken

Dinh Khoi Le, Michael Thies und Peter Pfahler
{le,mthies,peter}@upb.de
Universität Paderborn, Fachbereich 17, Fürstenallee 11
D-33102 Paderborn, Deutschland

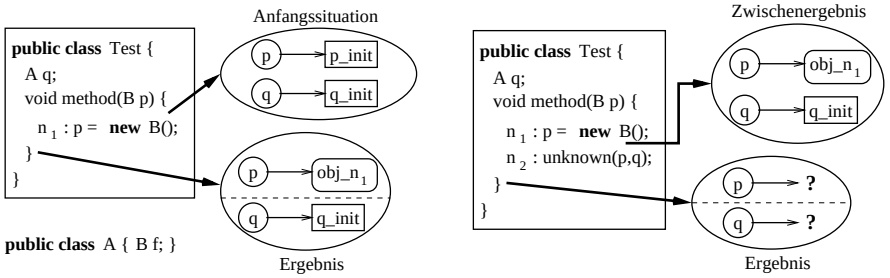
Abstract. Referenzziel-Analyse (*Points-to analysis*) beschreibt, wie ein interprozedurales und fluss-sensitives Analyseverfahren auf unvollständige Java-Programme (Bibliotheken), die als Bytecode vorliegen, statisch angewandt werden kann. Mit Hilfe der Analyse lassen sich Informationen über die Menge der Objekte gewinnen, die eine Programmvariable referenzieren kann. Ein neuartiger Ansatz analysiert jeweils eine isolierte Java-Bibliothek. Fehlende Informationen über Methoden außerhalb der untersuchten Bibliothek werden durch *Modelle* exakt repräsentiert. Zur Laufzeit wird die vollständige Information zusammengesetzt und das Programm dynamisch optimiert [9].

Das Verfahren wurde prototypisch in Java implementiert und auf einige größere Java-Programme und Java-Bibliotheken angewandt. Die Ergebnisse zeigen, dass Referenzziel-Analyse für unvollständige Java-Programme praktikabel und sinnvoll ist. Beispielsweise beim statischen Auflösen dynamischer Methodenbindung liefert die Referenzziel-Analyse bessere Ergebnisse als die konventionelle Klassenhierarchie-Analyse.

1 Einleitung

Da in Java alle Objekte über Referenzen angesprochen werden, ist die Programmanalyse dafür genauso schwierig wie für Programmiersprachen mit typsicheren Zeigern. Durch die Referenzziel-Analyse lassen sich Informationen über die Menge der Objekte gewinnen, die eine Programmvariable an einer Programmstelle referenzieren kann. Viele spezielle Analysen, wie die statische Auflösung von dynamisch gebundenen Methoden, *Alias*-Analyse und Typinferenz können die Referenzziel-Ergebnisse weiterverwenden. Analysiert man eine isolierte Java-Bibliothek, können die Resultate zusammen mit dieser Bibliothek wiederverwendet werden. Da die Analyse für Java-Bibliotheken auf Bytecode-Ebene geschieht, lässt sich auch kommerzielle Software analysieren und damit für eine Optimierung vorbereiten.

Die Referenzziel-Analyse liefert für jede Methode als Ergebnis eine Menge von Datenflusselementen. Ein Datenflusselement repräsentiert eine Programmvariable und die Objektmenge, die diese Variable am Ende der Methode referenzieren kann. Die



(a) Referenzziel-Analyse (b) Aufruf unbekannter Methode

Abbildung 1. Beispiel für Referenzziel-Analyse

Abbildung 1(a) verdeutlicht dies: Das Ergebnis für die Methode *method* besteht aus zwei Datenflusselementen, je eins für die Variablen *p* und *q*. Am Ende von *method* referenziert *p* das an der Stelle n_1 erzeugte Objekt, *q* dagegen seinen unbekannten Anfangswert, in der Abbildung als *q_init* bezeichnet.

Jede Bibliothek bietet eine öffentliche Schnittstelle für sie benutzende Programme und benutzt ihrerseits andere Bibliotheken. Die aufgerufenen Methoden, die nicht zur Bibliothek selbst gehören, werden als unbekannte externe Methoden bezeichnet und stellen somit offene Schnittstellen dar. Da die Referenzziel-Analyse jeweils eine Softwarebibliothek als Einheit betrachtet, fehlen an solchen Stellen die Analyseinformationen (Abbildung 1(b)). Durch Einsatz von *Modellen* für die Auswirkungen solcher Methoden können die fehlenden Informationen substituiert werden. Bei der späteren dynamischen Optimierung werden die entsprechenden Informationen über andere Bibliotheken eingesetzt.

