Algorithmik- und logikbasierte Points-to Analysen im Vergleich
Eine ausführliche Diskussion der Vor- und Nachteiel beider Ansätze anhand eines gemeinsamen Frameworks

Markus Herhoffer

12. Dezember 2012

Zusammenfassung

Die vorliegende Diplomarbeit implementiert die logikbasierte Points-to-Analyse DOOP in das Analyseframework WALA. DOOP wird zudem um eine Exception-Analyse erweitert, um Stil und Skalierbarkeit von DOOP praktisch zu untersuchen. Anschließend wird DOOP in Präzision und Performance den vorhandenen algorithmischen Points-to-Analysen in WALA gegenübergestellt.

Es zeigt sich, dass die Implementierung einer Points-to-Analyse in einer logischen Sprache präzisere Ergebnisse liefert. Die Performance übertrifft etablierte Alternativen um ein Vielfaches, was jedoch nicht auf die Points-to-Analyse, sondern auf über die Jahre gereifte und optimierte Laufzeitumgebungen zurückzuführen ist.

Kapitel 1

Motivation

Eine Points-to-Analyse ist neben der Konstruktion von Callgraphen die Hauptdisziplin der statischen Programmanalyse. Eine Points-to-Analyse berechnet, auf welche Objekte im Speicher eine Programmvariable zur Laufzeit zeigen kann.

Die Programmiersprache Java ist objektorientiert, statisch typisiert und sie bindet dynamisch. Dabei ergibt sich eine wechselseitige Abhängigkeit zur Callgraph-Berechnung: Für eine Zuweisung `x = y.f()` ist die Information über `f()` aus dem Callgraphen nötig. Diese Information wiederum muss die Informationen aus der Points-to-Analyse kennen, auf welche Speicherobjekte die Variable `y` zeigen kann. [**?**].

DOOP [BS09b] ist eine Points-to-Analyse mit integrierter Callgraph-Erzeugung, die im Gegensatz zu der Mehrzahl der verbreiteten Ansätze keinen rein algorithmisch-prozeduralen Ansatz verfolgt, sondern auf einer logischen Programmiersprache aufbaut. In verschiedenen Veröffentlichungen wird sie als besonders präzise beschrieben und soll in der Laufzeit vergleichbare Analysen um ein zweistelliges Vielfaches unterbieten. Eine Übersicht über die Architektur gibt 1.1, links.

WALA ist ein Framework zur Programmanalyse in Java. Die dort eingebaute Points-to-Analyse verfolgt einen algorithmischen Ansatz und ist komplett in Java geschrieben. Die Laufzeit dieser Analyse ist im Vergleich zu SOOT sehr hoch, die Präzision ist bis dato noch nicht untersucht und verglichen worden. Ein sehr grobes Komponenten-Diagramm dieser Architektur zeigt Abb. 1.1, rechts.

Die vorliegende Arbeit portiert die DOOP-Analyse in das WALA-Framework. Dadurch soll das WALA-Framework Nutzen aus der schnellen und präzisen DOOP-Analyse ziehen. Außerdem ist damit zum ersten Mal möglich, die beiden Ansätze der algorithmische und logischen Points-to-Analyse miteinander zu vergleichen. Zudem wird DOOP um eine einfache Exception-Analyse erweitert, um die Skalierbarkeit und Erweiterbarkeit von DOOP zu evaluieren. Abb.

Abbildung 1.1: Der vorhandene status quo: Das logik-basierte SOOT ist im Framework DOOP implementiert, Wala bietet eine eigene algorithmische Points-to Analyse

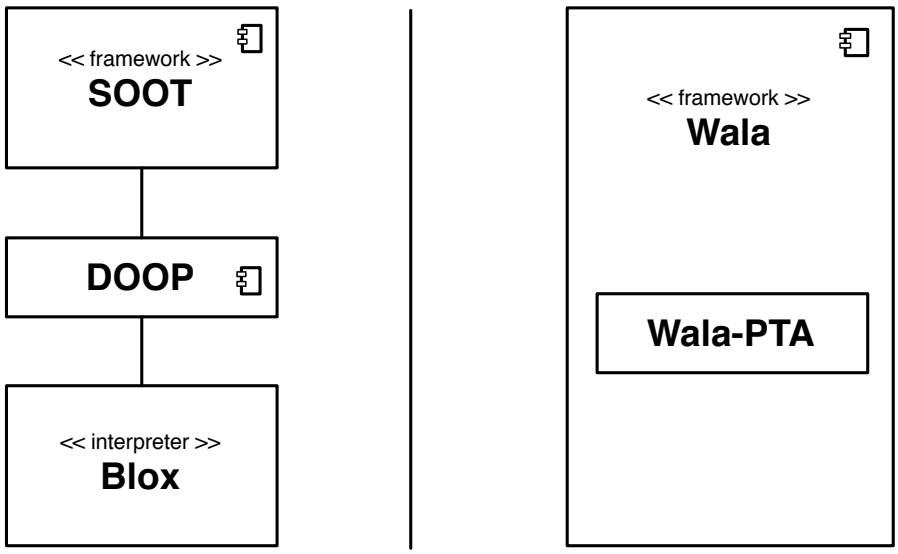
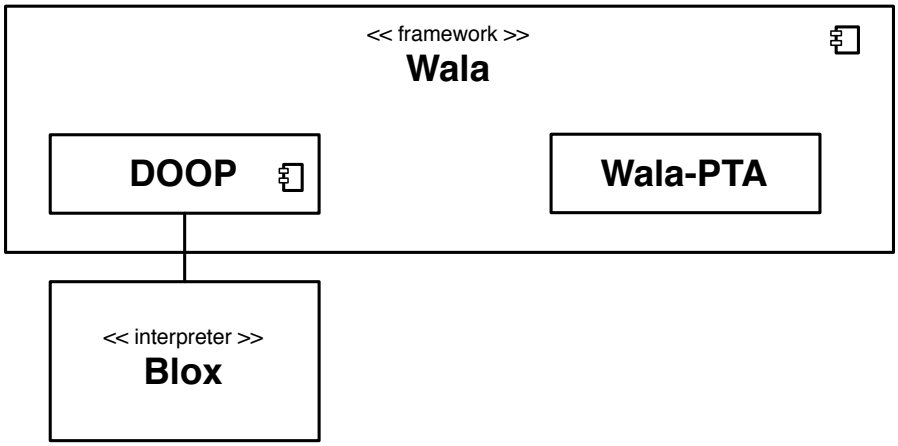


Abbildung 1.2: Die Interation von DOOP in WALA in einer schematischen Übersicht. Eine genauere Darstellung der kompletten Architektur bietet Abb. 4.1



Kapitel 2

Theoretische Grundlagen

2.1 Points-to Analyse

Eine Points-to-Analyse ist eine statische Programmanalyse, die für jeden Zeiger eines Programms berechnet, auf welches Objekt im Heap er zeigen kann. Diese Heapobjekte sind typischerweise in Java mit `new` erzeugt [Str00]. Zeiger sind typischerweise Variablen im Programmcode, insbesondere aber auch Variablen in der SSA-Zwischensprache. Objekte im Heap sind in Java Objekt-Instanzen einer Klasse, insbesondere auch erzeugte Exceptions.

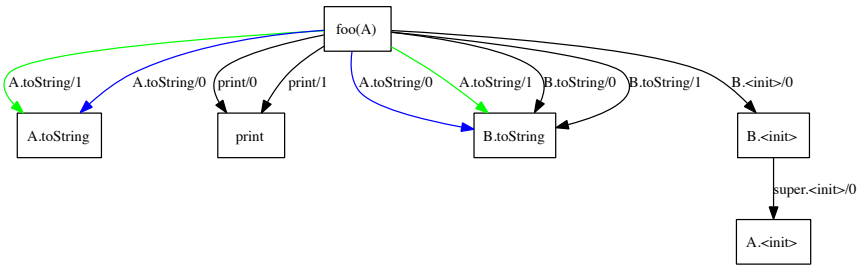
Die *Kontext-Sensitivität* einer Points-to Analyse ist wesentliche Voraussetzung für ihre Präzision. Kontext-sensitive Points-to Analysen beziehen eine Abstraktion des Kontextes (bezeichnet mit Δ) mit ein. Dieser Kontext ist ein statisches Abbild des dynamischen Kontextes einer Methode. Typische Kontexte sind auf die Aufrufstelle der Methode bezogen und/oder auf das Empfängerobjekt. Der Einbezug der Aufrufstelle wird im Folgenden als *call-site sensitive* bezeichnet. Wenn das Empfängerobjekt einbezogen ist, ist von *object sensitive* die Rede. Beide Typen können noch durch eine vorangestellte Zahl, die die Tiefe repräsentiert, näher definiert werden. Eine Analyse, die sowohl die Aufrufstelle als auch das Empfängerobjekt nicht berücksichtigt, wird *insensitive* oder *context insensitive* genannt [BS09b]. Eine genaue Definition und Erklärung zur Kontextsensitivität wird in Abschnitt 2.3 erfolgen.

2.2 Callgraph

Eine Points-to Analyse steht in wechselseitiger Abhängigkeit zu dem Callgraphen (auch *Aufrufgraph*). Ein Callgraph *CG* ist ein gerichteter Graph mit den Callsites (Aufrufstellen) als Knoten. Wenn eine Aufrufstelle c eine Methode m aufrufen kann, gibt es eine Kante vom Knoten für c zum Knoten für m [ALSU06].

Die vorliegende Arbeit verfeinert die Definition aus [ALSU06]:

Abbildung 2.1: Callgraph zu dem Quelltext aus 1



Für jede Aufrufstelle c gibt es genau eine Kante vom Knoten für c zum Knoten für m, der mit dem Namen p/m bezeichnet wird. Die Zahl n ist ein Index, der bei mehrmaligem Aufruf der Methode m von 0 beginnend hochgezählt wird. Eine Callsite und eine Kante wird mit der Bezeichnung $c/p/m$ eindeutig definiert.

2.2.1 Beispiel

Quelltext 1 Programm für Callgraph 2.1

```
1 class A {String toString() { return "A"}}
2 class A extends A {String toString() { return "B"}}
3 void foo(A a) {
4   B b = new B();
5   print(a.toString());
6   print(b.toString());
7 }
```

Der Quelltext aus 1 mit dem Aufruf in Quelltext 2 resultiert in den erweiterten Callgraphen aus Abb. 2.1. Jede Kante ist genau einer Callsite zuzuordnen. Gleichfarbige Kanten sind identische Callsites, die aufgrund der dynamischen Bindung sowohl zur Implementierung von *toString* in der Klasse A als auch zu derer in B führen können.

2.2.2 Wechselseitige Abhängigkeit von Callgraph und Points-to Analyse

Das Beispiel 1 zeigt, dass es eine wechselseitige Abhängigkeit zwischen den Ergebnissen der Points-to-Analyse und der Generierung des Callgraphen gibt. Wenn über den Typ des Parameters A a Informationen aus der Points-

to-Analyse vorhanden sind, können Kanten im Callgraphen u.U. ausgeschlossen werden, wenn der einzige Aufruf von *foo* lautet:

Quelltext 2 Einzige Callsite mit Aufruf von `foo`

```
1 foo(new B);
```

So ergibt sich eine wechselseitige Abhängigkeit von Callgraph-Erzeugung und Points-to-Analyse. Beide Prozesse benötigen die Ergebnisse des anderen. Um den Parameterfluss in der Points-to-Analyse zu verfolgen, sind Callgraphen nötig. Um Informationen über Methodenaufrufe abhängig von der dynamischen Bindung zu erhalten, ist für die Callgraph-Erzeugung die Information aus der Points-to-Analyse nötig. Zu diesem Schluss kommen auch Bravenboer et al. in dem Artikel [**?**], welcher auch die Grundlage für die Points-to-Analyse DOOP bildet.

Die großen Ausmaße dieser wechselseitigen Abhängigkeit lassen sich auch an der Methode *toString* verdeutlichen: Potentiell müsste es eine Callgraph-Kante von *java.lang.StringBuilder.toString* zu jeder Implementierung der Methode *toString* im gesamten Programm (Application-Scope) geben. Nur mit einer Points-to-Analyse können jene Kanten ausgeschlossen werden, die im Programmablauf gar nicht auftreten können.

2.3 Kontexte

Das in Quelltext 1 und dem Callgraph in Abb. 2.1 vorgestellte Beispiel zeigt, dass Methodenaufrufe nicht ohne weiteres linear verfolgt werden können, denn eine Methode kann in einem oder mehreren Kontexten aufgerufen werden. Diese Kontexte haben Einfluss auf die Points-to-Mengen der in der Methode deklarieren Variablen [1], da sich in jedem Kontext Parameter und Empfängerobjekt (das Objekt, auf dem die Methode aufgerufen wird, also `this`) unterscheiden können. Je mehr Kontexte unterschieden werden können, desto präziser ist die Points-to-Analyse.

Für eine Points-to Analyse ist es darum notwendig, die unterschiedlichen Kontexte zu unterscheiden. Ein in der Literatur verbreiteter Ansatz ist das Klonen der Callgraph-Knoten. Hierfür werden die Knoten repliziert, wobei jeder Klon eines Knoten einem seiner Kontexte entspricht. Die Points-to- Analyse arbeitet dann auf dem gesamten Graphen mit allen geklonten Knoten (nach [ALSU06]). So kann eine Points-to-Analyse, die per se keine Kenntnis über Kontexte hat dennoch kontextsensitiv analysieren.

Die Kontexte variieren in zwei Dimensionen. Einmal hinsichtlich den Stellen an denen sie aufgerufen werden (Callsite-Sensitivität) und einmal

[1] genauer: SSA-Variablen, welche in Abschnitt 2.4.2 motiviert und eingeführt werden

hinsichtlich der Stellen, an denen das Objekt erzeugt wurde, auf dem sie aufgerufen werden (Objekt-Sensitivität).

2.3.1 *k*-Callsite-Sensitivität und Aufrufstrings

Die k-Callsite-Sensitivität (in älterer Literatur synonym für die generelle Kontext-Sensitivität gebraucht) erstellt einen Kontext für jede Aufrufstelle einer Methode mit der tiefe k.

Formal ist sie etwa in [NNH99] beschrieben. **Lab** ist die Menge aller Label ℓ, wobei an dieser Stelle nur folgende besondere Label von Relevanz sind:

- ℓ_c, die eine Methodenaufruf bezeichnen
- ℓ_r, die den Returnpunkt einer Methode bezeichnen
- ℓ_n, die den Einstiegspunkt in den Methodenkörper bezeichnen
- ℓ_x, die den Ausstiegspunkt einer Methode bezeichnen

Im vorherigen Beispiel Quelltext 1 lassen sich also folgende Label definieren:

- ℓ_c, die Aufrufe von `toString()` und `print()` in Zeile 5 und 6
- ℓ_r, die expliziten `return`-Anweisungen in Zeile 1 und 2 sowie das implizite Methodenende in Zeile 7
- ℓ_n, die öffnenden geschweiften Klammern der Methoden `toString()` in Zeile 1 und 2 und `foo()` in Zeile 3
- ℓ_x, die schließenden geschweiften Klammern der Methoden `toString()` in Zeile 1 und 2 und `foo()` in Zeile 7

Ein Methodenaufruf ist also ein Pfad der Form $(\ell_c; \ell_n)$ und seine Rückkehr bei ℓ_r oder ℓ_x).

Der Kontext Δ ist also eine Menge

$$\Delta = \mathbf{Lab}^*$$

.

