Points-To-Analyse für Java

Mirko Streckenbach
Lehrstuhl für Softwaresysteme
Universität Passau

September 2000

Zusammenfassung

Points-To-Analyse ist eine statische Programmanalyse, die potentielle Beziehungen zwischen Pointern und Daten eines Programms berechnet. Zur Points-To-Analyse für C sind in den letzten Jahren sind viele Algorithmen veröffentlicht worden, jedoch unterscheidet sich Points-To-Analyse für Java von der für C oder C++. Wir haben ein allgemeines Framework entwickelt, das es ermöglicht, verschiedene Algorithmen wie Andersen, Steensgard und deren Erweiterungen auf Java anzuwenden. Mittels einer Implementation, die den vollen Sprachumfang von Java abdeckt, wenden wir verschiedene Algorithmen auf mehrere Beispiele an und vergleichen die Ergebnisse. Dabei stellt sich heraus, daß Steensgaards Algorithmus für Java nicht ohne Einschränkungen benutzbar ist.

1 Einleitung

Points-To-Analyse ist eine statische Programmanalyse, die potentielle Beziehungen zwischen Pointern und Daten eines Programms berechnet. Sie wird für Programm-Optimierungen ebenso wie für Werkzeuge zum Programm-Verstehen eingesetzt. Für imperative Programmiersprachen gibt es zahlreiche Arbeiten in diesem Bereich (Landi/Ryder [12], Andersen [2], Steensgaard [20] Sharpiro/Horwitz [16] und andere), die meisten dieser Verfahren sind anwendbar für C. Für objektorientierte Sprachen ist zusätzlich wichtig, dynamische Bindung behandeln zu können. Für C++ kann dynamische Bindung wie Funktions-Pointer in C behandelt werden (z.B. [11]).

Java allerdings unterscheidet sich von C++ in einigen Punkten, und fast alle sind für die Points-To-Analyse relevant: Pointer-Arithmetik und Pointer auf Pointer fallen weg, die Semantik von Arrays ist anders, zudem kennt Java nur typ-sichere Type-Casts. Dynamisches Nachladen von Klassen und Festlegung von Typen für Objekte erst zur Laufzeit ist zudem ein so grundlegender Bestandteil von Java, daß es nicht vollständig ignoriert werden kann.

Wir werden im folgenden ein generelles Framework vorstellen, das es möglich macht, verschiedene bekannte Points-To-Verfahren auf Java anzuwenden.

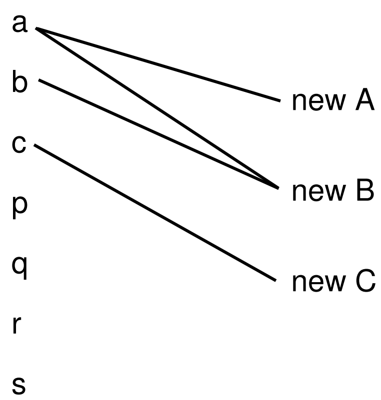
Beispiel

Ein kleines Beispiel zeigt die Probleme, die auftreten, wenn man versucht, objekt-orientierten Programmen mit dem Mitteln von Points-To-Analyse für nicht-OO Sprachen zu begegnen:

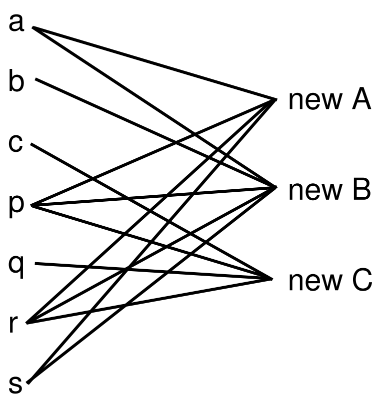
```
class MyException { Object u; MyException(Object t) { u=t } }
class A { A f(A g) { return g; }
class B extends A { A f(A g) { throw new MyException(g); } }
class C extends A { A f(A g) { return this; } }

...
```