Dieses Papier ist wie folgt aufgebaut: Kapitel 2 stellt kurz das zugrundeliegende Verfahren für die Referenzziel-Analyse vollständiger Programme vor. Im Anschluß wird die Anwendung des Verfahrens auf unvollständige Java-Programme (Bibliotheken) konzeptionell beschrieben. Kapitel 4 präsentiert die Ergebnisse, insbesondere einen Vergleich der Referenzziel-Ergebnisse mit den analogen Ergebnissen der Klassenhierarchie-Analyse. Das anschließende Kapitel diskutiert verwandte Arbeiten. Eine Zusammenfassung schließt das Papier ab.

2 Die Referenzziel-Analyse

Ein wichtiges Problem bei der Analyse von Java-Programmen ist die Bestimmung von konkreten Programmobjekten, auf die eine Objektreferenz verweisen kann. Die Referenzziel-Analyse (*Points-to analysis*) ist ein modulares Analyseverfahren für statisch typisierte, objekt-orientierte Programmiersprachen wie Java. Das Problem der Referenzziel-Analyse soll mittels abstrakter Werte, wie in [2] beschrieben, gelöst werden.

Der hier vorgestellte Algorithmus ist interprozedural und fluss-sensitiv. Er benutzt einen iterativen *Worklist*-Algorithmus für die Datenfluss-Analyse, der die Datenflusselemente in vier verschiedenen Phasen propagiert. Als Eingabe dient der ICFG [6] (*Interprocedural Control Flow Graph*). Die vier Phasen des Algorithmus erledigen folgende Aufgaben:

Phase 0: Konstruiert durch Auflösung der dynamischen Methodenbindung eine konservative Abschätzung des ICFGs, also des Anfangsaufrufgraphen. Dies geschieht mit Hilfe der Klassenhierarchie-Analyse [5]. Durch Anwendung des Algorithmus aus [3] wird der Anfangsaufrufgraph in starke Zusammenhangskomponenten zerlegt. So entsteht in linearer Zeit der gerichtete azyklische SCC-DAG (*Strongly Connected Component - Directed Acyclic Graph*).

Phase 1: Besucht den SCC-DAG in umgekehrter topologischer Ordnung und analysiert jede Methode unter der Annahme, dass alle Parameter und globalen Variablen zunächst unbekannte, abstrakte Anfangswerte haben. Für jede Methode berechnet der Algorithmus ausgehend von diesen unbekannten Werten, die Transferfunktion. Die Transferfunktion einer Methode bildet Mengen von Datenflusselementen auf ebensolche Mengen ab und beschreibt, wie die aufgerufene Methode die Datenflusselemente verändert. Transferfunktionen für Methoden innerhalb einer SCC weisen zyklische Abhängigkeiten auf. Sie werden simultan und iterativ berechnet, bis ein Fixpunkt erreicht wird. Im Gegensatz dazu weisen die Transferfunktionen zu Methoden aus verschieden SCCs nur hierarchische Abhängigkeiten auf und können in einem Aufwärtsdurchgang durch den SCC-DAG direkt berechnet werden. Als Ergebnis dieser Phase liegen die Transferfunktionen für alle Methoden und die Referenzziel-Lösungen für jede Programmstelle in Form von Datenflusselementen vor. Diese können unbekannte Anfangswerte enthalten.

Phase 2: Besucht den SCC-DAG in topologischer Ordnung und propagiert die konkreten Werte von jeder Aufrufstelle zu den unbekannten Anfangswerten an den Methodenanfängen. Diese Phase betrachtet nur erreichbare Methoden.

Phase 3: Die in der Phase 1 berechneten unbekannten Anfangswerte werden mit ihren in Phase 2 berechneten konkreten Werten instanziiert und vom Methodenanfang ausgehend propagiert. Diese Phase ist vollständig bedarfsgesteuert. Die Berechnung erfolgt nur an solchen Programmstellen, an denen man die endgültige Lösung benötigt. Nach der Phase 3 liegt die Referenzziel-Lösung an jeder Programmstelle in Form der Programmvariablen und Stellvertreter vor. Ein Stellvertreter repräsentiert eine Äquivalenzklasse aller an einer Programmstelle erzeugten Objekte.

Für von mehreren *Threads* zugreifbare Objekte liefert das Verfahren keine präzise Lösung. Details zum Algorithmus finden sich in [7]. Die Referenzziel-Analyse ist eine sehr aufwändige Aufgabe. Deshalb kann sie nur sinnvoll statisch durchgeführt werden, wenn ihre Ergebnisse über mehrere Programmläufe hinweg wiederverwendbar gespeichert bleiben. Diese wertvolle Information bereitet dynamische Optimierungen vor. Als praktikabler Analysekontext bietet sich die größte als unteilbar zu behandelnde Softwarestruktur, die Bibliothek, an. Um das Verfahren auf isolierte Bibliotheken anwenden zu können, muss es, wie im nächsten Abschnitt beschrieben, modifiziert werden.