Technisch sind potentiell unendliche oder sehr große Mengen nicht verarbeitbar. Darum wird im Folgenden die Anzahl der Elemente in Δ auf k limitiert. Demnach:

$$\Delta = \mathbf{Lab}^{\leq k}$$

wobei das jüngste Label ℓ_c eines Methodenaufrufes jeweils rechts steht. So wird in Δ also der mögliche Aufrufpfad, der zu einer Methode führen kann,

definiert. Die Aufrufpfade nennt man auch *callstrings* oder *Aufrufstrings*. Technisch betrachtet ist ein Callstring ein Abbild des Callstacks.

Δ bezieht sich nach dieser klassischen Definition, welche die objektorientierte Programmiersprachen gar nicht umfasst, nur auf Callsites. Darum wird diese Kontextmenge der Callsite-Sensitivität im Folgenden mit Δ_{cs} bezeichnet.

2.3.2 *k*-Objekt-Sensitivität

Die Callsite-Sensitivität genügt nicht, um einen präzisen Kontext für objektorientierte Programmiersprachen zu definieren. Das Verhalten, die Eigenschaften und die Ausgabe eine Methode unterscheidet sich nicht nur nach der Aufrufstelle der Methode, sondern auch nach dem Objekt, auf dem die Methode aufgerufen wird. Dies ist insbesondere für die korrekte Analyse der Points-to-Mengen von Feldvariablen wichtig. Für einen Aufruf `myObject.setValue(a)` ist nicht nur die Callsite relevant, sondern auch auf welchem Heap-Objekt `myObjekt` die Methode `setValue()` aufgerufen wird. Nur so kann die Points-to-Menge seines Feldes `value` präzise berechnet werden.

Bei der *Objekt-Sensitivität* wird jede dynamisch gebundene Methode separat für jedes Objekt, auf der sie aufgerufen wird, analysiert. Die Objekt-Sensitivität wurde zuerst in [MRR02] motiviert und definiert.

Analog zum Kontext der Callsite-Sensitivität definieren wir die Kontextmenge der Objekt-Sensitivität ebenfalls als Menge

$$\Delta_o = \mathbf{Lab}^{\leq k}$$

wobei wir dieses Mal keine Callstrings aus ℓ_c verwenden, sondern Label ℓ_{new}. ℓ_{new} bezeichnen Stellen im Programm, an denen mit `new` ein neues Objekt auf dem Heap erzeugt wird. Sie identifizieren also ein Heap-Objekt über die Stelle, an dem es erzeugt wurde.

2.3.3 Kombination der Kontexte und Beispiel

Nach der Definition von Callsite-Sensitivität und Objekt-Sensitivität werden diese beiden Kontexte nun vereinheitlicht. Der neue Kontext ist also eine Kombination beider Kontexte:

$$\Delta = \Delta_{cs} \times \Delta_o$$

Bei einer k-Objekt-Sensitivität ist eine Callsite-Sensitivität implizit enthalten, da zusammen mit dem Empfänger-Objekt auch die Methode festgehalten wird, entweder explizit über den Kontext als Signatur (beliebige k für k-Objekt-Sensitivität) oder implizit über die gerade analysierte Methode (*1*-Objekt-Sensitivität).

Beispiel

Quelltext 3 Java-Beispiel für unterschiedliche Kontexte, nach [SBL11] erweitert

```
1  class A {
2    void foo(Object o) {..}
3  }
4
5  class Main {
6    void main {
7      A a1 = new A();
8      A a2 = new A();
9      ..
10     a1.foo(o1);
11     a2.foo(o2);
12   }
13 }
```

Die Menge der Aufrufkontexte nach Callstrings für die Methode `A.foo` ist $\Delta_{cs} = [10], [11]$ (Zeilennummern aus Quelltext 3 in eckigen Klammern), je ein Callstring pro Callsite. Die Menge der objekt-sensitiven Kontexte ist anders. Er bezieht nur die Stellen als Label mit ein, an denen das Empfänger-Objekt erzeugt wurde. Demnach ist $\Delta_o = [7], [8]$.

Die Kombination beider Kontexte ergibt demnach:

$$\Delta = ([10], [7]), ([10], [8]), ([11], [7]), ([11], [8])$$

Die Points-to Analyse läuft also auf einem Callgraphen, der vier Klone der Methode `A.foo()` enthält, jeweils einen für jeden Kontext.

2.4 SSA-Form

2.4.1 Motivation

Im Umfeld von Programmanalyse und Code-Optimierungen im Compilerbau hat sich eine besondere Form der Programmdarstellung etabliert, die so genannte Static Single Assignment Form. Optimierungen wie Konstantenfaltung und Eliminierung von nicht erreichbarem Code profitieren stark von dieser Form.

Auch Wala und die Wala-eigene alogorithemische Points-to Analyse arbeiten in ihrer Zwischensprache auf einer SSA-Form, wie in Abschnitt 3.1 technisch erklärt wird. Im Folgenden werden nun die theoretischen Grundlagen geschaffen.

2.4.2 Definition

Die SSA-Form (kurz für *static single assignment*) eines Programms ist eine Darstellung, bei der alle Zuweisungen zu einer Variable über eindeutige Namen erfolgen. Dies wird erreicht, indem gegebenenfalls neue Variablennamen eingeführt werden. Folgendes Beispiel zeigt die Anwendung der SSA-Form auf eine Reihe von Zuweisungen [ALSU06]:

Quelltext 4 Ursprünglicher Java-Quelltext

```
1 p = a + b
2 q = p - c
3 p = p * d
4 p = e - p
5 q = p + q
```

Quelltext 5 Quelltext aus 4 in SSA-Form

```
1 p1 = a + b
2 q1 = p1 -c
3 p2 = q1 * d
4 p3 = e - p2
5 q2 = p3 + q1
```

Die ursprüngliche Form aus Quelltext 4 ist in Quelltext 5 in SSA-Form übertragen.

Wenn es mehrere Kontrollflusspfade gibt, in denen eine Variable definiert wird, kommt die so genannte Φ-Funktion zum Einsatz. Die Φ-Funktion kombiniert mindestens zwei Zuweisungen einer Variablen [ALSU06]. In der Praxis kombiniert eine Φ-Funktion immer genau zwei Zuweisungen. Quelltext **??** zeigt den Einsatz der Φ-Funktion am bereits begonnenen Beispiel:

Quelltext 6 Verzweigter Kontrollfluss

```
1 if (flag)
2   x1 = -1;
3 else
4   x2 = 1;
```

Wenn nun eine allgemeine Points-to-Analyse auf einer SSA-Form angewendet werden muss, sind Zuweisungen von Zeigern auf Zeigern bzw. von Variablen auf Variablen der Form `q = p` gar nicht mehr möglich. Die Analyse konzentriert sich also auf den interprozeduralen Kontrollfluss, insbesondere

Quelltext 7 Phi-Funktion zu Quelltext 6

```
1 x3 = Φ(x1,x2);
```

auf die Points-to-Informationen der Funktionsparameter und die Zuweisungen der Φ-Funktionen.

2.5 Formale Definition einer Points-to Analyse

Nach Klärung aller Begriffe kann eine Points-to-Analyse nun wie folgt definiert werden: Eine Points-to-Analyse berechnet für jede Variable $\mathtt{i}_{m,\Delta}$ der Methode m im Kontext Δ eine Points-to-Menge $P(\mathtt{i}_{m,\Delta})$ mit $\ell \in P(\mathtt{i}_{m,\Delta})$ wobei alle ℓ vom Typ ℓ_{new} sind, also Stellen an denen mit `new` neue Objekte erzeugt werden. Die Points-to-Menge umfasst demnach alle Heap-Objekte, auf die die Variable potentiell zeigen kann. Da die Heap-Objekte durch ihre Labels ℓ) identifiziert werden, sind die Elemente in P Äquivalenzklassen. In Codebeispiel 8 werden alle Objekte, die in Zeile 3 erzeugt werden, zu einer Äquivalenzklasse in P zusammengefasst, obwohl es sich um mehrere einzelne Heap-Objekte handelt.

Quelltext 8 Mehrere A-Objekte innerhalb einer Äquivalenzklasse der PTA

```
1 while (\ldots)
2 {
3         A a = new A();
4         foo(a);
5 }
```

Die Menge P ist immer endlich.

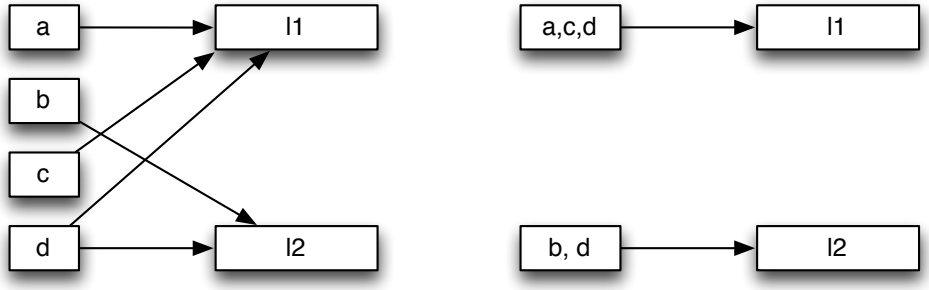
Ein *Points-to Graph* ist ein Graph mit ℓ_{new} und SSA-Variablen $\mathtt{i}_{m,\Delta}$ als Knoten. Es gibt genau dann eine Kante von $\mathtt{i}_{m,\Delta}$ zu ℓ, wenn $\ell \in P(\mathtt{i}_{m,\Delta})$. Ein Beispiel eines PTG ist in Abb. 2.2, links, zu sehen.

2.6 Traditionelle Ansätze: Andersen und Steensgaard

Für die Programmiersprache C gibt es traditionelle Points-to Analysen: Das Verfahren nach Andersen [And94] und das Verfahren nach Steensgaard [Ste96].

Die Sprache C unterscheidet sich jedoch fundamental von Java. C kennt Pointer und Pointer auf Pointer, jedoch keine Objektorientierung. Insbeson-

Abbildung 2.2: Vergleich der Points-To Graphen zu Quelltext 9. Links Anderssen, rechts Steensgaard



dere gibt es in C keine dynamische Bindung. Methodenaufrufe können also immer kontext-insensitiv betrachtet werden.

Obwohl sowohl Andersen als auch Steensgaard darum in ihrer ursprünglichen Form nicht für eine präzise Points-to-Analyse in Frage kommen, sollen beide Verfahren hier kurz erklärt werden.

Der Points-to-Graph des Andersen-Verfahrens [And94] besteht aus Pointern und Objektrepräsentanten. Eine Kante $i \to \ell$ existiert genau dann wenn i auf o zeigen könnte – auch transitiv: $o \in P(i) \Leftrightarrow i \to i' \to i'' \to \ldots \ell$. Der PTG wird aufgebaut, indem bei jeder Zuweisung der Form `i = j` $P(i)$ neu definiert wird:

$$P(i) = P(i) \cup P(j)$$

Der Andersen-PTG definiert also eine Kante für jede Variable im Code zu jedem Objekt, auf das sie zeigen könnte.

Das Steensgaard-Verfahren [Ste96] arbeitet auf einem ungenaueren, aber kleineren PTG als Andersen. Bei einer Zuweisung `i = j;` werden die beiden zugehörigen Knoten verschmolzen.

Quelltext 9 Beispielcode zum Vergleich von verschiedenen PTA

```
1 A a = new A(); //l1
2 A b = new A(); //l2
3 A c = a;
4 A d;
5
6 if (cond)
7   d = a;
8 else
9   d = b;
```

2.7 Points-To Analysen auf BDD

Weit verbreitet sind Points-to-Analysen auf *Binary Decision Diagrams* (BDD) (etwa [LH08]). DOOP verwendet keine BDD, sondern repräsentiert die Points-to-Relationen explizit. Zur Abgrenzung von DOOP zu einem BDD-Ansatz soll dieser kurz umrissen werden.

Traditionell wird auch für einen BDD-Ansatz eine logische Programmiersprache wie Datalog verwendet. Die booleschen Funktionen werden auf besondere Graphen, die BDD, abgebildet. Eine Einführung in die Lösung mittels BBD ist etwa in [ALSU06, Kap. 12.7] zu finden.

Der Nachteil einer Points-to-Analyse, die auf BBDs basiert, ist die Notwendigkeit einer strikten Ordnung für die BDD-Variablen. Aufgrund der exponentiell vielen Kontexten ist es selbst für kleine Java-Programme nicht trivial, eine Ordnung der Variablen herzustellen. [ALSU06] schlägt zur Lösung dieses Problems Ansätze wie aktives Maschinenlernen vor.

BDDs sind demnach für eine präzise Point-to-Analyse mit Objekt- und Callsite-Sensitivät nicht gut geeignet. In der Literatur lässt sich zudem keine Analyse finden, die versucht, jene Präzision mit BDDs zu erreichen.

2.8 Hornklauselprogramm

Die Points-to Analyse DOOP ist in einer logischen Programmiersprache implementiert. DOOP arbeitet demnach nicht wie Java objektorientiert, sondern als *Hornklauselprogramm*.

Die Eigenschaften der Boolschen Algebra und Prädikatenlogik werden an dieser Stelle vorausgesetzt. Eine *Klausel* ist definiert wie in [Sch95, S. 38].

Ein Hornklauselprogramm ist nach [Sch95] ein Programm, welches aus *Tatsachenklauseln* und *Prozedurklauseln* besteht.

Tatsachenklauseln sind einelementige positive Klauseln. Diese Klauseln stellen die Behauptung eines Faktums dar (nach [Sch95]). Im Folgenden werden Tatsachenklauseln schlicht als Fakt oder engl. *fact* bezeichnet. Sie haben die Form

$$Bezeichnung(Parameter_0, \ldots, Parameter_n)$$

Prozedurklauseln haben die Form $\{P, \neg Q_1, \ldots, \neg Qk\}$, wobei $P, Q_1, \ldots, Q_k$ prädikatenlogische Atomformeln sind. Eine alternative und im Folgenden verwendete Darstellung ist

$$P \leftarrow Q_1, Q_2, \ldots, Q_k$$

wobei der Implikationspfeil auch durch die Textzeichen `:-` ersetzt werden kann. P heißt *Prozedurkopf* oder schlicht *Kopf*. $Q_1, \ldots, Q_k$ ist der *Prozedurkörper* oder *Körper*. Tatsachenklauseln können auch als spezielle Proze-

durklauseln ohne Prozedurkörper aufgefasst werden. Prozedurklauseln werden im Folgenden schlicht als `Regeln` bezeichnet.

Ein Prozedurkörper kann insbesondere *stratifizierbar* sein. Die später verwendete Programmiersprache Datalog setzt die Stratifizierbarkeit voraus. Die Menge der Klauseln ist ein mathematischer Körper und Stratifizierbarkeit wird nach strenger Definition auch als Ordnungsrelation auf einem Körper angewandt. Da die Körpereigenschaft aber nicht benötigt wird, wenden wir die Definition direkt auf den speziellen Fall eines Körpers aus Klauseln an. Die Ordnung wird hergestellt, indem jeder Klausel eine natürliche Zahl, die Stratifikationsnummer, zugewiesen wird. Es gilt dann die Ordnung der natürlichen Zahlen.

Ein *Hornklauselprogramm* besteht aus einer endlichen Menge von Tatsachen- und Prozedurklauseln.