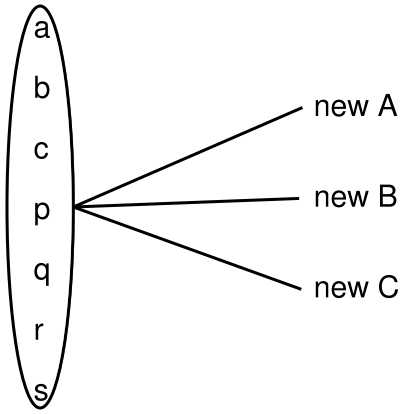
```
A a=new A(), p, q, r, s;
B b=new B();
C c=new C();

if(...)
   a=b;

try {
   p=a.f(c);
} catch(MyException e) {
   q=e.u;
}

try {
   r=p.f(a);
} catch(MyException f) {
   s=f.u;
}
```

Ignoriert man erst einmal den Methodenaufruf mit dynamischer Bindung, kann man mit Andersens Algorithmus[2] folgenden Points-To-Graphen erstellen:



Ein naiver Ansatz zur Behandlung von dynamischer Bindung wäre, für einen Methodenaufruf mit Hilfe der Klassenhierarchie alle möglichen Methoden auszurechnen und im Sinne einer konservativen Approximation ihren Aufruf anzunehmen. In diesem Beispiel könnten alle drei Methoden f aufgerufen werden. Fügt dem bisherigen Graphen die entsprechenden Kanten hinzu, erhält man folgenden Graphen (Andersen links, rechts zum Vergleich mit Steensgaards Algorithmus[20] rechts):

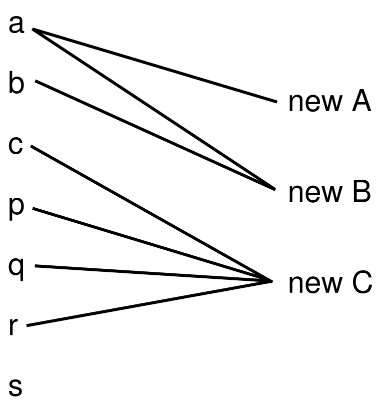


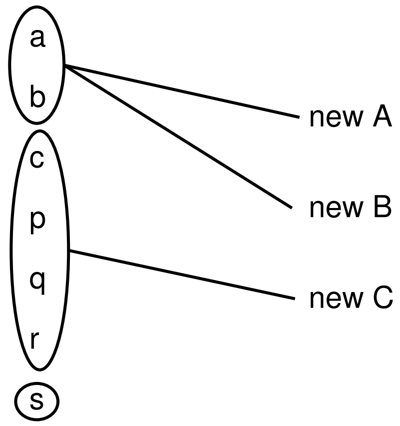


Diese Lösung bringt zwei Probleme mit sich. Zum einen ist die Qualität der Ergebnisse schlecht. Für z.B. p wird ausgerechnet, daß es auf drei verschiedene Objekte zeigen kann, dabei ist ohne weiteres ersichtlich, daß p zur Laufzeit nur auf eins zeigen kann.

Problematischer ist, daß die Ergebnisse nicht typkorrekt sind. Wir nehmen an, daß der Methodenaufruf `a.f(c)` auch f in C aufrufen kann, dabei kann a nur auf Objekte zeigen, die überhaupt nicht den notwendigen Typ für so einen Aufruf haben. Während derart undefiniertes Verhalten in C und C++ bis zu einem gewissen Grad normal ist, ist es für eine typ-sichere Sprache wie Java sinnvoll, auch die Typ-Korrektheit der Points-To-Mengen zu fordern.

Statt alle möglichen Methodenaufrufe anzunehmen, ist es sinnvoller, die Points-To-Menge der aufrufenden Variablen zu betrachten und für jedes Element dieser Menge die dynamische Bindung aufzulösen und dann entsprechende Kanten im Points-To-Graphen zu ziehen. Mit diesem Vorgehen erhält man folgende Graphen (wieder Andersen links, Steensgaard rechts):





Der Unterschied in der Qualität der Points-To-Mengen ist offensichtlich: 7 statt 13 Points-To-Beziehungen für Andersen, 8 statt 21 für Steensgaard. Dabei sind die Menge nach dem Andersen-Verfahren typ-korrekt, nach Steensgaard nicht.