3 Anwendung auf Java-Bibliotheken

Dieser Abschnitt zeigt die Verallgemeinerung des oben beschriebenen Referenzziel-Verfahrens für unvollständige Java-Programme (Bibliotheken). Wie in Abbildung 1(b) gezeigt fehlen an der Aufrufstelle einer unbekannten externen Methode die Analyseinformationen. Wir betrachten nun die Aufrufstelle einer Methode genauer: Der Aufruf einer Methode verursacht im allgemeinen Seiteneffekte, die bei der Berechnung der Datenflussinformation berücksichtigt werden müssen. Bei der klassischen interprozeduralen Analyse wird die Auswirkung der Methodenaufrufe auf die Datenflussinformation bestimmt. Eine Transferfunktion beschreibt diese Auswirkungen für die aufgerufene Methode. Für Methoden, die innerhalb der Bibliothek bekannt sind — d.h., sie sind in der Bibliothek definiert und werden *direkt* aufgerufen — kann die Transferfunktion unmittelbar bestimmt werden. Dadurch stellt man fest, welche Datenflusselemente durch einen Aufruf der Methode direkt oder indirekt verändert werden können (Abbildung 2).

Soll die Referenzziel-Analyse eine einzelne Bibliothek isoliert betrachten, muss sie auch Aufrufstellen unbekannter Methoden behandeln. Eine Methode ist unbekannt, wenn sie außerhalb der Bibliothek definiert ist oder außerhalb der Bibliothek überschrieben werden kann (siehe Abbildung 5(b)). Eine unbekannte Methode hat Zugriff auf ihre aktuellen Parameter und die für sie sichtbaren globalen Variablen, d.h. nur über solche Variablen können erreichbare Objekte von der unbekannten Methode modifiziert werden. Die Analyse verfügt aber nicht über die Kenntniss, ob und wie die Elemente innerhalb der Methode verändert werden. Deshalb werden die Auswirkungen eines Aufrufs dieser unbekannten Methode durch ein zunächst noch nicht vollständig bestimmtes Funktional ersetzt. Die Datenflusselemente, die nach der Aufrufstelle einer solchen Methode modifiziert worden sein können, erhalten in ihren Objektmengen zusätzlich einen speziellen Repräsentanten. Dieser stellt das auf die Referenzzielaspekte spezialisierte *Modell* der Methode dar. Bei dem Repräsentanten handelt es sich um ein Funktional, das die Substitution der formalen durch die aktuellen Parameter an der konkreten Aufrufstelle beschreibt und auf die noch unbekannte Transferfunktion Bezug nimmt. Es ist also abhängig vom Aufrufkontext.

p und *q* (vergleiche Abbildung 1(b)) haben in ihrer Objektmenge einen Repräsentanten für die unbekannte Methode *unknown* (Abbildung 3). Dieses *Modell* verar-