Definition 1 (Stratifikation)**.** *Eine Menge von Klauseln der Form*

$$P \leftarrow Q_1 \wedge \ldots \wedge Q_n \wedge \neg Q_{n+1} \wedge \ldots \wedge \neg Q_{n+m}$$

ist genau dann stratifizierbar, wenn es eine Abbildung S von der Menge der Prädikate nach $\mathbb{N}$ gibt, für die gilt:

1. *Ist ein Prädikat P positiv abhängig von einem Prädikat Q, kommt also P im Kopf einer Regel vor, in der Q positiv im Rumpf vorkommt, dann muss P eine größere oder die gleiche Stratifikationsnummer besitzen wie Q, also $S(P) \geq S(Q)$*

2. *Wenn ein Prädikat P von einem negierten Prädikat Q abhängt, also P im Kopf einer Regel vorkommt, in der Q negativ im Rumpf vorkommt, dann muss die Stratifikationsnummer von P echt größer als die von Q sein, also $S(P) > S(Q)$*

Wenn alle Klauseln stratifizierbar sind, dann ist ein Programm eindeutig formal interpretierbar.

Ein Beispiel: Das Datalog-Programm

Quelltext 10 Beispiel für Stratifikation

```
1 p(X) <- q(X);
2 q(X) <- not(r(X)), s(X,Z);
```

ist mit $S = \{p \mapsto 2, q \mapsto 2, r \mapsto 1, s \mapsto 1\}$ stratifizierbar und damit eindeutig formal interpretierbar.

Der Einsatz von Hornformeln in DOOP beziehungsweise generell in der logischen Programmierung hat diverse Vorteile. Zum einen lassen sich mathematische und algorithmische Zusammenhänge der Form „wenn A und B

dann gilt auch C“ sehr einfach und intuitiv in Hornklauseln formulieren, was eine übersichtliche Implementierung zur Folge hat.

Zum anderen sind Hornformeln in polynomialer Zeit auf Erfüllbarkeit testbar, da das HORNSAT-Problem P-vollständig ist.

Kapitel 3

Verwendete Technologien

3.1 WALA und WALA-IR

Die *T. J. Watson Libraries for Analysis*[1], kurz *WALA* sind ein Programmanalyse-Framework implementiert in Java. Ziel der vorliegenden Arbeit ist es, die DOOP Points-to Analyse in WALA zu integrieren und mit den dort vorhanden Analysen zu vergleichen.

WALA arbeitet auf der SSA-Zwischensprache (s. 2.4.2) *IR*, kurz für *Intermediate Representation*. `IR` ist ein Bestandteil von Wala und vielmehr eine Datenstruktur als eine formal spezifizierte Sprache. *IR* repräsentiert die Instruktionen einer Methode auf einem stark reduzierten Befehlssatz und in SSA Form. Statt einer maschinennahen Speicherverwaltung stehen symbolische Register zur Verfügung. *IR* organisiert Instruktionen in einem Kontrollflussgraphen aus Basic-Blocks, wie sie etwa in [ALSU06] definiert sind.

Zudem bietet *IR* Unabhängigkeit von der Quellsprache. So können auch JavaScript-Programm in *IR* dargestellt werden, ein Frontend für Dalvik ist bereits in der Implementierung.

Die verwendeten SSA-Variablen einer Methode werden über einen Index von 0 bis n durchgezählt und sind über diesen Index identifizierbar. Bei einer nicht-statischen Methode ist der This-Zeiger implizit die SSA-Variable mit dem Index 0. Die Parameter der Methode sind die SSA-Variablen mit den Indizes 1 bis p bei einer nicht-statischen Methode bzw. 0 bis $p-1$ bei einer statischen Methode. Dabei ist p die Anzahl der Methodenparameter.

IR an sich trägt keine Information über den Typ einer SSA-Variable. Dieser kann (unter bestimmten Voraussetzungen und in bestimmten Instruktionen) über die Schnittstelle `TypeInference` inferiert werden.

Jede Instruktion benutzt eine Menge von SSA-Variablen, die sie benutzt. Diese Menge ist *Use-Menge* und über `int[] getUse()` abrufbar. Die in der Instruktion definierte, also gesetzte SSA-Variable, ist über `int getDef()`

[1] http://wala.sourceforge.net/wiki/index.php/Main_Page

zugänglich. Nach Konvention kann nur eine Variable pro Instruktion definiert werden. In der Praxis gibt es jedoch Ausnahmen, da auch Exception-Pointer als definierte Variable betrachtet werden. Wie sich *IR* bei Exceptions verhält und wie der Kontrollfluss bei Exceptions modelliert wird, ist in [Her11] und in Abschnitt 3.1.1 ausführlicher beschrieben.

Je nach Typ der Instruktion tragen die definierte SSA-Variable und die Use-Menge eine unterschiedliche Semantik. Insbesondere kann die Use-Menge auch leer oder nur einelementig sein. Auch die definierte SSA-Variable kann leer sein. Die SSA-Instruktion, die zwei Werte vergleicht, hat eine zweielementige Use-Menge, nämlich die beiden zu vergleichende Werte. Eine Throw-Instruktion hat eine leere Use-Menge und definiert nur eine SSA-Variable mit der geworfenen Exception.

Für eine Points-to-Analyse besonders relevant sind die Instruktionen `SSANewInstruction`, `SSAPhiInstruction` sowie die Instruktionen, die einen Methodenaufruf modellieren.

Die `SSANewInstruction` alloziert Speicher und referenziert das erzeugte Objekt in der definierten SSA-Variable der Instruktion. Die `SSAPhiInstruction` ist die einzige mögliche Zuweisung einer Variable innerhalb einer Methode (wegen der SSA-Form sind Zuweisungen von einer lokalen Variablen zu einer anderen nicht möglich). Die beiden Eingangsvariablen sind in der zweielementigen Use-Menge, die Ausgangsvariable ist die definierte SSA-Variable der Instruktion.

Das Aufrufen von Methoden orientiert sich eng an dem Java-Bytecode. Demnach erfolgt ein Methodenaufruf über die `SSAInvokeInstruction`, die wiederum spezialisiert wird zu `InvokeSpecial`, `InvokeStatic` und `InvokeVirtual`.

Bei der Adaption der Point-to-Analyse DOOP nach Wala ist es wichtig, nach den unterschiedlichen Invoke-Varianten Java Bytecode zu unterscheiden:

- mit der Instruktion `invokespecial` werden Konstruktoren (Instanzinitialisierungsmethoden) und mit `private` deklarierte Methoden aufgerufen.
- mit `invokevirtual` werden Instanzmethoden, aber kein Konstruktorern, aufgerufen
- mit `invokestatic` werden statische Methoden (Klassenmethoden) aufgerufen
- Methoden, die in einem Interface deklariert sind, werden mit `invokeinterface` aufgerufen

Besondere Methoden sind die Konstruktoren, die in WALA/IR mit `<init>` bezeichnet werden. Für die Points-to-Analyse von Belang ist, dass ein Konstruktor stets den Konstruktor der Oberklasse ebenfalls aufruft.

Statische Initialisierer sind in WALA/IR mit `<clinit>` bezeichnet. Für eine Points-to-Analyse relevant ist, dass dort statische Felder gesetzt werden können.

Für eine umfassende Dokumentation der IR-Instruktionen ist die offizielle API-Dokumentation zu empfehlen. Letzten Endes sind alle Instruktionen für eine erfolgreiche Points-to-Analyse von Belang, jedoch im Rahmen dieser kurzen Übersicht nicht vollständig dokumentierbar.

3.1.1 Exceptions

Java-Exceptions werden ebenfalls in SSA-Variablen gespeichert und sind demnach auch relevant für eine Points-to-Analyse. In IR wie auch im Java Bytecode sind Exceptions reguläre Heap-Objekte, die wie normale Objekte auch mit `<init>` erzeugt werden. Nach der Erzeugung können sie von der Points-to- Analyse wie ein reguläres Referenzobjekt behandelt werden (eben mit dem Typ der Exception als Referenztyp). Das Erzeugen der Exceptions bedarf in IR einer gesonderten Betrachtung.

Ein im Quelltext explizit vorhandenes "`throw` wird von IR mit einer `SSAThrowInstruction` abgebildet. Das Exception-Objekt ist dann in der von der Instruktion definierten SSA-Variable referenziert und kann von da an in die Points-to-Analyse einbezogen werden.

Exceptions werden aber nicht zwangsläufig mit einer `SSAThrowInstruction` eingeleitet, sondern können (wie etwa eine `NullPointerException`) implizit entstehen. Diese impliziten Exception werden von Wala konservativ behandelt, sie werden also an jeder Stelle implizit erzeugt und geworfen, an der sie potentiell auftreten können. Wala prägt hier den Begriff der *Potentially Excepting Instruction* (PEI). Exceptions dieser Art werden von Wala in einer zweiten von der Instruktion definierten SSA-Variable referenziert. Dies muss bei allen PEI gesondert behandelt werden, um auch diese Exceptions in die Points-to Analyse korrekt einzubeziehen.

3.1.2 Points-to Analysen in WALA

In WALA ist bereits eine Points-to Analyse implementiert, die zusammen mit der Callgraph-Generierung erfolgt.

Wala implementiert mehrere Policies zur Callgraph-Generierung. Die *0-CFA*-Policy ist kontextinsensitiv; es werden nur Objektinstanzen gemäß ihres Typs unterschieden, nicht nach ihrer Allocation-Site. Die *0-CFA*-Policy implementiert nach Abschnitt 2.5 keine Points-To-Analyse, wie wir sie definieren, da die Identifikation nicht über Allocation-Sites ℓ_{new} erfolgt, sondern nur über den Typ.

Die *0-1-CFA*-Policy implementiert eine Points-To-Analyse nach Andersen ([And94] und Abschnitt 9), bei der die Allocation-Sites als Label für

Objekte verwendet werden (ℓ_{new}). Demnach genügt die *0-1-CFA*-Policy der Definition in 2.5.

Das Definieren der Kontexte und deren Tiefe kann über einen ggf. selbst implementierten *Kontext-Selektor* erfolgen. Damit ist es theoretisch möglich, sowohl n-Callsite-Kontextsensitivität als auch n-Objekt-Sensitivität zu generieren, um Vergleichbarkeit mit DOOP (Abschnitt **??**) zu schaffen.

Die Callgraph-Knoten bzw. die zugrunde liegenden Methoden werden für jeden neuen Kontext geklont, der nach dem Kontext-Selektor unterschieden wird. Neue Knoten im geklonten Callgraphen sind also ein Kreuzprodukt aus den alten Knoten und den möglichen Kontexten.

Wala bietet noch einige Optimierungen an. So können zum Performancegewinn die Typen `String` und `StringBuffer` auch in einer *0-1-CFA*-Policy nur über ihre Typen und nicht über ihre Allocation-Sites identifiziert werden. Selbiges ist auch für Exception-Typen möglich. Die *0-1-Container-CFA*-Policy. Sie bietet unlimitierte n-Objekt-Sensitivität für Container-Objekte.

Die DRPA

Mit [SB06] wurde eine *Refinement-Based Context-Sensitive Points-To Analysis für Java* in Wala implementiert, die auf einem vorberechneten Callgraph arbeitet. Der besondere Ansatz der DRPA ist, dass für eine präzise Points-to-Analyse Subpfade im Callgraphen nicht betrachtet werden müssen und demnach in der Untersuchung ausgelassen werden können. Nach diesem Prinzip „verfeinert“ die Analyse nur jene Objekte und Typen, die auch wirklich unterschieden werden müssen. Letzten Endes ist damit also die im Absatz 2.3 vorgestellte Kontext-Tiefe mehr oder minder variabel, die Unterscheidung von Objekten hinsichtlich Ober- oder Unterklasse ist auch nicht präzise, wenn dies im Programmkontext keinen Unterschied macht.

Die DRPA ist demnach nicht mit DOOP vergleichbar, da sie keine feste und klar definierte Präzision bietet, sondern mit dem Einsatz von Ressourcen präzisiert werden kann.

3.1.3 Die Referenz: Paddle

Auch außerhalb von Wala gibt es Implementierungen von Points-to-Analysen.

Für imperative Programmiersprachen gibt es eine Menge etablierter Verfahren (Landi/Ryder, Andersen, Steensgaard und andere), die jedoch nicht ohne Weiteres auf eine objektorientierte Sprache wie Java angewendet werden können. Zudem sind einige Details wie Pointeraritmetik und Pointer auf Pointer nicht vorhanden.

Eine aktuelle Implementierung einer Points-to-Analyse ist das von Lhotak in seiner Dissertation vorgestellte Paddel [LH08]. Sie gilt als besonders präzise und bildet gegenwärtig die Referenz in Präzision und Performance. Paddle ist in der Sprache *Jedd* geschrieben und basiert auf BBD (Bina-

ry Decision Trees, siehe Abschnitt 2.7). Jedd ist eine Spracherweiterung für Java, die das Arbeiten auf BBDs vereinfacht. Als zugrunde liegendes Programmanalyse-Framework verwendet Paddle SOOT.

Ähnlich wie die DRPA zusammen mit Wala ausgeliefert wird, ist Paddle ein Teil von Soot geworden. Hinreichend viele Literatur vergleicht Doop mit Paddle, so ist selbst im grundlegenden Paper von DOOP ([BS09b]) Paddle die Referenz, gegen die evaluiert wird. Auf eine erneute Untersuchung und Evaluation von Paddle wird darum an dieser Stelle verzichtet.

Doop verwendet Eins zu Eins den Algorithmus von Paddle oder vielmehr dessen Regeln. Der syntaxgenaue Algorithmus ist in [LH08][S. 125] beschrieben. Wichtig für die Adapation ist, dass dieser Algorithmus sehr präzise nach *virtual*, *special* und *static* unterscheidet. Demnach muss auch DOOP und die Adaption nach Wala diese Präzision voraussetzen. Da IR sich eng an den Java-Bytecode hält, ist diese Unterscheidung nach den Spezifikationen aus Absatz 3.1 möglich.

3.2 SOOT

Alle Teile von DOOP, die mit Java zu tun haben, sind im Framework SOOT implementiert. SOOT ist, vergleichbar mit Wala, ein Analyseframework für Java. Auch SOOT arbeitet auf einer SSA-Zwischensprache, die aber im Gegensatz zu Wala weit entfernt vom Bytecode liegt und deutlich leichtgewichtiger als IR ist. Die SSA-Zwischensprache von SOOT ist „Shimple“, welches auf „Jimple“ basiert. Jimple kennt keine SSA-Form.

Der Name impliziert, dass die Architektur „einfach“ gehalten ist. Die Nähe zum Bytecode ist nicht immer erkennbar, vielmehr ist die nähe zum Quelltext dominant. Die Abstraktion einer Java-Methode in SOOT hat noch Kenntnis über einzelne Quelltextzeilen. SOOT arbeitet nicht wie IR mit echten Instruktionen, sondern mit „Statements“, die dem Quelltext näher sind als den Bytecode-Instruktionen. So gibt es etwa Statements für `switch` und `break`.

SOOT behandelt Strings gesondert und grundsätzlich als Konstante. Generell werden Typinformationen in SOOT sehr statisch verwaltet. Eine SSA-Variable ist zu jeder Zeit abfragbar, mit welchem Typ sie im Quelltext deklariert wurde. Dies ist in Wala nur über Typinferenz und einen Symboltable möglich.

3.3 DOOP Points-to Analyse

DOOP ist ein Points-to Analyse Framework für Java, geschrieben in der Sprache Datalog, ein Dialekt von Prolog. Der Name ist ein Eigenname und keine Abkürzung. Die Entwickler Martin Braveonboer und Yannis Smaragdakis postulieren in [BS09b], dass DOOP 15 Mal schneller sei als das bisher

als Referenz angesehene Paddle 3.1.3, ohne dabei an Präzision zu verlieren (bei `1-call-site-sensitive`). Zudem soll DOOP in der Präzision skalierbarer sein als alle übrigen vorhandenen Points-to Analysen. Der modulare Aufbau sei zudem übersichtlicher und durch seine deklarative Natur leichter zu beweisen als andere Implementierungen [BS09b].