Das liegt daran, daß Steensgaard für eine Zuweisung $p = q$ immer auch eine $q = p$ annimmt und damit erzwingt, daß die Points-To-Mengen von p und q gleich sind. Ist $p = q$ jedoch ein impliziter Upcast, ist die Umkehrung ein Downcast und dieser kann typ-inkorrekte Points-To-Mengen verursachen. Java garantiert, daß diese Points-To-Beziehungen zur Laufzeit nie vorkommen können, deshalb können sie in diesem Fall ignoriert werden.

Das kleine Beispiel zeigt, daß es sinnvoll ist, sich mit Points-To-Analyse speziell für typ-korrekte objekt-orientierte Sprachen zu beschäftigen und dies nicht ausschließlich als Erweiterung für imperative Programmiersprachen zu betrachten.

```
p=new A();
q=new B();
p=q;
```

Abbildung 1: Points-To-Graphen nach Andersen (links) und Steensgaard (rechts)

2 Erweiterung bekannter Points-To-Algorithmen für Java

2.1 Points-To-Analyse für Java

Viele Points-To-Algorithmen beschäftigen sich mit der Programmiersprache C. Java hat zwar eine gewisse Ähnlichkeit mit C und C++, aber gerade die Unterschiede der beiden Sprachen haben Auswirkungen auf die Points-To-Analyse. Die wesentlichen Unterscheide von Java zu C/C++ sind:

- nur Objekte auf dem Heap
  Im Gegensatz zu C, wo beliebige Speicherblöcke auf dem Heap belegt werden können, werden dort in Java ausschließlich mit `new` erzeugte Objekte abgelegt.
- dynamische Bindung anstelle von Funktions-Pointern
- andere Modellierung von Arrays
  In C ist ein Array wenig mehr als ein Speicherbereich und syntaktische Unterstützung für den Zugriff, in Java sind sie echte Objekte und haben maximale eine Dimension, mehrdimensionale Arrays werden als Arrays, die Arrays beinhalten, implementiert.
- keine Pointer-Arithmetik
- keine Pointer auf Pointer
- Typ-Korrektheit
  Java ist Typ-korrekt. Das hat unter anderem zur Folge, daß fehlerhafte Typumwandlungen Laufzeitfehler verursachen.

Die meisten dieser Punkte machen Points-To-Analyse für Java einfacher und übersichtlicher als für C/C++. Auf der anderen Seite ist dynamische Bindung in Java viel elementarer als Funktions-Pointer in C, so daß es nicht möglich ist, sie zu ignorieren, wie es manche Analysen mit Funktions-Pointern in C machen.

2.2 Andersens und Steensgaards Algorithmen

Andersen [2] und Steensgaard [20] haben Algorithmen formuliert, die informell sehr einfach zu beschreiben sind. Bei beiden Verfahren wird ein Points-To-Graph erstellt. Die Knoten des Graphen repräsentiert die Pointer und Objekte eines Programms, eine Kante zwischen zwei Knoten zeigt eine Points-To-Beziehung an. Abbildung 1 zeigt ein kleines Beispiel-Programm und die Points-To-Graphen.

Andersens Algorithmus berechnet einen ersten Points-To-Graphen aus den Zuweisungen und führt dann eine Fixpunkt-Iteration mit folgender Regel durch:

> Enthält das Programm eine Zuweisung $l = r$, so muß l auf mindestens das zeigen, worauf r zeigt.

In dem kleinen Beispiel werden durch die ersten beiden Zuweisungen die Kanten von p nach A und q nach B erzeugt. Für die dritte Zuweisung wird die Regel angewendet, und eine Kante von p nach B gezogen, da q auf B zeigt.