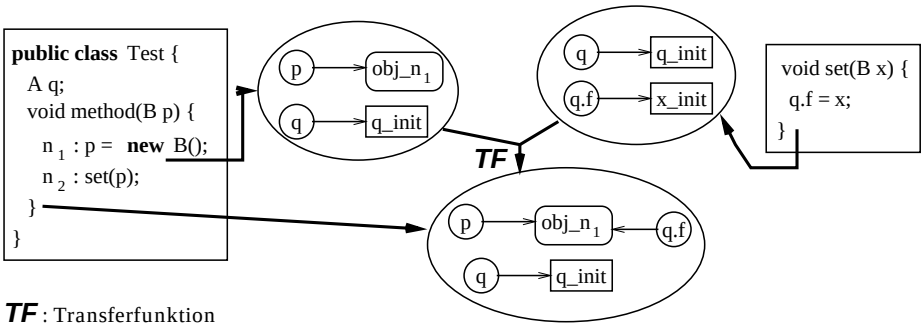


Abbildung 2. Anwendung der Transferfunktion (TF) für eine bekannte Methode

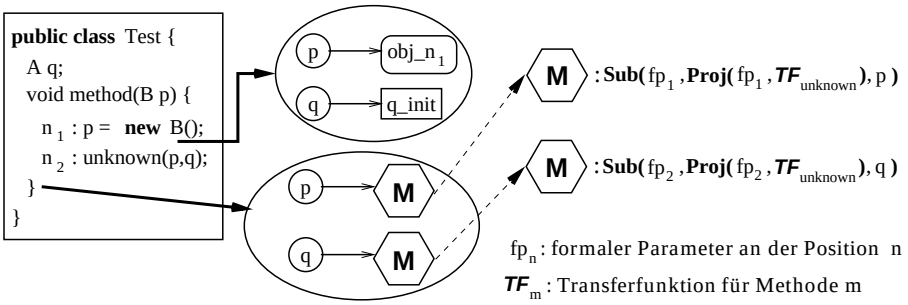


Abbildung 3. Einsatz des *Modells* für die unbekannte Methode *unknown*

beitet die Information aus der noch unbekannten Transferfunktion $TF_{unknown}$, projeziert also das Funktionsergebnis auf den jeweiligen formalen Parameter bzw. die für diese Methode sichtbaren globalen Variablen. Außerdem beschreibt es die Substitution der formalen durch die aktuellen Parameter. Wenn die Transferfunktion später bekannt wird, kann das *Modell* vollständig ausgewertet werden.

Nach der Aufrufstelle wird die Berechnung der Transferfunktion für die aufrufende Methode weiter fortgesetzt, wobei mehrere derartige *Modelle* vorkommen können: in der Regel eines für jede Aufrufstelle, an der eine bestimmte Variable als aktueller Parameter auftritt. Beim Besuch aller Knoten des Aufrufgraphen in *bottom-up*-Ordnung berechnet der Referenzziel-Algorithmus auf diese Weise die Transferfunktion für jede Methode aus den bereits bestimmten Transferfunktionen für die von ihr aufgerufenen Bibliotheks-Methoden. Dadurch wird das ganze Datenflußgleichungssystem aufgestellt. Es enthält Anfangswerte und *Modelle* als Unbekannte und Repräsentanten für Objekte als Konstante. Durch algebraische Vereinfachung können einige *Modelle* entfallen. Ziel der Analyse ist es, dieses Gleichungssystem soweit wie möglich zu lösen, indem die unbekannten Anfangswerte durch Objektrepräsentanten ersetzt werden. Die wichtigen charakteristischen Informationen der Funktionale, wie zum Beispiel die Identität der aufgerufenen Methode und die Information über formale und aktuelle Parameter, werden aufbewahrt, so dass sie später einfach mit den Informationen über andere Bibliotheken zusammengesetzt werden können. Abbildung 4 führt das Beispiel aus Abbildung 3 fort und verdeutlicht die Auswertung eines Modells nach Bekanntwerden der Transferfunktion. Vor der Komposition kann man evtl. die Referenzziel-Ergebnisse für bestimmte Analyseaufgabe spezialisieren, z.B. für Lösung der Typinferenz sind nur die Typinformationen der Objekte relevant, nicht die Objektidentität.

4 Die Referenzziel-Ergebnisse

Die Referenzziel-Analyse wurde in Java prototypisch implementiert und in eine an der Universität Paderborn entwickelte Programmanalyse-Umgebung eingebettet. Das Verfahren wurde unter anderem für zwei Java-Bibliotheken und ein zugehöriges Anwendungsprogramm eingesetzt: die *JavaClass*-Bibliothek [4] in der Version 3.2.4 (285 Klassen, 1579 Methoden und 438208 Bytes groß); unsere Programmanalyse-

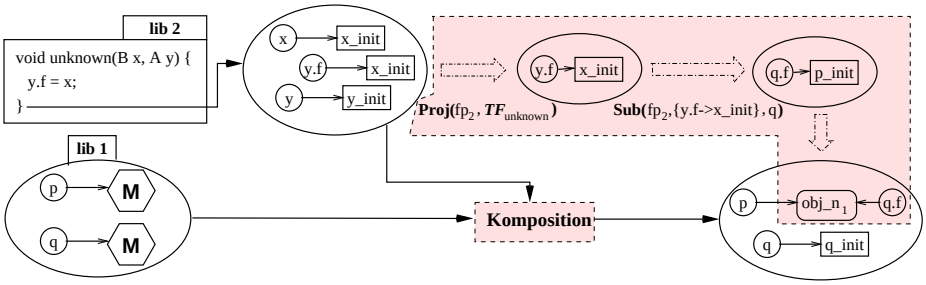


Abbildung 4. Die Komposition der Informationen

Umgebung (*Analysis*-Bibliothek: 225 Klassen, 1960 Methoden und 582691 Bytes groß) [7]. Da sie unter anderem *JavaClass* benutzt, dient sie für die Referenzziel-Analyse sowohl als eine Anwendung von *JavaClass* als auch als eine eigenständige Bibliothek; das Anwendungsprogramm *Jopt* (13 Klassen, 85 Methoden mit 37043 Bytes groß), ein statischer Optimierer für Java-Klassendateien und dient auch als ein Anwendungsprogramm für die *JavaClass*-Bibliothek.