DOOP profitiert vor allem durch die aggressiven Optimierungen in der Interpretation der Sprache Datalog durch den Interpreter `Datablox`, welcher die tiefen rekursiven Regeln und Fakten der Analyse besonders effizient optimiert und ausführt. Dies unterscheidet DOOP auch von früheren Ansätzen. Der Einsatz von logikbasierten Programmiersprachen zur Programmanalyse war schon länger etabliert und untersucht [BS09b], die Optimierungen und der damit verbundene Gewinn an Performance und Präzision ist jedoch neu. Nicht optimierte Analysen laufen um die Größenordnung 1000 langsamer und sind daher in der Präzision von DOOP in der Praxis nicht ausführbar.

DOOP umfasst sowohl die Generierung des Callgraphen als auch das Behandeln der kompletten Java-Semantik wie statische Initialisierung, Threads usw. Wegen der wechselseitigen Abhängigkeit von Callgraph-Erzeugung und Points-to-Analyse (siehe dazu auch 2.2.2) kann durch das zeitgleiche Ausführen beider Analyse weitere Präzision und Performance gewonnen werden [**?**].

Durch die vollständige deklarative Darstellung des Algorithmus in Form von Datalog-Regeln bietet DOOP eine sehr präzise Spezifikation der Points-to- Analyse, frei von potentiellen Implementierungsfehler und Unübersichtlichkeiten einer Java-Implementierung. Die Logik von Datalog ist ob ihrer direkten Abbildung in die Regeln der Boolschen Algebra selbstdeklarativ und muss nicht erst in einen Java-Algorithmus übertragen werden.

Der Algorithmus, der DOOP zu Grunde liegt, ist keine Neuentwicklung. Er ist die direkte logische Nachimplementierung der Paddle-Analyse ([LH08]). Der Performance-Gewinn von DOOP ist auch immer im Vergleich zu dieser Referenzimplemetierung gemessen. Eine Aussage über die Präzision kann wegen der identischen Implementierung nicht getroffen werden.

DOOP bildet einen Points-to-Graph, der analog dem Andersen-Verfahren (Abschnitt 2.6) ist. Es werden also keine Vereinigung der Points-to-Mengen wie bei Steensgaard 2.6 eingeführt, sondern jede SSA-Variable i im Kontext Δ im Points-to-Graph zeigt explizit auf alle möglichen Heap-Objekte, die sie referenzieren kann.

Eine komplette Dokumentation und Einführung in DOOP ist in den Quellen [BS09b] und [BS09a] tiefgehend behandelt. Um einen groben Überblick über den „Kern“ der Analyse, zeigt Quelltext 11 die Definition des Prädikates `VarPointsTo` aus dem DOOP-Quelltext:

Man sieht, dass DOOP einen ungewöhnlichen Syntax aufweist, der weder dem klassischen Datalog noch Prolog entspricht. Der Syntax und die genaue Funktionsweise wird darum in Abschnitt 3.4 tiefgreifender vorgestellt.

Quelltext 11 `VarPointsTo` aus dem DOOP-Quelltext

```
1 VarPointsTo(?ctx, ?var, ?heap) <-
2   AssignHeapAllocation(?var, ?heap, ?inmethod),
3   CallGraphEdge(_, _, ?ctx,?inmethod).
4
5 VarPointsTo(?toCtx, ?to, ?heap) <-
6   Assign(?fromCtx, ?from, ?toCtx, ?to, ?type),
7   VarPointsTo(?fromCtx, ?from, ?heap),
8   HeapAllocation:Type[?heap] =
9       ?heaptype, AssignCompatible(?type, ?heaptype).
```

DOOP arbeitet auf seiner Laufzeitumgebung Datablox/Datalog nicht auf BDD (siehe Abschnitt 2.7), darum ist eine strikte Ordnung der (BDD-)Variablen nicht nötig.

3.3.1 Optimierungen

DOOP beinhaltet eine separate und demnach optimierte Behandlung für `String`. Da Strings in Java *immutable* sind, werden alle Strings schon bei der Generierung der Fakten als Stringkonstanten aufgefasst. Dies ist nur möglich, weil das umgebende Framework SOOT diese Typinformation bereitstellen kann. Wala kann dies nicht, da *IR* keine Sonderbehandlung von Strings als Singleton vorsieht, sondern Strings als reguläre Objekte betrachtet werden. Darum ist die diese Optimierung in einer Adaption nach Wala nicht einsatzfähig.

DOOP unterstützt Reflections in vollem Umfang. Auch hier wird die Adaption an Grenzen stoßen, da Wala Reflections nur bezüglich der Aufrufe von `Object.clone()` kontextsensitiv betrachtet. Auch hier muss eine Wala-Adaption Einbußen an der Präzision hinnehmen.

3.3.2 Exceptions

DOOP selbst behandelt nur explizit geworfene Exceptions. Zur Laufzeit implizit auftauchende Exceptions (insbesondere `NullPointerExceptions`) werden nicht behandelt. In [BS09a] schreiben die Autoren, dies wäre vielmehr nur für einige Sonderfälle relevant. Vorgeschlagen wird hier, den Typ der Exception selbst als Heap-Objekt zu betrachten und sie dadurch in die Points-To- Analyse mit einzubeziehen. Die Autoren stellen zudem fest, dass das Auslassen der impliziten Exceptions, insbesondere der `NullPointerExceptions` (NPE), die resultierende Points-to Menge „drastisch einschränkt“.

Codebeispiel 12 zeigt nicht nur, dass die Points-to-Menge „drastisch eingeschränkt“ ist, sondern auch unpräzise ist. Insbesondere ist der Kontroll-

fluss, den NPE auslösen können, nicht korrekt im Callgraph abgebildet. Jede potentiell geworfene NPE erzeugt neuen Kontrollfluss, der die Semantik eines Programmes ändern kann. Demnach sind die von DOOP erzeugten Callgraphen nicht nur nicht präzise, sondern auch nicht korrekt. Die darauf aufbauende Points-To Analyse ist folglich nicht präzise und in einigen Sonderfällen nicht korrekt.

Quelltext 12 Beispielprogramm mit Kontrollfluss abhängig von implizit geworfenen Exceptions

```
1  @Test
2  public void testFoo() {
3    String msg;
4    try {
5      foo(mock);
6      msg = "initialized";
7    } catch (Exception e) {
8      msg = "execution␣failed␣with␣" + mock.toString();
9      fail();
10   }
11   log(msg);
12 }
13
14 public static void foo(MyObject o) {
15   o.getValue().initialize();
16 }
```

Quelltext 12 zeigt eine Methode, die typisch für einen JUnit-Testcase ist. Die Methode `public static void foo(Object o)` steht unter Test. Sie wirft keine explizite Exception. Die Methode initialisiert ein Attribut auf dem `MyObject o`.

Wenn das Attribut `mock.getValue() = null` ist, wirft die Methode eine NPE. Diese NPE beeinflusst den Kontrollfluss der Testmethode `testFoo()` und damit auch insbesondere die Points-to-Menge der Variablen `msg`. DOOP erkennt nur explizite Exceptions, demnach wird der Catch-Block im Callgraph nicht erreicht. Die Analyse ist damit in beiden Aspekten, Callgraph und Points-To-Menge, nicht korrekt und nicht präzise.

Die in Kapitel 6 vorgestellte Erweiterung baut auf diesem Defizit auf und versucht, einen Lösungsansatz einzuführen.

3.4 Die Programmiersprache Datalog unter Logicblox

Datalog ist in ihrem Ursprung eine turing-mächtige Datenbankabfragesprache, deren Syntax sich von Prolog ableitet. Da es keine kommerziellen Datenbanken gab und gibt, die diese Sprache implementieren, ist sie bis heute eher von theoretischer beziehungsweise akademischer Bedeutung.

In der folgenden Vorstellung der Besonderheiten von Datalog werden die Syntax von Funktionsweise von Prolog vorausgesetzt und nur auf die Unterscheide zu Prolog eingegangen.

Datalog unterscheidet sich von Prolog in folgenden Punkten:

1. Terme dürfen nicht Argumente von Prädikaten sein. So ist zum Beispiel `P(1,2)` erlaubt, `P(f(1),2)` dagegen nicht möglich.
2. Negation und Rekursion müssen stratifiziert sein 1
3. Die Reihenfolge der Regeln ist frei
4. Es gibt keine Konstruktoren und Funktionen

Insbesondere der letzte Punkt prägt den Charakter der Sprache sehr stark. Da es keine Funktionen gibt, erfolgt die Programmierung rein rekursiv mittels Klauseln, wie sie in Absatz 2.8 eingeführt und diskutiert wurden.

DOOP läuft auf einer proprietären Implementierung von Datalog mit einem eigenen Datalog-Dialekt, der Plattform Logicblox. Wenn im Folgenden nun von Datalog-Syntax die Rede ist, bezieht sich der Syntax stets auf den von Logicblox verwendeten Dialekt.

Wie auch in Prolog hat sich die Unterscheidung zwischen Regeln und Fakten etabliert. Fakten sind einfache Informationen in Prädikatform, wie sie etwa aus einer Datenbank gelesen werden können. Sie sind formal eine Tatsachenklausel, wie in Absatz 2.8 definiert. Abgeschlossen werden sie mit einem Punkt. Quelltext. 13 zeigt ein Beispiel einfacher Fakten.

Quelltext 13 Fakten (Tatsachenklauseln) in Datalog

```
1 Mann(adam).
2 Mann(bob).
3 Frau(claire).
4 Mann(david)
5 Mutter(claire, bob).
6 Vater(bob,adam).
7 Vater(david,claire)
```

Regeln sind Implikationen von Konjunktionen und haben die Form einer Hornklausel (siehe Abschnitt 2.8). Der Kopf steht zu Beginn einer Zeile,

gefolgt von `<-`, einem stilisierten Implikationspfeil, der in beide Richtungen zeigen darf. Er entspricht dem Operator `:-` in Prolog. Die einzelnen Klauseln im Körper werden mit einem Komma getrennt und einem Punkt abgeschlossen.

Die dabei verwendeten Variablen werden in Datalog unter Logicblox mit einem vorangestellten Fragezeichen eingeleitet.

Quelltext 14 Einfache Regeln in Datalog

```
1 Grossmutter(?x,?y) <- Mutter(?x,?z), Vater(?z,?y).
2 Grossmutter(?x,?y) <- Mutter(?x,?z), Mutter(?z,?y).
```

Im vorliegenden Anwendungsfall einer Points-to-Analyse werden eine erhebliche Zahl von Fakten generiert. Darum verwendet Logicblox eine vereinfachte Syntax. Pro Factum-Bezeichner (`Mann`, ...) wird eine eigene Datenbank verwendet. Die Parameter der einzelnen Fakten werden durch das Tab-Zeichen getrennt. In den im Folgenden aufgeführten Quelltexten werden zur Unterscheidung von Leerzeichen und Tab-Zeichen verschieden Lange Unterstriche verwendet. Nach jedem Fakt erfolgt ein Zeilenumbruch. Die Zeilen 6 und 7 aus 13 können also zur Datenbank in Quelltext 15 zusammengefasst werden.

Quelltext 15 Datenbank „Mann“

```
1 bob_____adam
2 david___claire
```

3.4.1 DOOP im Kontext von Datalog/Datablox

In Datalog lässt sich nun ein Regelsatz definieren, der die Points-to-Analyse implementiert. Dieser Regelsatz ist der Kernbestandteil von DOOP.

Ein vereinfachter Regelsatz wäre etwa:

Quelltext 16 Einfache Regeln in Datalog

```
1 VarPointsTo(?var, ?heap) <-
2         AssignHeapAllocation(?var, ?heap).
3 VarPointsTo(?var, ?heap) <-
4         Assig(?from, ?to), VarPointsTo(?from, ?heap).
```

Der Fakt `VarPointsTo` ist die von der Analyse errechnete Information, auf welche Heap-Objekte die Zeiger zeigen.

Bevor dieser Regelsatz Anwendung findet, müssen in einem vorgelagerten Prozess der *Fact-Generation* die Fakten aus dem zu untersuchenden Programm extrahiert werden. In diesem minimalen Beispiel die Fakten zu `Assign` und `AssignHeapAllocation`.

Das Programm zur Generierung dieser Fakten ist neben der eigentlichen Analyse in Datalog der zweite große Bestandteil von DOOP. Die Fact-Generation ist in Java implementiert und generiert über das Programmanalyseframework SOOT für jede Instruktion die von DOOP benötigten Fakten. Verwendet wird dabei die oben beschriebene dateibasierte Kurzschreibweise.

Quelltext 17 Ausschnitt aus der Fakten-Datenbank „AssignHeapAllocation“

```
1  edu.kit.ipd.wala.pointsTo.sources.Simple.simpleAssignments/new␣edu.kit.ipd.wala.pointsTo.
     sources.A/0␣␣␣␣edu.kit.ipd.wala.pointsTo.sources.Simple.simpleAssignmentsr0␣␣␣␣<edu.kit.ipd.
     wala.pointsTo.sources.Simple:␣void␣simpleAssignments()>
2  edu.kit.ipd.wala.pointsTo.sources.Simple.simpleAssignments/new␣edu.kit.ipd.wala.pointsTo.
     sources.B/0␣␣␣␣edu.kit.ipd.wala.pointsTo.sources.Simple.simpleAssignments/r8␣␣␣<edu.kit.ipd.
     wala.pointsTo.sources.Simple:␣void␣simpleAssignments()>
```

Quelltext 17 zeigt exemplarisch zwei Zeilen aus den Fakten, die in der Fact-Generation geschrieben werden müssen. Zu lesen sind die Daten tabellarisch. Pro Zeile stehen je drei Parameter des Faktum `AssignHeapAllocation`. Das Faktum `AssignHeapAllocation` definiert, wo Heap-Objekte erzeugt werden und welcher SSA-Variable sie zugewiesen werden.

In Absatz 2.2 wurde bereits ein Label für eine Callsite definiert. Genau dieser Label-String kommt auch bei DOOP zum Einsatz. Die Zeilen in Quelltext 17 haben die Form `A/B/i`. `A` ist die Methodensignatur der Aufrufstelle. `B` die Creation-Site des erzeugten Objektes und `i` der Index, um auch bei nicht eindeutigen Creation-Sites Eindeutigkeit sicherzustellen.

3.4.2 Abfragen der DOOP-Ergebnisse

Der Datalog-Interpreter Blox erzeugt nach der Ausführung eines Datalog-Programmes eine Datenbank, die man ebenfalls mit Datalog-Befehlen abfragen kann. Letzten Endes unterscheiden sich die berechneten Fakten (Ausgabe-Fakten) formal nicht von den Eingabe-Fakten. Auch die Ausgabe-Fakten sind prädikatenlogische Fakten. Diese lassen sich mit Datalog-Befehlen abfragen. Der Syntax einer Datalog-Abfrage ist identisch mit dem von Prolog und kann in jeder belieben Spezifikation von Datalog nachgelesen werde.

Alle Abfragen der Datenbank erfolgen über das Programm `bloxbatch`, welches Teil von Blox ist. Eine einfache Abfrage kann über `bloxbatch -db <pfad> -query <Abfrage-String>` ausgeführt werden. Eine komplette Ausgabe aller Fakten ist über den `-print`-Parameter möglich. So kann etwa das komplette Ergebnis der Points-To-Analyse über folgende Abfrage ausgegeben werden:

```
bloxbatch -db <pfad> -print VarPointsTo
```

Im kontextsensitiven Fall hat das Prädikat `VarPointsTo` die Signatur `VarPointsTo(?heap, ?ctx, ?var)`.[2] `?ctx` bezeichnet den Kontext (siehe auch Absatz 2.3), `?var` benennt die SSA-Variable über die Signatur ihrer Methode und ihres Indizes und `?heap` definiert das Heap-Objekt, auf das die Variable zeigen kann. Das Heap-Objekt wird über das Label definiert, an dem es per `new` erzeugt wurde.