Der Algorithmus von Steensgaard dagegen nimmt für eine Zuweisung immer auch deren Umkehrung an und kann dadurch nach Behandlung einer Zuweisung zwei Knoten im Points-To-Graphen verschmelzen. Er berechnet Äquivalenz-Klassen von Pointern, die die gleichen Points-To-Mengen haben, nach folgender Regel:

> Enthält das Programm eine Zuweisung $l = r$, so müssen l und r auf das gleiche zeigen.

Im dem kleinen Beispiel zeigt sich der Unterschied in den Points-To-Graphen. p und q sind hier zu einem Knoten verschmolzen, als Folge wird angenommen, daß q auf A zeigen kann, was durch das Programm klar ausgeschlossen ist.

Im Gegensatz zu C/C++ ändert sich für Java das Erscheinungsbild der Points-To-Graphen. Durch den Wegfall von Pointern auf Pointer haben alle Pfade die maximale Länge 1 und die Graphen sind bipatit.

2.3 Points-To-Analyse als Mengen-Constraint-System

Da das Ziel der Analyse die Berechnung von Points-To-Mengen ist, liegt es nahe, die ganze Analyse als Constraint-System über diese Mengen zu beschreiben (vgl. [9]). Dabei verwenden wir zusätzlich zu den Points-To-Mengen drei weiteren Mengen:

- Ptr, die Menge aller Pointer
- Obj, die Menge aller Objekte
- Ass, die Menge aller Zuweisungen, wobei eine Zuweisung `l=r` als Paar (l, r) in der Menge gespeichert wird.

Die vorgestellten Algorithmen lassen sich leicht in diesem System formulieren. Die Grundregel behandelt eine Zuweisung zwischen einem Pointer und einem Objekt:

$$(p, o) \in Ass \land o \in Obj \Rightarrow o \in Pt(p)$$

Zuweisungen zwischen zwei Pointern werden von den Algorithmen verschieden behandelt. Andersen erzwingt eine Untermengen-Beziehung:

$$(p, q) \in Ass \Rightarrow (Pt(q) \subseteq Pt(p))$$

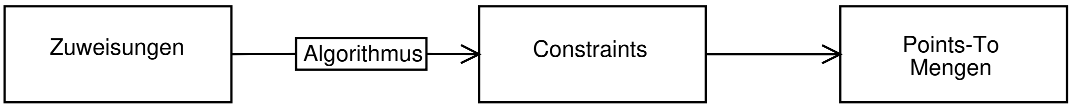
Steensgaard dagegen vereinigt die beiden Mengen:

$$(p, q) \in Ass \Rightarrow (Pt(q) = Pt(p))$$

Dieses Constraint-System wird später für die Besonderheiten von Java erweitert.

2.4 Erweiterung der Algorithmen für Java

Der Ablauf einer Points-To-Analyse mit einem Constraint-System ist also folgender:

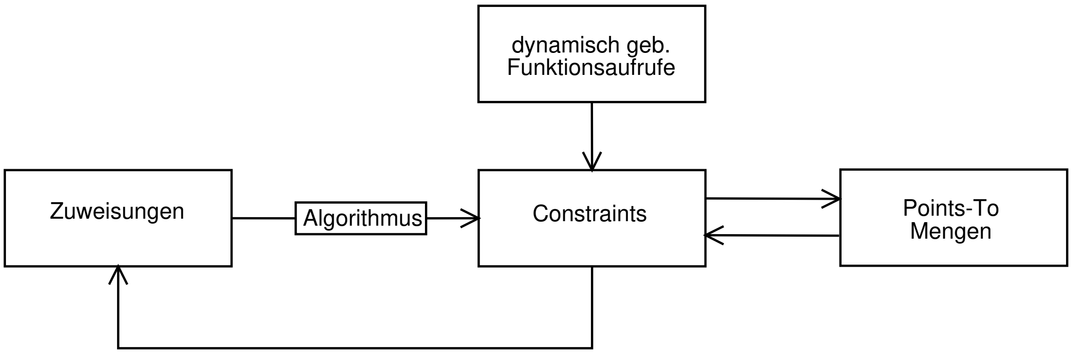


Aus den Zuweisungen eines Programms werden, abhängig vom verwendeten Algorithmus, verschiedene Constraints generiert, mit diesem Constraint-System können dann die Points-To-Mengen berechnet werden.