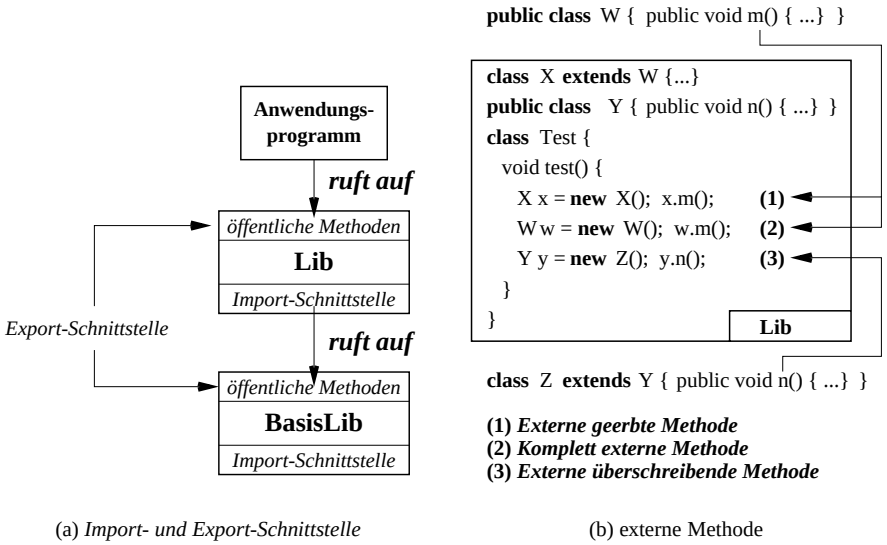
Die Evaluation der Referenzziel-Ergebnisse für mehrere Java-Bibliotheken und Java-Programme ist sehr ausführlich in [7] beschrieben. Dieses Papier greift nur die Referenzziel-Ergebnisse bezüglich der offenen Schnittstelle der oben genannten Programme auf.

4.1 Die Bibliothek-übergreifende Struktur

Für eine Java-Bibliothek gibt es zwei Arten von offenen Schnittstellen. Die erste ist die *Export-Schnittstelle*. Sie besteht aus Methoden, die von anderen Programmen oder Bibliotheken aufgerufen werden können, und Klassen, von denen andere Klassen in anderen Programmen erben können. Die öffentlichen Bibliotheks-Klassen stellen eine Vererbungsgrenze zwischen zwei Softwarekomponenten dar. An dieser Grenze kann man die Information für die Methoden-*Modelle* ermitteln, die Aufrufe der exportierten Methoden der Klassen außerhalb der Bibliothek repräsentieren.

Durch die Sichtbarkeit der Klassen und Methoden kann man grob abschätzen, wie breit die öffentlichen Schnittstellen einer Bibliothek sind. Nach den Messwerten machen alle analysierten Programme und Bibliotheken durchschnittlich über 78% ihrer Methoden und 76% ihrer Klassen nach außen sichtbar. Nach der Referenzziel-Analyse stehen dann die Informationen über die Programmobjekte dieser Schnittstellen für die Komposition zur Verfügung. In der Regel werden nicht alle dieser Informationen beim Zusammensetzen der Ergebnisse mit einem anderen Programm benötigt. Der Anteil hängt noch von der Breite der tatsächlich ausgenutzten Schnittstellen auf der Anwendungsseite ab.

Die zweite Art der offenen Schnittstellen sind die *Import-Schnittstellen*. Sie werden durch die Aufrufstellen innerhalb der Bibliothek gebildet, die sich auf externe Methoden beziehen. Dadurch läßt sich feststellen, wie eng diese Bibliothek in Beziehung zu anderen Bibliotheken steht, und welche Informationen über welche externen Methoden



(a) *Import- und Export-Schnittstelle* (b) externe Methode

Abbildung 5. Offene Schnittstellen einer Bibliothek

an diesen Stellen eingesetzt werden müssen. Abbildung 5 verdeutlicht die beiden Arten von offenen Schnittstellen einer Bibliothek und die möglichen Arten externer Methoden.

Das Erben von einer externen Klasse führt zum Fall (1). Nach der Untersuchung erben über 90% aller Klassen in den analysierten Programmen von den Klassen der Java-Standard-Bibliothek, vor allem von *java.lang.Object*. Wenn die Standard-Bibliothek ausgeschlossen wird, trägt das Erben von Bibliotheks-Klassen kaum zur Breite der Schnittstellen bei. Der entscheidende Faktor für die Breite der Schnittstellen zwischen den Programmteilen ist die Anzahl der Aufrufstellen von *komplett externen Methoden (2)* und *externen überschreibenden Methoden (3)*. Die Histogramme in der Abbildung 6 veranschaulichen diese Daten. Dabei zeigt die senkrechte Achse an, wieviele Aufrufstellen einer externen Methode vorkommen. Die waagerechte Achse entspricht den unterschiedlichen Methoden. Da die Anzahl der externen Methoden zu groß ist, werden ihre Namen nicht aufgeführt. Es gibt ein allgemeines Muster für die am häufigsten aufgerufenen Methoden, z.B. tritt die Methode *java.lang.StringBuffer.toString* am häufigsten auf.