Folgende Zeile in Quelltext 18 ist eine von mehreren Millionen, die das Prädikat `VarPointsTo(?heap, ?ctx, ?var)` bei einer durchschnittlichen Analyse definiert. Die Methodensignaturen sind der besseren Lesbarkeit wegen gekürzt.

Quelltext 18 Beispielhafte Zeile des Prädikates `VarPointsTo(?heap, ?ctx, ?var)`

```
1 A.<init>/new java.lang.String[]/0, foo/new B/0, A.<init>/2
```

Dabei sind die Parameter für `?heap` und `?ctx` jeweils als `HeapAllocationSite` zu lesen. Beide Parameter verwenden die DOOP-Kurzform, ohne komplette Signatur. Sie wird immer dann angeboten, wenn eine Methode oder ein Klassennamen eindeutig im gesamten Scope des Programmes ist.

$$\underbrace{\texttt{A.<init>}}_{Call-Site} / \underbrace{\texttt{new java.lang.String[]}}_{Creation-Site} / \underbrace{\texttt{0}}_{Index}$$

Der letzte Parameter `?var` ist vom Typ `VarRef` und hat die Form

$$\underbrace{\texttt{A.<init>}}_{Methoden-Prefix} / \underbrace{\texttt{2}}_{Index}$$

und repräsentiert eine SSA-Variable.

Anstatt dem Index kann in einem reinen DOOP-Output auch `@param`i oder `@this` stehen. DOOP pflegt hierfür einen Lookup-Table, um Parameter und den This-Zeiger richtig zuordnen zu können.

Um die Ausgabe von DOOP nicht nur syntaktisch zu verstehen, ist ein Blick in den zugrunde liegenden Quelltext 19 sinnvoll.

Zu lesen ist die Ausgabe in Quelltext 18: In der Methode `foo` (Kontext, `?ctx`) zeigt das Feld in der SSA-Variable 2 (`?var`) auf das im Kontruktor von `A` erzeugte `String`-Array.

Die Ausgabe des Prädikats `VarPointsTo` hat bei mittelgroßen Programmen, wie sie auch in der Evaluation dieser Arbeit betrachtet werden, zwischen 5 und 100 Mio. Einträge (Zeilen) und ist zwischen 1 und 20GB groß.

[2] Die Implementierung von DOOP hält sich hier nicht an die Dokumentation. Die Dokumentation definiert die Signatur mit `VarPointsTo(?ctx, ?var, ?heap)`.

Quelltext 19 Java-Programm zu dem Analysergebnis von 18

```
1  class A {
2    public String[] stringarray = {"0", "1", "2"};
3  }
4  class Main {
5    ..
6    public static B foo(A a){
7      B b = new B();
8      for (int i = 0; i > a.stringarray.length; i++){
9        b.stringarray[i] = a.stringarray[i];
10     }
11     return b;
12   }
13 }
```

Das komplette Protokollieren der Prädikate lohnt daher nicht, zu empfehlen ist ein ständiges Neuberechnen aus der nur ca. 500 MB bis 2 GB großen Datenbank.[3]

3.5 Besonderheiten

3.5.1 Points-To Analyse auf einer SSA-Form

Die vorliegende Arbeit integriert die DOOP-Analyse in das Analyse-Framework WALA. Wala arbeitet auf der wala-eigenen SSA-Zwischensprache *IR*, siehe dazu auch Abschnitt 3.1. DOOP selbst ist nicht auf eine SSA-Form limitiert, weshalb einige Einschränkungen und Besonderheiten gelten. Die SSA-Form wurde in 2.4.2 definiert.

Solche Zuweisungen würden in der Fact-Generation von DOOP über das Faktum `AssignLocal(?from, ?to, ?MethodSig)` definiert. In der SSA-Form kommen diese Zuweisungen in expliziter Form jedoch nicht vor.

Da nicht entschieden werden kann, welcher der beiden Parameter einer Φ-Funktion zugewiesen wird, müssen beide Parameter als lokale Zuweisung betrachtet werden. Eine Φ-Funktion, in der Praxis in der Form einer Phi-Instruktion, wird also in zwei lokale Zuweisungen übersetzt.

Quelltext 20 führt das angefangene Beispiel aus Quelltext 6 fort und zeigt, wie eine Phi-Instruktion als Fakt abgebildet wird.

Eine Phi-Instruktion muss also als ein Kombination mehrerer gleichwertiger lokaler Zuweisung behandelt werden, die sich nicht gegenseitig überschreiben

[3] Alle Zahlen beziehen sich im Minimum auf ein Testprogramm von ca. 100 LOC und im Maximum auf eine Anwendung in der Größe von HSQLDB (277122 LOC) oder jEdit 176668 LOC

Quelltext 20 Datenbank zu „AssignLocal“

```
1 x1 x3 <methodSig>
2 x2 x3 <methodSig>
```

oder obsolet machen. Entgegen der Definition einer Φ-Funktion mit genau zwei Eingängen (wie bei [ALSU06]), haben Phi-Instruktionen in Wala beliebig viele Eingänge. Das Übersetzen einer Φ-Funktion mit mehreren Parametern in eine Verkettung von Φ-Funktionen mit nur zwei Parametern ist trivial.

Quelltext 20 ist also verundet zu lesen: Sowohl `x1` als auch `x2` werden `x3` zugewiesen. Dies unterscheidet sich fundamental von der Semantik lokaler Zuweisungen, wenn keine SSA-Form vorliegt. In diesem Fall macht Zeile 2 in Quelltext 20 Zeile 3 hinfällig, da der alte Inhalt (`x1`) durch die neue Zuweisung überschrieben wird und `x3` demnach nicht mehr auf das vorher in `c1` referenzierte Heap-Objekt zeigen kann. Sobald also eine SSA-Form vorliegt (wie im vorliegenden Fall der Integration in Wala), muss die Semantik einer lokalen Zuweisung korrekt behandelt werden.

Diese Zusammenhänge werden bei der Implementierung eine Rolle spielen, da DOOP in seiner ursprünglichen Form nicht auf einer SSA-Form arbeitet.

Kapitel 4

Architektur

Dieses Kapitel beschreibt die Architektur der Integration und Erweiterung von SOOT in das Wala-Framework. Dabei wird sowohl die bereits vorgegebene Architektur von DOOP erklärt als auch die Architektur der Integration und Erweiterung. Details zur Implementierung werden in Kapitel 5 erklärt. Auf ein Auflisten des Quelltextes der Implementierung wird zugunsten einer Erklärung verzichtet. Der implementierte Quelltext besteht inkl. Kommentaren, Testcases, Erweiterung von DOOP und Helfer-Skripte in Ruby und Perl aus mehr als 4000 Codezeilen (LOC).

4.1 Ausgangssituation

DOOP besteht in dem ausgelieferten Paket aus drei relevanten Komponenten:

1. die Analyse als Hornklauselprogramm (s. 2.8) in Datalog/Blox
2. ein Programm zum Generieren der Fakten in Java
3. eine Reihe von Run-Scripten

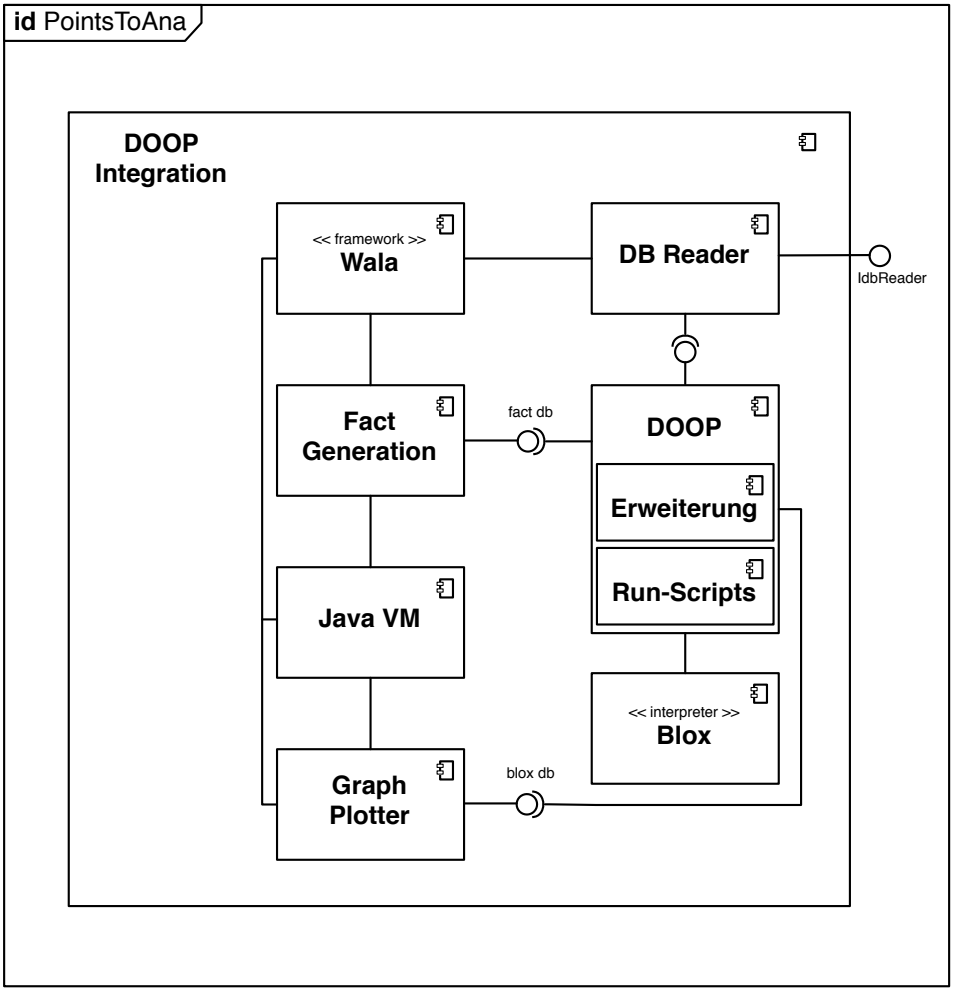
Die Analyse selbst ist ein Satz von Textdateien, der Regeln in Datalog/Blox beinhaltet. Diese erlauben das Ausführen der Analyse mit verschiedenen Kontexten. DOOP bietet verschiedene Arten der Kontextsensitivität mit verschiedenen Tiefen k an.

Das Ergebnis der Analyse ist eine Blox-Datenbank, die mit Datalog-Queries abgefragt werden kann.

4.2 Komponenten

Um DOOP in Wala zu integrieren, ist es primär nötig, die Fact-Generation komplett nach Wala zu portieren und auf die in Wala vorhandenen Datenstrukturen und formale Voraussetzungen (SSA-Form, unterschiedliche

Abbildung 4.1: UML-Komponentendiagramm zur Architektur der Integration von DOOP in Wala.



Zwischensprachen, keine Abstraktion von Strings) anzupassen. Die theoretischen Grundlagen hierfür haben die vorherigen Kapitel behandelt.

Die Run-Scripte müssen so angepasst werden, dass sie mit der neuen Fact-Generation zusammenarbeiten. Die Fact-Generation bietet als Interface eine Fakten-Datenbank in der unter 3.4.1 vorgestellten Kurzschreibweise.

Die Komponenten des gesamten Systems sind in Abb. 4.1 dargestellt. Die Erweiterung von DOOP selbst ist als modulare Komponente innerhalb von DOOP zu implementieren. DOOP selbst hat Blox als Abhängigkeit, dem Datalog-Interpreter, der die Berechnungen ausführt. Die Run-Scripte sind noch von der Shell `bash` abhängig (nicht im Diagramm).

Ein separater `Plotter` erlaubt das Rendern von Points-to-Graphen auf den Ergebnissen von DOOP, die in einer Blox-Datenbank gespeichert werden.

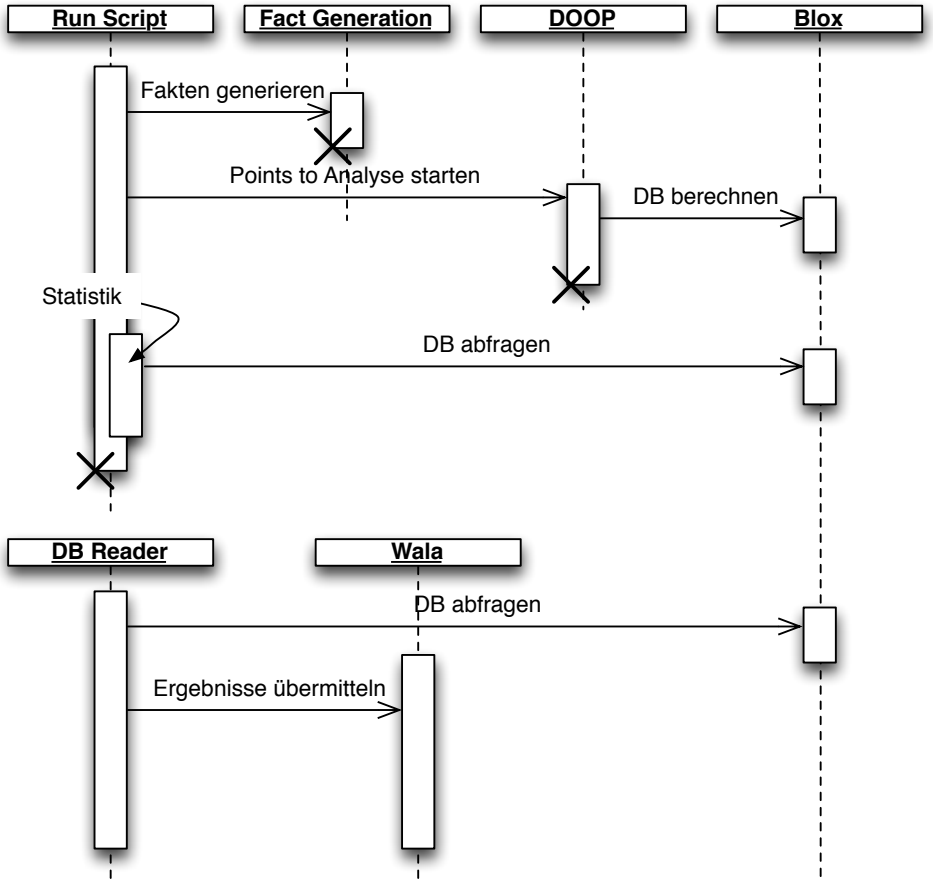
4.3 Ablauf der in Wala integrierten DOOP-Analyse

Abb. 4.2 zeigt ein UML-Ablaufdiagramm der gesamten, in Wala integrierten Analyse. Gestartet wird die Analyse über ein Run-Script, welches dann die neu implementierte Fact-Generation für Wala aufruft. Diese schreibt die Fakten-Datenbank. Das Run-Script übergibt diese Datenbank an DOOP, welches auf Blox eine neue Datenbank mit den Ergebnissen der Points-to Analyse schreibt. Das Run-Script gibt dann noch einige Statistiken zur erfolgten Analyse aus.

Im Nachhinein kann die Datenbank von dem `DB Reader` wieder in Wala zurück importiert werden.

Diese Architektur ist der „standalone“ Betrieb der Analyse. Es ist ohne Weiteres möglich, das Run-Script auch direkt aus einem Java-Programm aufzurufen oder es sogar komplett in Java zu ersetzen. Für den vorliegenden Fall, dass mehrere unterschiedliche Messungen und Benchmarks ausgeführt werden sollen, ist das bereits vorhandene Run-Script mit seiner Vielzahl an Optimierungen zu Caching und einigen Funktionen zum einfachen Aufruf und Konfigurieren der Analyse unverzichtbar.