Wie im Beispiel gezeigt, ist es sinnvoll, für die Behandlung dynamischer Bindung auf bereits berechnete Points-To-Beziehungen zurückzugreifen. Dadurch ergeben sich ebenfalls Constraints, die aber eine etwas andere Form haben:

$$o \in Pt(p) \Rightarrow (\ldots, \ldots) \in Ass$$

Um auch diesen Constraints Rechnung tragen zu können, muß der Ablauf erweitert werden:



Die Rückkoppelung von den Constraints zu den Zuweisungen ist nötig, da sich durch dynamische Bindungen neue Zuweisungen ergeben können.[1].

Auch die Iteration in Andersens Algorithmus kann so formuliert werden, daß sie diesem Schema entspricht, indem die Regeln zur Constraint-Erzeugung etwas umformuliert werden [2]:

$$\begin{aligned} & (p,q) \in Ass \Rightarrow (Pt(q) \subseteq Pt(p)) \\ \Leftrightarrow \quad & (p,q) \in Ass \Rightarrow (o \in Pt(q) \Rightarrow o \in Pt(p)) \\ \Leftrightarrow \quad & (p,q) \in Ass \Rightarrow (o \in Pt(q) \Rightarrow (p,o) \in Ass) \end{aligned}$$

Die letzte Umformung ist erlaubt, da es innerhalb unseres Constraint-Systems immer möglich, Pointer-Objekt Beziehungen, die nicht direkt im Programm enthalten sind, sondern nur von der Analyse berechnet werden, als Zuweisung hinzuzufügen, ohne die berechneten Mengen des Constraint-Systems dabei zu verändern.

Im folgenden werden wir erweiterte Constraint-System nicht nur für dynamische Bindung, sondern auch zur Behandlung von Array-Inhalten, Data-Members von Objekten und Type-Casts benutzen.

2.5 Anwendbarkeit auf anderen Algorithmen

Die skizzierte Erweiterung läßt sich auf andere Algorithmen, die mit Steensgaards oder Andersens System verwandt sind, anwenden. Zwei Beispiele seien genannt:

Sharpiro und Horwitz[16] beschreiben einen Algorithmus, der zwischen Andersen und Steensgaard liegt, indem er nur unter parametrisierbaren Bedingungen Knoten des Points-To-Graphen

[1] [11] verwendet eine ähnliche Technik, genannt Interleaving, um Funktions-Pointer für C zu behandeln und weist bereits auf die Anwendbarkeit für dynamische Bindung hin

[2] Die letzte Umformung impliziert $o \in Pt(p) \Rightarrow (p,o) \in Ass$, welches man sinnvoll annehmen kann, da das Hinzufügen eines bereits existierenden Points-To-Beziehung als Zuweisung bei Andersen die Analyse nicht beeinflußt.

verschmilzt. Sind die Bedingungen im Rahmen des Constraint-Systems formulierbar, kann diese Analyse mit unserem Ansatz auf Java angewendet werden.

Das[6] erhöht die Genauigkeit von Steensgaards Algorithmus, indem bei einer Zuweisung nicht die Knoten direkt, sondern nur abhängige Knoten verschmolzen werden. Dies ist in Java für Data-Members und Array-Elemente anwendbar.

3 Erzeugung von Constraints für Java-Bytecode

Im folgenden werden wir beschreiben, wie Constraints zur Points-To-Analyse aus Java-Bytecode gewonnen werden können. Wir haben uns dabei für die Analyse von Bytecode entschieden, weil dieser in der Vergangenheit "stabiler" war als Java selbst. Die Einführung von Inner Classes hätte an einem Source-Code Parser größere Änderungen nötig gemacht, ein Bytecode-Parser war davon nicht oder kaum betroffen.