Jede Aufrufstelle einer externen Methode führt bei der Referenzziel-Analyse dazu, dass ein Funktional als *Modell* für diese Methode erzeugt wird. Je mehr Aufrufstellen externer Methoden eine Bibliothek oder ein Programm enthält, desto breiter ist die zugehörige Schnittstelle. Die Anzahl der *Modelle* beeinflusst direkt die Komplexität des späteren Zusammensetzens der Ergebnisse. Je größer die Zahl, desto mehr Aufwand ist für die Komposition erforderlich. An den Histogrammen der analysierten Programme erkennt man, dass für einige wenige externe Methoden sehr viele verschiedene Aufrufstellen existieren ($\geq$ 20 Aufrufstellen). Sie stammen alle aus der Standard-

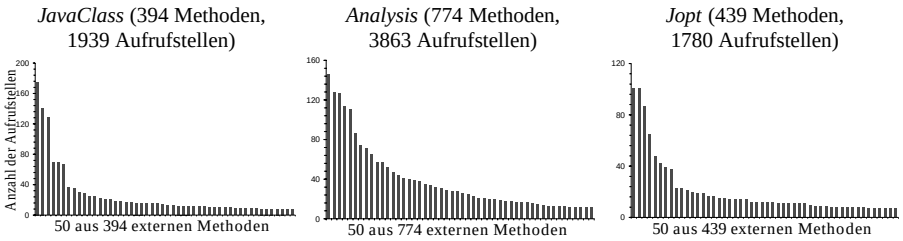


Abbildung 6. Häufigkeitshistogramme der Aufrufstellen externer Methoden

Bibliothek. Aus den Meßwerten ergibt sich, dass die Benutzungsschnittstelle zwischen einem analysierten Anwendungsprogramm und einer einzelnen analysierten Bibliothek in Wirklichkeit viel enger ist. Zum Beispiel zeichnet die *JavaClass*-Bibliothek für nur 118 aus der *Analysis*-Bibliothek aufgerufene, externe Methoden verantwortlich. Das entspricht circa 15% der 774 aus dieser Bibliothek aufgerufenen externen Methoden. Durch die gezeigten Daten wurde ein Einblick in die Breite der offenen Schnittstellen gegeben. Um aber genauere Informationen zu erhalten, muss man die Referenzziel-Ergebnisse betrachten. Interessant für die offene Schnittstelle ist die Anzahl der *Modelle*, die in den Ergebnissen für eine isolierte Bibliothek auftreten. Nur sie werden bei der Komposition durch konkrete Informationen substituiert. Ihre Anzahl spiegelt als Maß den Aufwand für die Komposition recht genau wider.

Die Abbildung 7 veranschaulicht, wieviele *Modelle* in den Ergebnissen für die Bibliotheks-Methoden auftreten. Die senkrechte Achse zeigt, wieviele Methoden eine bestimmte Zahl von *Modellen* (waagerechte Achse) in ihren Referenzziel-Ergebnissen aufweisen. Ein *Modell* kann mehrfach auftreten, wenn mehrere Aufrufstellen für dieselbe Methode mit demselben aktuellen Parameter vorkommen. Betrachtet man die *Modelle* in den Objektmengen genauer, ergibt sich, dass relativ viele *Modelle* tatsächlich mehrfach auftreten. Nach dem Ansatz in [9] könnte die Komplexität des Kompositionsprozesses dadurch verbessert werden, dass die Information für mehrere identische *Modelle* nur einmal eingesetzt würde. Der Aufwand für das Zusammensetzen der fehlenden Informationen über andere Bibliotheken oder Programme reduziert sich dadurch drastisch. Er ist nur noch proportional zur Anzahl der unterschiedlichen *Modelle*. Allgemein gesehen weisen die meisten Methoden ($\geq$ 90%) null bis vier (auch doppelte) *Modelle* auf. Das ist wegen der intuitiv komplex erscheinenden Abhängigkeitsstruktur