Abbildung 4.2: UML-Sequenzdiagramm zum Ablauf der Analyse



Kapitel 5

Implementierung

5.1 Fact-Generation

5.1.1 Ausgangszustand und Ansatz

Nicht kompatibel mit Wala ist die Generierung der Fakten-Datenbank, auf der dann DOOP die Points-to Analyse vornimmt. DOOP benötigt alle Informationen zur Semantik von Klassen, Feldern, Instruktionen und deren Beziehungen zueinander.

DOOP verwendet nativ das Programmanalyse-Framework SOOT. SOOT ist im Gegensatz zu WALA sehr leichtgewichtig und strebt nicht an, ein möglichst komplettes und umfangreiches Toolset für eine Vielzahl an Analysen bereitzustellen, wie WALA es tut.

SOOT unterscheidet sich von WALA in einigen Punkten wesentlich: WALA transformiert die Klassen und Methoden vollständig in die Zwischensprache IR, wohingegen SOOT weiterhin auf einem nur schwach abstrahiertem AST arbeitet. Eine SOOT-Methode hat weiterhin einen codezeilenweise vorhandenen *Body*, wohingegen bei WALA bereits eine vollständige Transformation eine die instruktionsbasierte Zwischensprache IR vorliegt. Das Hauptaugenmerk der Integration von DOOP in WALA liegt also im Generieren der Fakten für die eigentliche Analyse in Datalog/Logicblox. Hierfür muss mit WALA ein kompletter Satz von Fakten (im Folgenden Datenbank) generiert werden, der identisch mit den von DOOP generierten Fakten sind. Dabei muss das auf Methoden-Bodies und -Units basierte Konzept von SOOT auf das IR-Konzept von WALA adaptiert werden und im Detail sorgfältig das maschinennähere Sprachkonzept von WALA-IR geparst und verarbeitet werden.

5.1.2 Design der SOOT-Factgeneration

SOOTs eigene Fact-Generation auf Basis von DOOP unterteilt die Generierung der Fakten in folgende Aspekte:

Eine `Main`-Klasse parst die Parameter der Kommandos und bereitet die Klassen auf. Mögliche Eingabe sind sowohl Java-Sourecodedateien als auch kompilierter Java-Code, nativ oder in einem JAR-Archiv.

Die Klasse `FactGenerator` generiert die eigentlichen Fakten rekursiv. Dabei wird über jede Klasse iteriert, dann über jedes Feld und jede Methode und innerhalb jeder Methode über jede Instruktion. So werden alle Member (Klasse/Interface, Methode, Feld, lokale Variable, zuweisende Instruktionen) in Datalog-Fakten abgebildet.

Innerhalb der generierten Fakten müssen die Member eindeutig identifizierbar sein, insbesondere auch, um im Nachhinein die Ergebnisse wieder in das Framework abbilden zu können. SOOT hat hierfür eine eigene Nomenklatur verwendet, die sich sehr eng an den Java-Quelltext orientiert.

Repräsentation der Members

Der `FactGenerator` verwendet Methoden der Klasse `Representation` als Klasse mit Schablonenmethoden, um Darstellungen der Member zu generieren. Dies sind etwa eindeutige Signaturen von Klassen und Methoden und deren Kontext sowie eindeutige Identifikatoren für die Felder. Methoden werden über ihre komplette Signatur identifiziert, etwa

$$< \texttt{java.lang.Object} : \texttt{voidwait(long, int)} >$$

Klassen (und Typen) werden über den kompletten Klassenpfad identifiziert, genestete Klassen werden mit einem Dollarzeichen von der Elternklasse getrennt. Primitive Typen verwenden den selben Namen wie in Java:

$$\texttt{java.io.ObjectOutputStream\$PutField}$$

Felder werden über die sie definierende Klasse und ihren Typ identifiziert. Der Feldname aus dem Quelltext wird beibehalten:

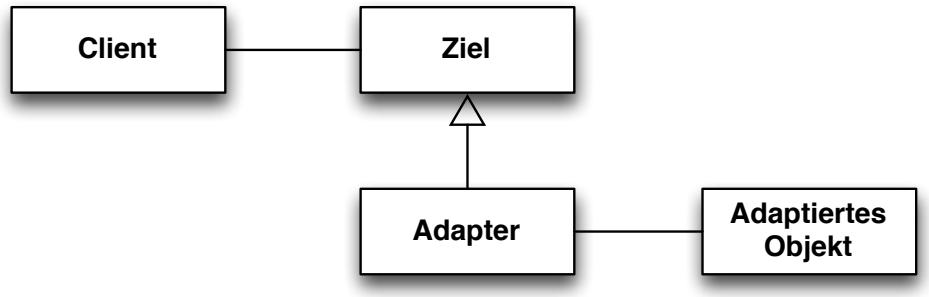
$$< \texttt{java.lang.StringBuilder} : \texttt{longserialVersionUID} >$$

Lokale Variablen an sich gibt es wegen der Darstellung in SSA-Form nicht. Vorhanden ist nur pro Methode eine Menge an SSA-Variablen. Eine Information, welche Quelltext-Variable auf welche SSA-Variable „übersetzt“ werden kann, ist in der Zwischensprache IR nicht mehr vorhanden. Demnach kann eine Identifikation nur noch über Indizes erfolgen. Der Index wird durch einen Querstrich von der Methodensignatur getrennt:

$$\texttt{edu.kit.ipd.wala.pointsTo.sources.A.fooFromIB/2}$$

SSA-Variablen, die der This-Zeiger oder Methodenparameter sind, werden durch vorangestellte Schlüsselwörter gesondert gekennzeichnet.

Abbildung 5.1: UML-Klassendiagramm Entwurfsmuster Adapter



< java.lang.Throwable : void < init > (java.lang.String) > /@this

< java.lang.Throwable : void < init > (java.lang.String) > /@param0

Das Schreiben der eigentlichen Fakten im Datalog-Syntax übernimmt die Klasse `FactWriter`.

5.1.3 Probleme bei der Umsetzung mit dem Adapter-Entwurfsmuster

Erster Ansatz war, die Hauptklassen von DOOP mittels des Entwurfsmusters *Adapter* an WALA anzupassen (siehe hierzu das Design des Entwurfsmusters in Abb. 5.1). Adaptiert werden müssten in diesem Falle die Repräsentationen von einer Java-Klasse, einer Methode, einem Feld usw. Das Entwurfsmuster sieht dabei vor, für jede zu adaptierende Klasse eine Unterklasse zu bilden, die die Methoden und Schnittstellen überschreibt und entsprechend erweitert.

Im vorliegenden Fall müsste also etwa eine `WalaClassAdapter`-Klasse als Adapter für das Ziel `SootClass` abgeleitet werden. Dieser Adapter hat dann als Feldattribut als adaptiertes Objekt die original WALA-Klasse (`IClass`).

Diese klassische Herangehensweise hat sich im Verlauf der Implementierung als nicht erfolgreich erwiesen. Zum einen war die Anzahl der abhängigen und zu adaptierenden Klassen zu groß. Inklusive aller potentiell auftretenden Unterklassen wären mehr als 50 Klassen zu adaptieren gewesen. Zudem ist die konsequente Umsetzung der internen Funktionen von WALA über nicht-gierige Algorithmen allgegenwärtig: So sind etwa die Methoden einer Klasse nicht dauerhaft als `IMethod`-Objekte verfügbar, sondern werden erst bei einer Abfrage erzeugt. Eine Adaption über das Adapter-Entwurfsmuster würde voraussetzen, dass diese Objekte (in einer durschnittlichen Analyse mehrere Hunderttausend) entweder ständig im Arbeitsspeicher geladen sind oder durch einen Cache hätten verfügbar sein müssen.

Weiterer Grund zur Ablehnung des Adapter-Musters ist der fundamentale Unterschied: WALA arbeitet auf einer SSA-Form, SOOT arbeitet auf den Methodenbodies, die noch sehr nahe am AST sind. Die formal sehr auf einen AST aufbauende original `FactGeneration` ist für IR nicht passend. Neben der Adaption muss also eine Vielzahl an Sonderbehandlungen und Anpassungen vorgenommen werden.

Das klassische Entwurfsmuster des Adapters ist für die Implementierung einer Adaption von DOOP nach WALA also nicht zielführend.

5.1.4 Neuimplementierung der Fact-Generation für eine SSA-basiertie Zwischensprache

Darum wurde die Faktengenerierung komplett neu implementiert. Die Architektur der ursprünglichen FactGeneration wurde beibehalten. Die Klassen wurden neu implementiert, jeweils mit dem Prefix `Wala`.

Die Funktionen der Klasse `WalaFactGenerator` hat im Gegensatz zum Original weniger Aufgaben – ausschließlich die Bearbeitung der vier Stufen Klasse/Interface, Methoden, Felder und Instruktionen wird darin behandelt. Dies ist der stärkeren Abstraktion von IR zu verdanken, die nur diese vier Strukturen unterscheidet. Die Bearbeitung der Instruktionen wurden auf eine Subklasse von `SSAInstruction.IVisitor` ausgelagert. Die Bearbeitung lokaler Zuweisungen wurde auf den Sonderfall der in WALA vorliegenden SSA-Form reduziert und angepasst.

Die Klasse `WalaSession` ist wie im Original zur Buchhaltung und zum Zählen gleichnamiger Methoden und Kontexte nötig, arbeitet allerdings nicht mehr auf den SOOT-Units (Teile eines Methodenkörpers), sondern auf den feingrunalareren WALA-Instruktionen.

Um die originalen Bezeichnungen von SOOT zu bekommen, implementiert die Klasse `WalaRepresentation` die Bezeichnungen von SOOT nach und gibt sie als String zurück.

Das komplette Klassendiagramm der adaptierten und an WALA angepassten FactGeneration ist in Abb. 5.2 zu sehen.

5.1.5 Besonderheiten

Beim Implementieren der FactGeneration in WALA müssen viele Details von IR betrachtet werden, exemplarisch werden im Folgenden einige dieser Besonderheiten vorgestellt:

Die Besonderheiten hinsichtlich **SSA-Form** und der Unterscheidung zwischen **Bytecode-Instruktionen und Statements** wurden in der Theorie schon behandelt, technische Implemetierungsdetails hierzu können im Quelltext nachvollzogen werden.

Points-To-Zeiger auf Objekte in Arrays

SOOT ermöglicht durch die enge Nähe zum Java-Code das Betrachten und Identifizieren von Objekten, die in einem Array referenziert werden. Da WALA im Gegensatz zu SOOT auf IR arbeitet, ist die Semantik zum Durchlaufen von Arrays über eine For-Schleife nicht mehr ohne Weiteres nachzuvollziehen. Die original Fact-Generation beachtet diese Semantik über das Prädikat `ArrayIndexPointsTo`, das definiert, welcher Arrayzeiger auf welches Heap-Objekt zeigt. Die vorliegende Implementierung setzt den Array-Index stets auf 0, also auf den Start des Arrays.

Da DOOP Array-Elemente nicht nach unterschiedlichen Typen unterscheiden kann, ist der Verlust an Präzision nur marginal. Insbesondere ist ohne die Implementierung einer Arrayindex-Semantik die Integration in WALA hinsichtlich Arrays komplett identisch und vergleichbar mit der in WALA bereits vorhanden Points-to-Analyse, was in der vergleichenden Evalutation präzisere Ergebnisse bringt.

Sonderbehandlung für Strings

Die in Abschnitt 3.3.1 vorgestellte Sonderbehandlung für Strings ist in WALA nicht ohne Weiteres abzubilden. Nur direkt initialisierte Strings (also konstante Zuweisungen der Form `String s = SString";`) können analog zur original Fact-Generation erfasst werden. Demnach ist die neu implementierte Fact-Generation bei `String` und `StringBuffer` nicht so performant wie das Original. Die Präzision ist identisch, da Strings weiterhin als normale Objekte auf dem Heap betrachtet werden.

Generell bietet es sich an, Strings von einer Points-To-Analyse auszunehmen, was in der vorliegenden Implementierung über einen Filter einfach möglich ist. Strings sind unveränderlich (immutable) und in der Regel nicht in Klassenhierachien eingebunden, was den erheblichen Overhead, den sie in einer Points-To-Analysen verursachen, in den meisten Betrachtungen nicht lohnt.

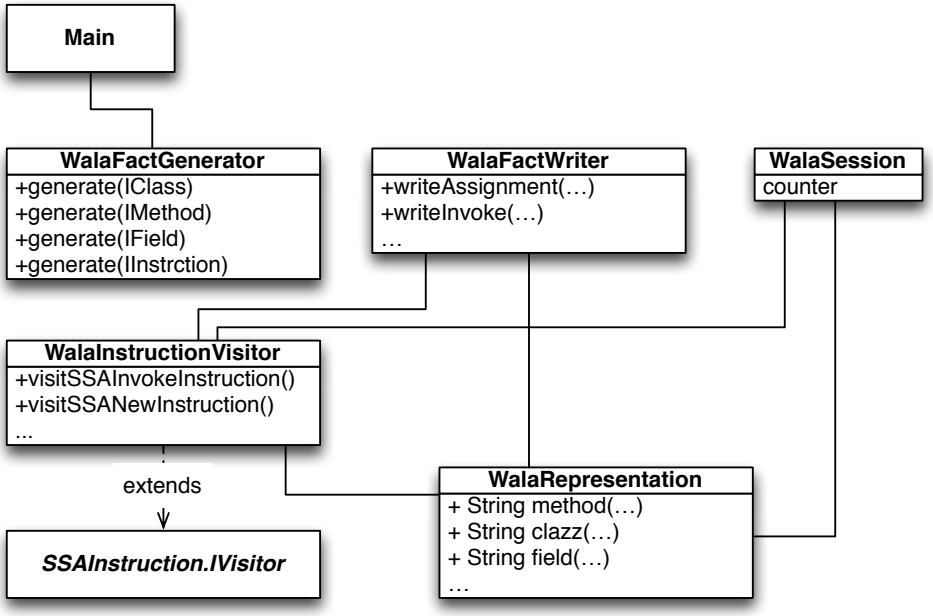
5.2 Blox als Dienst in einer VM-Umgebung

Der Datalog-Interpreter Blox ist ein proprietäres System, welches eigentlich nur als Software-as-a-Service genutzt werden kann. Für akademische Zwecke gibt es jedoch lokal ausführbare Binärdateien. Diese sind für amd64 und GNU/Linux vorkompiliert.

In der vorliegenden Arbeit entstand die Entwicklung und das Ausführen der Fact-Generation unter Mac OS X. Die DOOP-Analyse unter Blox läuft dann in einer virtuellen Maschine unter Ubuntu unter VMware fusion.

Der Austausch der Fact-Datenbanken, sowohl die eingehende als auch die ausgehende, erfolgt über ein gemeinsam gemountetes Dateisystem. Das

Abbildung 5.2: UML-Klassendiagramm der kompletten adaptierten FactGeneratin

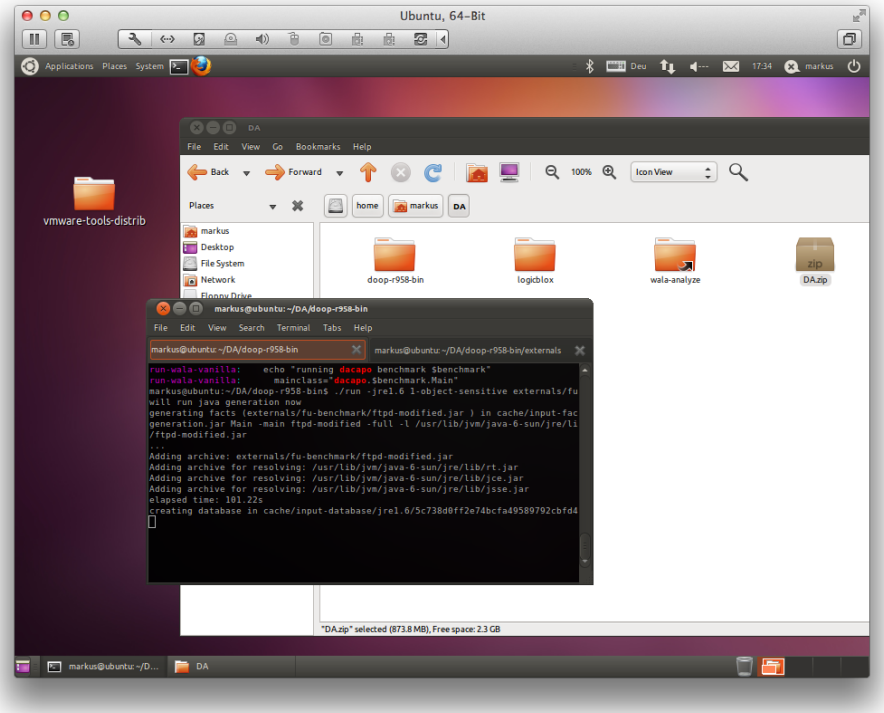


Deployment der beiden Komponenten *DOOP* inklusive Kind-Komponenten und *Blox* aus Abb. 4.1 erfolgt im vorliegenden Fall also einschließlich einer virtuellen Maschine.

Ein erster Ansatz, Blox über nodejs als lokalen SaaS-Dienst zu implementieren wurde verworfen. Die I/O ist ein zeitintensiver Abschnitt der gesamten Analyse. Allein die eingehenden Fakten sind, je nach Größe des Programms, zwischen 270 MB und 2 GB groß. Die Datei-I/O soll bewusst in die anschließende Evaluation mit einbezogen werden, da die Datei-I/O ein elementarer und Bestandteil von SOOT ist, welcher in rein algorithmischen Ansätzen nicht vorhanden ist. Diese Datei-I/O über einen SaaS-Dienst mit Netzwerk-Overhead noch weiter zu erhöhen, erscheint unter dem Gesichtspunkt der Vergleichbarkeit von DOOP mit anderen Points-To Analysen nicht zielführend.

Die Entwickler von Blox planen, Blox selbst als SaaS zur Verfügung zu stellen. Sollte dies einmal verfügbar sein, kann die virtuelle Maschine gegebenenfalls mit dieser SaaS-Komponente ersetzt werden.

Abbildung 5.3: DOOP/Blox in einer VM (Ubuntu 64bit) auf einem OS X Hostsystem



Kapitel 6

Erweiterung

DOOP ist nicht nur eine Points-to-Anaylse, sondern letzten Endes ein ganzes Framework unter dem Paradigma der logischen Programmierung. Um zu untersuchen, in wie weit das Framework erweiterbar ist und wie die Präzision und Performance dieser Erweiterungen ist, soll eine vorhandene WALA-Erweiterung in DOOP portiert werden.

Portiert werden soll eine Optimierung auf Callgraphen, die der vorliegenden Diplomarbeit als Studienarbeit vorausging [Her11]. Dort wurde die Optimierung in ¿3000 LOC Java implementiert. Performance und Arbeitsspeicherverbrauch waren große Herausforderungen. Die Analyse selbst konnte unter Anderem wegen dem hohen Speicherplatzverbrauch und dem hohen Implementierungsaufwand keine Callsite-Sensitivität erreichen.

6.1 Motivation

Kurz zusammengefasst analysiert [Her11], bei welchen Zugriffen auf Referenz-Variablen keine `NullPointerException` (NPE) geworfen werden kann, da das Empfängerobjekt sicher nicht `null` ist. Dies ist besonders hilfreich, um den Kontrollfluss (welcher konservativ betrachtet immer `NullPointerExceptions` mit einbezieht) zu optimieren, indem sicher nicht auftretende Callgraph-Kanten durch NPE eliminiert werden.

6.1.1 Beispiel

Im Beispiel Quelltext 21 kann beim Zugriff auf die Variable `d` in der Methode `needsAParameter(D d)` keine NPE geworfen werden, da in der Methode `invokesMethod` die Variable `d` bereits mit einem Objekt belegt wurde.

Quelltext 21 Beispielcode für einen Prozeduraufruf mit Nullpointer-Exception-Behandlung auf [Her11]

```
1  public class D {
2    int i;
3    public void invokesMethod(){
4      D d = new D();
5      needsAParameter(d);
6      return;
7    }
8
9    private void needsAParameter(D d) {
10     int j = d.i;
11     j++;
12     this.i = 3;
13     return;
14   }
15 }
```

6.2 Transformation des Problems in DOOP/Datalog

[Her11] löst das Problem über einen rekursiven Algorithmus in WALA/Java, der über komplexe Datenstrukturen hinweg einen interprozeduralen Kontrollfluss analysiert und dann nicht auftretende Callgraph-Kanten markiert.

Im DOOP-Callgraphen sind diese Kontrollflusskanten, die von NPE erzeugt werden, gar nicht vorhanden, da DOOP implizite Exceptions nicht behandelt (siehe dazu auch Absatz 3.3.2). DOOP selbst hat noch keine Semantik für Variablen definiert, die nicht auf Heap-Objekte zeigen. So ist insbesondere `null` niemals Element einer Points-to Menge, die von DOOP berechnet wird. Wenn eine Variable i_Δ im Kontext Δ nur auf `null` zeigt (etwa, weil sie nicht initialisiert wurde oder weil sie explizit `null` zugewiesen wurde), dann wird `VarPointsTo()` für i_Δ gar nicht definiert.

6.2.1 Auf `null` zeigende Variablen

Null-Zeiger können durch zwei Operationen entstehen: Einmal das explizite Zuweisen (`a = null`) oder durch ausbleibendes Initialisieren einer Variable `A a;`. Diese beiden Fälle sind kontextlos. Darum ist ein erster Schritt in der Implementierung das Erfassen dieser Fälle. Dazu führen wir das Prädikat `MaybeNull(?var)` ein, das aussagt, dass eine Variable (je nach Kontroll-

fluss) auf `null` zeigen kann.[1] Diese beiden Fälle lassen sich zu folgendem Datalog/DOOP-Code übersetzen (Quelltext 22):

Quelltext 22 Kontextlose `null`-Zuweisungen

```
1 MaybeNull(?var) <-
2   Var:Type(?var,?null_type),
3   NullType(?null_type).
4
5 MaybeNull(?var) <-
6   AssignLocal(?from,?var),
7   MaybeNull(?var).
```

Das Prädikat `Var:Type(?var,?type)` ordnet einer Variable ihren Typ zu, das Prädikat `NullType(?null_type)` hat nur eine Zeile, nämlich `null` als Definition des Nulltyps von Java. Beide Prädikate werden von DOOP definiert.

Aufbauend auf dieser kontextlosen Definition, welche Variable `null` sein kann, wird eine Definition von `MaybeNullContext(?type,?ctx,?var)` vorgenommen, die einen Kontext (siehe Definition unter 2.3) einführt. Der Kontext `?ctx` ist hierbei beliebig; er „erbt“ die Kontexte, die DOOP definiert. `?type` ist schlicht der Typ der Variable, der potentiell `null` sein kann.

Zur kontextbehafteten Definition von `MaybeNullContext` verwenden wir das von DOOP angebotene Prädikat `Assign`. Es kombiniert alle interprodzeduralen Assignements, also alle Zuweisungen aufgrund von Übergabe von Parametern und aufgrund von Returnwerten. `Assign` hat die Signatur

$$\texttt{Assign}(?\texttt{type}, \texttt{Context}(?\texttt{callee}, ?\texttt{formal}), \texttt{Context}(?\texttt{caller}, ?\texttt{actual}))$$

wobei `type` der Typ der Variable ist, der erste Kontext der Kontext der definierten Variable und der zweite Kontext der der zugewiesenen Variablen ist.

Demnach lässt sich `MaybeNullContext` wie folgt definieren (Quelltext 23):

Quelltext 23 Definition von `MaybeNullContext`

```
1 MaybeNullContext(?type,?ctx,?var) <-
2   Assign(?type,?ctx,?var,_,?from_var),
3   MaybeNull(?from_var).
```

[1] „Maybe“ bedeutet in diesem Fall nicht, dass sie „vielleicht möglicherweise“ auf `null` zeigt, sondern dass `null` *sicher* in der Points-To-Menge von `?var` liegt

Der Unterstrich in Zeile 3 von Quelltext 23 bedeutet, dass der Kontext der auf `null` zeigenden Variablen beliebig sein kann. Dies verliert Präzision in den Sonderfällen, in denen der Kontext, in der die Variable `null` sein kann, kein Kontrollfluss in den neuen Kontext möglich ist. Hier ist eine weitere Präzisierung möglich.

6.2.2 Zugriff auf `null`

Auch beim Methodenzugriff betrachten wir zuerst den kontextlosen Fall. Ein Methodenaufruf auf einer Variable, die in jedem Kontext `null` sein kann, ist auch mit Sicherheit ein `NullInvoke`. Demnach definieren wir das Prädikat `NullInvoke(?method,?var)` wie folgt (Quelltext 24):

Quelltext 24 Definition von `NullInvoke(?method,?var)`

```
1 NullInvoke(?method,?var) <-
2   VirtualMethodInvocation:Base(?method,?var),
3   MaybeNull(?var).
```

Da Null-Invokes nur bei Methodenaufrufen auf Objekten erfolgen können, genügt es, nur die mit `invokevirtual`[2] aufgerufenen Methoden mit einzubeziehen.

Für den intraprozeduralen Fall müssen aber die Kontexte miteinbezogen werden. Da das Ziel der Analyse ist, solche potentiellen Null-Invokes im Kontrollflussgraphen zu finden, stößt hier DOOP an seine Grenzen. Ein interprozeduraler Kontrollflussgraph, wie er [Her11] vorgestellt und vorausgesetzt wird, ist nicht möglich. Demnach sind die Tupel aus Kontext und Variable, die `MaybeNullContext` ausgibt, das Ergebnis der Analyse, wie es in [Her11] das Ziel war. Statische bzw. kontextlose Null-Zugriffe, die *immer* eine NPE werfen, werden von `NullInvoke` ausgegeben.

6.3 Bewertung

Die im letzten Abschnitt vorgestellte Erweiterung soll vielmehr als „proof of concept“ dienen, um die Erweiterbarkeit und Flexibilität von DOOP zu untersuchen. Sie hat in einigen Punkten Defizite: So würde es wesentlich mehr Sinn machen, Null-Zeiger nicht über Prädikate einzuführen, sondern `null` als ein „leeres Heap-Objekt“ gleichberechtigt mit den normalen Heap-Objekten in die Points-to Analyse mit einzubauen. Dazu müsste die Semantik der gesamten Analyse an einigen Stellen umgeschrieben werden, da `null` nicht wie normale Heap-Objekte über die Stelle der Erzeugung identifiziert

[2] siehe Java Bytecode Spezifikation, `http://java.sun.com/docs/books/jvms/second_edition/html/Compiling.doc.html#14787`

werden kann. Zudem müssen auch explizite Zuweisungen von `null` generell verarbeitet werden. Diese werden von DOOP in der vorliegenden Version ausgelassen und nicht behandelt.

Wenn auch `null` Element einer Points-to Menge $P(i_\Delta)$ der Variablen i im Kontext Δ sein kann, dann ist die Lösung des NPE-Problems implizit:

$$\texttt{null} \in P(i_\Delta) \Leftrightarrow \texttt{MaybeNullContext}(\mathit{Type}(i), \Delta, i)$$

Demnach ist eine Erweiterung der gesamten DOOP-Analyse, die `null` analysiert und auswertet, eine sinnvolle zukünftige Arbeit.

6.4 Evaluierung

Die in der DOOP-Implementierung vorgestellten logischen Formeln sind nahezu identisch mit den theoretischen Vorüberlegungen zur Java-Implementierung. Eine Implementierung mit DOOP/Datalog hat also den Vorteil, dass ein theoretisches Konzept nahezu direkt und ohne eine fehleranfällige Implementierung in ein funktionierendes Programm übertragen werden kann.

Die im letzten Abschnitt vorgestellte Lösung soll nun mit der algorithmischen Implementierung von [Her11] verglichen werden. Die Kernklasse der Java-Implementierung beinhaltet ca. 800 LOC, dazu kommen noch weitere ca. 2000 LOC Infrastruktur. Dem gegenüber stehen die 15 LOC der Implementierung in DOOP. Von diesem Wert kann bei einer soliden Integration einer `null`-Semantik wie in Absatz 6.3 vorgestellt, nicht ausgegangen werden. Wir schätzen die LOC einer kompletten Implementierung der `null`-Semantik auf 120 LOC. Demnach rangiert der Implementierungsaufwand eines DOOP/Datalog-Programmes im Bereich von 5-10% verglichen mit einer Implementierung in Java (unter der Annahme, dass der Aufwand für eine Zeile Code gleich ist).

Vor dem Bewerten der Ergebnisse muss betrachtet werden, dass die DOOP-Analyse nicht konservativ ist. Eine Erhöhung der Präzision der NPE-Analyse bedeutet also, dass mehr NPE-Kanten gefunden werden können. Die Java-Implementierung entfernt nur NPE-Kanten, wenn sicher kein Zugriff auf `null` möglich ist. Die DOOP-Implementierung definiert nur die Methodenzugriffen, bei denen nachweisbar ist, dass potentiell auf `null` zugegriffen wird.

Auch hier kann erhöhte Präzision durch die Einführung einer `null`-Semantik gewonnen werden. Eventuell ist es möglich, die Regeln von DOOP so neu oder verändert zu definieren, dass die Exceptionanalyse konservativ möglich ist. Die benötigte Datenstrukturen hält DOOP in dem eigenen Callgraphen (prädikatenlogische Formel `CallGraph`) bereits vor. Hier bietet sich weitere Forschung an.

In [Her11] wird die Java-Implementierung u.A. an HSQLDB getestet. Ein Testlauf der DOOP-Implementierung an HSQLDB mit einer 1-object-

sensitive Analyse bringt unglaubliche Ergebnisse: Während die Java-Implementierung (1-callsite-sensitive) rund 8% der NPE-Kontrollflusskanten ausschließen kann, markiert die DOOP Analyse insgesamt 99,99987% aller NPE-Kanten als nicht möglich (157 von 628322 Aufrufen).

Vermutet werden kann, dass bei einer wie oben beschriebenen Erweiterung zur konservativen Behandlung der NPE-Kanten die Zahl der notwendigen NPE-Kanten weiterhin im Promillbereich liegt, wenngleich aber die Ausgangsmenge der überhaupt erkannten NPE-Kanten höher ist.

Diese Zahl erscheint aus mehreren Gründen glaubwürdig: Zum einen ist HSQLDB ein sicherheitskritisches Programm in einer hohen Versionsnummer. Zugriffe auf nicht-initialisierte Variablen sollten allein schon aus softwaretechnischen Gründen sehr wenige vorkommen. Zum anderen beinhaltet DOOP einen Mechanismus, um Zuweisungen (und damit auch Initialisierungen) von Array-Elementen zu untersuchen. Zugriffe auf Array-Elemente können von der Java-Implementierung nicht behandelt werden, wodurch ein beachtlicher Anteil nicht optimierbarer Kanten entsteht.