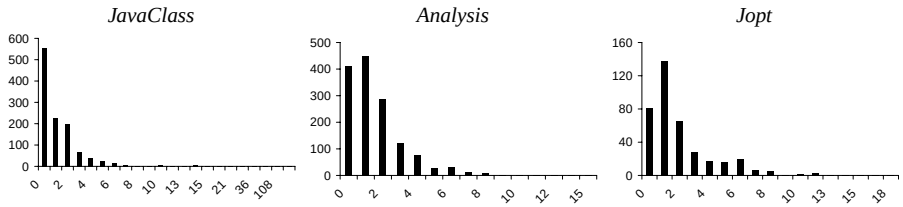


Abbildung 7. Anzahl der *Modelle*

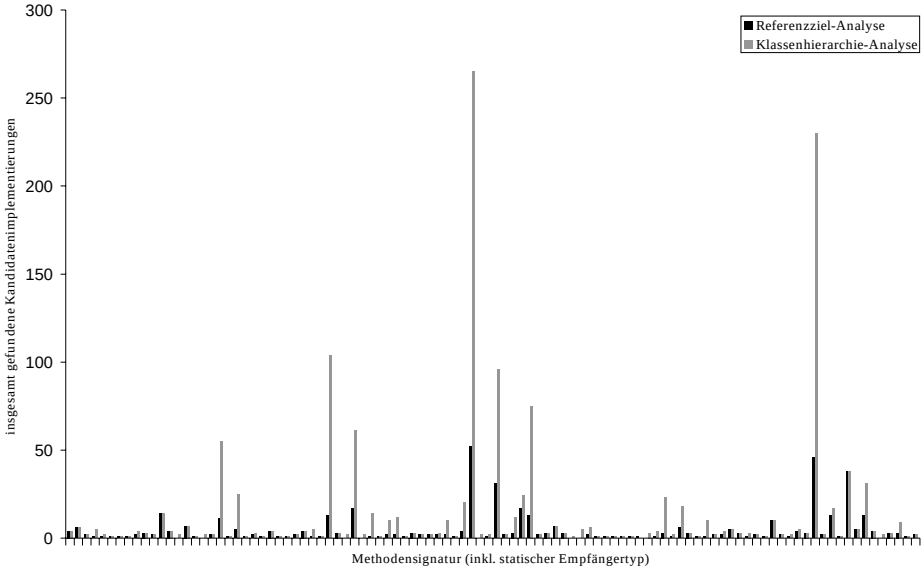


Abbildung 8. Vergleich für dynamische polymorpher Aufrufe zwischen Referenzziel-Analyse und Klassenhierarchie-Analyse

der Referenzziel-Analyse ein vielversprechendes Resultat für die effiziente Komposition der Informationen.

4.2 Vergleich mit den Ergebnissen der Klassenhierarchie-Analyse

Um die Qualität der Referenzziel-Lösung zu prüfen, haben wir die Ergebnisse der Referenzziel-Analyse und der Klassenhierarchie-Analyse bei der statischen Auflösung dynamisch gebundender Aufrufe miteinander verglichen. Die beiden Analysen wurden auf der *Analysis*-Bibliothek durchgeführt. Als Vergleichskriterium haben wir zunächst die von beiden Analysen gelieferten Ergebnisse ausgewählt. Abbildung 8 zeigt diese Ergebnisse für einige repräsentative Methoden aus der Bibliothek. Die senkrechte Achse gibt die Summe aller Aufrufziele über sämtliche Aufrufstellen einer bestimmten dynamisch gebundenen Methode an. Auf der waagerechten Achse sind die Methoden (Methodensignatur inklusiv statischer Empfängertyp) aufgetragen. Jedes Balkenpaar vergleicht die Ergebnisse der beiden Analyseverfahren für eine Methode. Die Klassenhierarchie-Analyse (heller Balken) liefert für jede Aufrufstelle sämtliche Methodenimplementierungen aus der Klassenhierarchie, der Balken entspricht also dem Produkt der beiden Zahlen. Ist der dunklere Balken niedriger, hat die Referenzziel-Analyse die Aufrufziele präziser eingegrenzt.

Als nächstes wird die Differenz in der Genauigkeit zwischen der Referenzziel-Analyse und der Klassenhierarchie-Analyse bewertet. Bei 1960 dynamisch gebundenen Aufrufstellen in der *Analysis*-Bibliothek gibt es 190 Stellen (entspricht 9.6%), bei denen die Referenzziel-Analyse genauere Ergebnisse liefert. Es gibt eine Aufrufstelle, für die

der Genauigkeitsgewinn am größten ist: während die Klassenhierarchie-Analyse alle 15 Implementierungen der Methode für Aufrufziele hält, wählt die Referenzziel-Analyse statisch genau eine aus.

Durch die genauen Informationen über die Menge der Objekte, die eine Variable an einer bestimmten Programmstelle möglicherweise referenziert, können in ähnlicher Weise Lösungen für weitere Analyseaufgaben verfeinert werden.