Eine Evaluierung der Laufzeit ist nicht möglich. Da die Erweiterung zusammen mit der Points-to Analyse ausgewertet wird, ist es nicht möglich nachzuvollziehen, welcher Zeitanteil auf welchen Programmabschnitt fällt. Die Analyse mit Erweiterung dauert im Schnitt ca. 2 Sekunden länger als ohne. Dies als signifikatente Veränderung der Laufzeit zu betrachten ist aber nicht sinnvoll. Demnach ist eine Erweiterung der Analyse um wenige zusätzliche Prädikate extrem kostengünstig.

Kapitel 7

Evaluierung

Die in Wala integrierte DOOP-Analyse soll nun mit der zum Wala-Kern gehörenden Andersen-Analyse (siehe Abschnitt 3.1.2) verglichen werden. Betrachtet werden sollen dabei die Aspekte der Präzision und des Laufzeitverhaltens.

7.1 Betrachtung

Präzision und Performance sind die beiden Größen, in denen DOOP mit der Wala-PTA vergleichbar ist. Die Performance lässt sich relativ einfach über die gemessene Laufzeit der Analyse bestimmen. Die Präzision ist aber nicht ohne Weiteres direkt messbar.

Ein direkter Vergleich der Form $P_{DOOP}(\mathtt{i}_{\Delta}) = P_{Wala}(\mathtt{i}_{\Delta})$ aller Variablen i im Kontext Δ macht nur begrenzt Sinn. Er würde nur die Korrektheit der beiden Analysen wechselseitig verifizieren, was nicht Gegenstand einer Präsisions-Analyse ist.

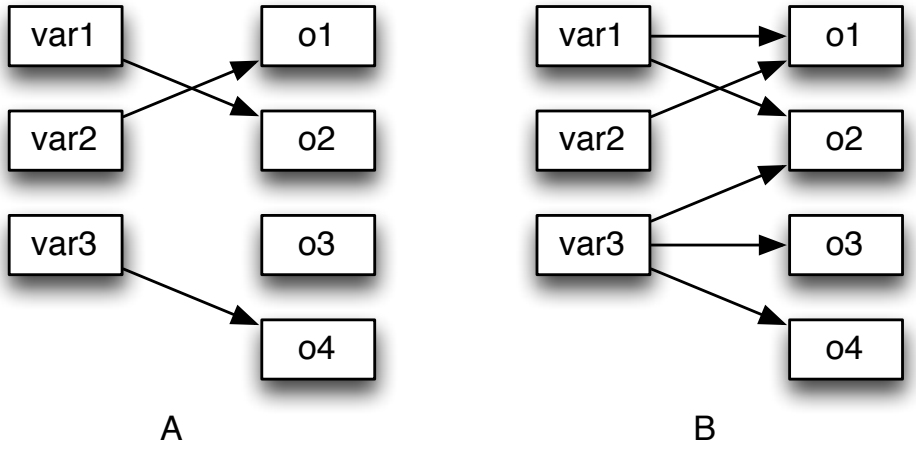
7.1.1 Eine statistische Bewertung der Präzision

Eine Metrik für die Präzision einer Points-To-Analyse ist die Größe Auflösung, also die Frage, wie differenziert die Points-To-Mengen sind.

Die Analyse A ist präziser als die Analyse B, da die Analyse B feiner auflösend ist als A. Das *Aliasing* ist geringer; von Alisasing wird gesprochen, wenn mehrere Variablen im PTG auf dasselbe Heap-Objekt zeigen. Formal bedeutes dies, dass die Anzahl der Elemente für alle Schnittmengen von $P_A(i_{A,\delta})$ und $P_A(j_{A,\delta})$ weniger sind, wenn i und j je zwei verschiedene Variablen sind. Wir definieren die Präzision also:

Die Points-to-Analyse A ist präziser als die Points-to-Analyse B genau dann wenn:

Abbildung 7.1: Vergleich der Präzision zweier Points-to-Graphen



$$\sum_{\forall i_{A,\Delta} \neq j_{A,\Delta}} |P_A(i_{A,\Delta} \cap P_A(j_{A,\Delta})| < \sum_{\forall i_{B,\Delta} \neq j_{B,\Delta}} |P_B(i_{B,\Delta} \cap P_B(j_{B,\Delta})|$$

Die Variablen $i_{P,\Delta}$ und $j_{P,\Delta}$ sind jeweils alle in der PTA vorkommenden paarweise verschiedenen Tupel aus Variable und Kontext.

Abbildung 7.1 zeigt zwei Points-to-Graphen. Gemäß unserer Definition von Präzision ist der Points-to-Graph A präziser als B, weil weniger Variablen auf dasselbe Heap-Objekt zeigen und damit das Aliasing geringer ist. Der Einfachheit wegen wurde auf die Angabe eines Kontextes zu jeder Variable verzichtet.

Eine komplette Untersuchung dieser Eigenschaft ist für alle paarweise verschiedenen Variablen in einer repräsentativ großen Anwendung nicht möglich. Darum werden zur Evaluation eine große Anzahl von Variablen-Paaren per Zufall herausgegriffen und dann überprüft, wie viele Elemente in der Schnittmenge beider Point-To-Mengen sind. Diese Anzahlen werden aufsummiert. Eine Points-To-Analyse ist also bei der selben hinreichend großen Anzahl an zufällig ausgewählten Paaren von Variablen präziser, wenn die Elemente in den Schnittmengen kleiner sind und die Referenzgruppe hinreichen groß ist.

DOOP und Wala behandeln die Objektauflösung und die zur Auflösung mit einbezogenen Klassenbibliotheken unterschiedlich. Der Scope von DOOP ist deutlich größer als der von Wala. Um dennoch eine Vergleichbarkeit der Präzsion zu gewinnen, werden die Variablen-Paare nur aus dem Anwendungs-Scope zufällig ausgewählt. Variablen etwa in `java.lang.*` oder `sun.*` bleiben davon ausgenommen. Die Initialisierungsphase eines Programmes (im Wesentlichen das Aufbauen eines Main-Threads in `java.lang.ThreadGroup`

und `java.lang.ThreadGroup`) wird nicht zum Anwendungs-Scope dazugezählt, da Wala diesen Teil einer Anwendung nicht betrachtet. Verallgemeinernd kann darum der Anwendungs-Scope auf den Namensraum des Programmes (etwa `org.hsqldb.*`) reduziert werden.

Der verwendete Algorithmus zur Messung der Präzision der DOOP-Analyse lautet also:

Quelltext 25 Algorithmus zur Messung der Präsizion einer PTA

Require: ptm = VarPointsTo-Ausgabe von DOOP
  $aliasTreffer \leftarrow 0$
  for $betrachtet = 1 \rightarrow max$ **do**
    $var1 \leftarrow$ ZUFÄLLIGEVARIABLE
    $var2 \leftarrow$ ZUFÄLLIGEVARIABLE
    ASSERT$(var1 \neq var2)$
    $aliasTreffer+ = |P(var1) \cap P(var2)|$
  end for
    return $\frac{aliasTreffer}{betrachtet}$

Eingabe ist die Datenbankausgabe, die die Abfrage des Prädikates `VarPointsTo` ausgbibt. Zufällig werden dann zwei verschiedene Variablen im zugehörigen Kontext ausgewählt. Zur Steigerung der Performance erfolgt diese Auswahl direk auf der Ausgabe des Prädikates `Context`; dies erspart das Einsammeln aller eindeutigen Tupel von Variable und Kontext aus der Ausgabe von `VarPointsTo`.

Dass mehrmals das selbe Tupel von Variablen erzeugt werden kann, ist möglich. Die Wahrscheinlichkeit ist aber so gering, dass dies das Ergebnis nicht siginfikant beeinflusst.

Die Quote von Alias-Treffern zur Anzahl der betrachteten Tupel ist das gewonnene Maß der Präzision.

7.1.2 Präzision bei 1-Objekt-Sensitivität

7.2 Vergleich der Ressourcennutzung

Beide Analysen, sowohl DOOP als auch die Wala-eigene PTA, benötigen eine hinreichend große Menge an verfügbarem Arbeitsspeicher, um überhaupt zu funktionieren.

In Experimenten zeigt sich, dass DOOP hinsichtlich der Performanz die in den grundlegenen Veröffentlichungen proklamierte Geschwindigkeit bietet, jedoch massive Ressourcen benötigt. Der ursprüngliche Ansatz, DOOP und WALA auf großen Programmen zu vergleichen, ist wegen dem Ressourcenverbrauch von DOOP nicht möglich. DOOP bzw. Blox legt für Pro-

gramme in der Größe von HSQLDB[1] oder jEdit [2] weit über 100GB an Auslagerungsdateien (SWAP) an, was für die zur Verfügung stehende Hardware nicht zu bewerkstelligen ist. Mit Wala ist eine Analyse dieser beiden Programme generell möglich, wie [Her11] gezeigt hat.

Demnach ist festzuhalten, dass DOOP im Ressourcenverbrauch die Wala-eigene PTA um ein Vielfaches überschreitet – bishin zum Abbruch der Analyse selbst auf gut ausgestatteter Hardware.

Diese Tatsache wird unterstützt von den der Dokumentation von DOOP, in der große Programme nur mit der kontextinsentiven DOOP-Analyse analysiert werden. Das Ergebnis dieser Analyse ist für uns aber uninteressant und wird demnach nicht in den Vergleich mit eingezogen.

Um einen Vergleich der Analysen zu ermöglichen, wird auf kleinere, Programme zugegriffen. Diese nur wenige Kilobyte großen JAR-Archive sind sowohl für DOOP als auch für Wala performant und unter Verwendung der vorhandenen Hardware analysierbar.

Gegenstand der Untersuchung ist das Programm KMY FTPD [3], welches auch Teil der Dacapo-Benchmarksuite ist. Verwendet wurde ein JAR-Archiv, welches alle Abhängigkeiten beinhaltet. FTPD ist unkomprimiert 76 KB groß (nur .class-Dateien betrachtet). Es hat sich in langwierigen Versuchen herausgestellt, dass diese Programmgröße das Maximum darstellt, welches auf der verfügbaren Hardware analysiert werden kann.

7.2.1 Ausführungszeit und Anzahl der Kanten

Zum Vergleich und zum Abschätzen der Leistung der Point-to-Analyse betrachten wir die Anzahl der (eindeutigen) Ausgabe-Kanten. Eine Ausgabe-Kante ist eine Kante

$$Var_{\Delta} \rightarrow AllocationSite$$

Die Ausführungszeiten wurden bei allen Ausführungen gemessen. WALA ist im Vergleich zu DOOP nicht in der Lage, den Arbeitsspeicher ressourcenschonend zu haushalten. Demnach musste bei der Analyse mit WALA der zu analysierende Scope stark eingeschränkt werden [4] Demnach ist der Durchsatz in PTG-Kanten je Minute eine vergleichbare Metrik.

„Wala 0-CFG“ ist die Wala-eigene PTA ohne Callsite-Sensitivität. Sie ist die einzige Wala-PTA, die stabil und sich ohne Überlaufen einer JVM-Heap-Size von 4GB als ausführbar erwiesen hat.

Das analysierte Programm ist das bereits vorgestellte „ftpd“.

[1] eine Datenbank geschrieben in Java mit 277122 LOC, `http://hsqldb.org/`

[2] ein Texteditor geschrieben in Java mit 176668 LOC, `http://jedit.org`

[3] http://peter.sorotokin.com/ftpd/kmy.net.ftpd.FTPDaemon.html

[4] Die Einschränkung ist im dieser Arbeit anhängenden Quelltext über Reguläre Ausdrücke dokumentiert.

PTA	Anzahl der PTG-Kanten	t in s	Durchsatz Kanten/s
DOOP	5856103	46800	125
Wala 0-CFA	18469	1560	12

Der Druchsatz von Wala ist deutlich geringer, zudem ist die überhaupt mögliche Größe des Programmumfanges (scope) deutlich eingeschränkt. Wala ist damit DOOP unterlegen.

7.2.2 Präzision

Die Präzision wird wie in Absatz 7.1.1 gemessen und erfasst. Wieder dienst das vorgestellte „ftpd“ als Testobjekt.

PTA	Anzahl der PTG-Kanten	Anzahl Alias	Quote Alias
DOOP	5856103	974050	16,63%
Wala 0-CFA	18469	9765	52,87%

Das Aliasing bei Wala ist mehr als doppelt so häufig wie bei DOOP. Demnach ist Wala in der Präzision bei der Analyse großer Programme DOOP unterlegen.

Kapitel 8

Zusammenfassung und Ausblick

DOOP ist eine leistungsstarke und vergleichsweise präzise Points-To-Analyse für Java, die im Gegensatz zu Wala in für den Analysten „annehmbarer Zeit“ und mit erfüllbarem Ressourcenverbrauch einen objekt-sensitiven Points-To-Graphen mit mehreren Millionen Kanten generieren kann. Im callsite-insensitiven Fall wird Wala in der Präzision um den Faktor 3 überboten.

Die vorliegende Arbeit portiert DOOP nach Wala, was die Grundlage weiterer Untersuchungen und Integrationen bilden kann. Es ist zu erwarten, dass vorhandene Analysen in Wala mit den Ergebnissen von DOOP präziser werden und sich Laufzeiten verringern.

Kompromisse müssen im Ressourcenverbrauch eingegangen werden, welcher insbesondere bei großen Points-To-Graphen die Analyse sehr erheblich ist. Leistungsstarke Hardware wird benötigt.

Absatz 3.3.2 hat zudem belegt, dass DOOP im Exception-Handlung unkorrekt ist.

Mit der Implementierung einer Null-Semantik wie in Kapitel 6.3 vorgeschlagen, kann eine sehr präzise Untersuchung auf potentielle NullPointer-Exceptions vorgenommen werden, welche auch im Bereich der Softwaresicherheit eingesetzt werden kann.

DOOP hat gezeigt, dass die Versprechungen in Performance und Leistungsfähigkeit auch in der Integration in Wala gelten und gehalten werden. Ein Verzicht auf eine bewusste Einschränkung des Analysescopes bei großen Programmen ist jedoch auch mit DOOP nicht möglich.

Anhang A

Quelltext der Implementierung

Teil dieser Diplomarbeit ist eine CD-ROM mit der im Kapitel 5 vorgestellten Implementierung in Java-Quelltext-Dateien sowie alle verwendeten und referenzierten Programmbibliotheken.

Anhang B

Eidesstattliche Erklärung

Hiermit versichere ich an Eides Statt, das diese Arbeit selbstständig und ohne Benutzung anderer Hilfsmittel als der angegebenen Quellen angefertigt wurde und alle Anführungen, die wortwörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. Diese Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Massenbachhausen, den 12. Dezember 2012

Abbildungsverzeichnis

Literaturverzeichnis

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2 edition, August 2006.

[And94] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[BS09a] Martin Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 1–12. ACM, 2009.

[BS09b] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses, 2009.

[Her11] Markus Herhoffer. Präziser kontrollfluss für exceptions durch interprozedurale datenflussanalyse. Technical report, KIT, IPD Sneltig, 2011.

[LH08] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–53, 2008.

[MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. *SIGSOFT Softw. Eng. Notes*, 27:1–11, July 2002.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[SB06] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Not.*, 41:387–400, June 2006.

[SBL11] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: Understanding object-sensitivity (the making of a precise and scalable pointer analysis). In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 17–30. ACM Press, January 2011.

[Sch95] Uwe Schöning. *Logik für Informatiker (4. Aufl.)*. Reihe Informatik. Spektrum Akademischer Verlag, 1995.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[Str00] Mirko Streckenbach. Points-to-analyse für java. Technical report, Uni Karlsruhe, 2000.