5 Verwandte Arbeiten

Der Referenzziel-Algorithmus wurde in Anlehnung an [1] und [2] beschrieben. Für die Anwendung auf Bibliotheken wurden darin zwei andersartige Ansätze vorgeschlagen. Die einfachere Möglichkeit ist, dass die externen (dann nicht mehr unbekannten) Methoden auch mitanalysiert werden müssen und mit ihnen auch die Bibliothek, die sie enthält. Bei dem zweiten Ansatz erhält jedes Datenflußelement zu Variablen oder Instanzvariablen, die durch den Aufruf einer unbekannten Methode von dieser modifiziert werden können, zusätzlich noch Werte *TUobjs (typed unknown objects)* als mögliche Referenzziele. Ein *TUobj* repräsentiert alle Objekte eines Typs. Genaueres über Auswirkungen auf Genauigkeit der Analyseergebnisse wurde aber in [1] noch nicht weiter untersucht. Der Unterschied zwischen unserem *Modell* und dem *TUobj* liegt daran, dass ein *TUobj* unabhängig vom Aufrufkontext erzeugt wird und nichts weiter als die Typinformation enthält. Ein *Modell* besitzt dagegen genauere Informationen und ist vom Aufrufkontext abhängig.

Whaley und Rinard [10] haben ein kombiniertes *Pointer* und *Escape* Analyseverfahren entwickelt, das auch auf Java-Bibliotheken angewandt werden kann. Die Analyse basiert auf einer Abstraktion, sog. *points-to escape*-Graphen, wobei die Knoten die Objekte und die Kanten die Referenzen zwischen denen darstellen. Dadurch werden die Interaktionen zwischen analysierten und unanalysierten Regionen des Programms repräsentiert. Unser Verfahren für Java-Bibliotheken wurde unabhängig davon entwickelt. Trotzdem haben beide Ansätze viele analoge Eigenschaften. Wir haben aber mehr auf die Beziehung zwischen Bibliotheken untersucht und nutzen die Analyseinformation für das Zusammensetzen der Ergebnisse später.

Durch die genaue Informationen über die Menge der Programmobjekte, die eine Variable an einer bestimmten Programmstelle referenziert, können genauere Lösung für viele Analysen erziel werden. Ein Vergleich mit der *Alias*-Analyse wurde ebenfalls in [7] präsentiert.

Die Objektgraph-Analyse [8] ist eine statische Analyse, die auf der Ebene einzelner Objekte, nicht der Typen dieser Objekte arbeitet. Im Unterschied dazu fasst die Referenzziel-Analyse mehrere an einer Stelle erzeugten Objekte zu einer Äquivalenzklasse zusammen.

6 Zusammenfassung

Es wurde gezeigt, dass der Einsatz der Referenzziel-Analyse auf unvollständige Java-Programme wie Java-Bibliotheken sinnvoll und praktikabel ist. Durch einen neuen Ansatz läßt sich das Referenzziel-Verfahren statisch auf einzelne Java-Bibliotheken oder -

Programme erfolgreich anwenden. Da die Analyse für Java-Bibliotheken auf Bytecode-Ebene geschieht, kann kommerzielle Software analysiert und damit für eine Optimierung vorbereitet werden. Die gewonnenen Referenzziel-Ergebnisse stehen daher nicht nur für eine spezielle Analyse, sondern für viele Analysen-Aufgaben zur Verfügung. Sie enthalten u.a. die Lösung für Typinferenz, die für objekt-orientierte Optimierungen nützlich ist. Durch die Ergebnisse kann auch die Frage beantwortet werden, ob zwei Variablen potentielle Aliase voneinander sind. Außerdem kann man mit den gewonnenen Informationen *Thread*-lokale Objekte identifizieren.

Literatur

[1] W. A.Landi, B. G.Ryder, and R. Chatterjee. Relevant Context Inference. Technical Report DCS-TR-360, Rutgers University & Siemens Corporate Research Inc, 1997.

[2] R. Chatterjee and B. Ryder. Modular Concrete Type-inference for Statically Typed Object-oriented Programming Languages. Technical Report DCS-TR-349, Rutgers University, 1997.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1992.

[4] M. Dahm. Byte Code Enigneering with the JavaClass API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, May 1999.

[5] A. Diwan, J. E. B. Moss, and K. S. Mckinley. Simple and Effective Analysis of Statically-typed Object-oriented Programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[6] W. A. Landi and B. G. Ryder. Pointer-induced Aliasing: A Problem Classification. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Priciples of Programming Languages*, 1991.

[7] D. K. Le. Referenzziel-Analyse für Java-Bibliotheken und Wiederverwendung ihrer Ergebnisse. Master's thesis, Universität Paderborn, 2000.

[8] A. Spiegel. Object Graph Analysis. Technical Report B-99-11, FU Berlin, FB Mathematik und Informatik, July 1999.

[9] M. Thies. A Closer Look at Inter-library Dependencies in Java-software. In *Tagungsband JIT '99*, Seite 48–57, 1999.

[10] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, October 1999.