3.1 Analyse von Java-Bytecode

Eine Bytecode-Anweisung erhält Informationen auf zwei verschiedene Arten. Ein Teil ist mit der Anweisung im Bytecode gespeichert (im folgenden *Parameter* genannt), der andere Teil (*Operanden*) wird zur Laufzeit vom Stack oder von den Registern der JVM genommen.

Um eine Bytecode-Anweisung vollständig analysieren zu können, ist es notwendig, alle Kombinationen von Parametern und Operanden für diese Anweisung zu ermitteln. Die Parameter sind fest im Bytecode und damit statisch, Stackelemente oder Registerinhalte werden verallgemeinert zu Variablen oder Typen, um den Aufwand der Analyse zu reduzieren. Ein Beispiel zeigt dies im Detail:

```
if(...)
  a=5;
else
  a=7;
if(...)
   b=new A();
else
   b=new B();
b.f(a);
```

Für die Bytecode-Anweisung, in die `b.f(a)` übersetzt wird müssen die zum Aufruf benutzte Objekt-Referenz und die Argumente auf dem Stack stehen. In diesem Beispiel gibt es vier mögliche Kombinationen von Operanden für den Aufruf von `f`: `(A,5)`, `(A,7)`, `(B,5)` und `(B,7)`. Als erstes werden Basis-Datentypen durch ihre Typen ersetzt, da ihre Werte auf die Points-To-Analyse keine Auswirkungen haben. Damit bleiben zwei Möglichkeiten: `(A,int)` und `(B,int)`. Nun werden die Objekte-Referenzen durch entsprechende Variablen ersetzt. Damit bleibt eine Paar von Operanden über: `(a,int)`. Zusammen mit der Information, daß `b` `A` oder `B` sein kann, ist dies für die Points-To-Analyse gleichwertig dazu, alle vier Fälle einzeln zu behandeln. Für die Ersetzung von Objekt-Referenzen durch Variablen greift unsere Analyse auf Debug-Informationen im Java-Bytecode zurück. Diese Debug-Informationen können von den meisten Java-Compilern erzeugt werden.

Mit dieser Verallgemeinerung ist es möglich, Codestücke zu simulieren, und für jede Bytecode-Anweisung alle möglichen Inhalte eines simulierten Stacks zu ermitteln.

Die im folgenden verwendeten Regeln beschreiben jeweils, welche Constraints für eine einzelne Bytecode-Anweisung und einen Stackinhalt erzeugt werden. Durch die Regeln werden neben den

Constraints Typisierungs-Informationen für Pointer und natürlich der neue Stackinhalt erzeugt. Die allgemeine Form einer Regel ist deshalb folgende:

Bytecode-Anweisung mit Operanden	S=[alter Stackinhalt]	
generierte Constraints	Typisierungen für Pointer	S=[neuer Stackinhalt]

Vom Stack werden immer nur die obersten Elemente betrachtet, der restliche Stack (im folgenden symbolisiert durch ...) bleibt unverändert. Nicht bei jeder Regel sind alle fünf Elemente belegt.

Weiterhin werden die folgenden Variablen und Funktionen innerhalb der Regeln verwendet:

- I, die betrachtete Anweisung
- m, der Name der aktuellen Methode
- $typ(p)$, der Typ eines Pointers, Objektes oder Data-Members p
- $adr(I)$, die eindeutige Bytecode-Adresse der Anweisung I
- $reg(m, r, a)$, die Belegung des JVM-Registers r in Methode m an Adresse a
- $par(m, i)$, die Variable für den i-ten Parameter von m
- $sig(m)$, die Signatur der Methode m in der Form $(t_1, \ldots, t_n) \to t_r/e_1, \ldots, e_{n'}$, wobei $t_1, \ldots, t_n$ die Typen der formalen Parameter, t_r der Typ des Rückgabewertes und $e_1, \ldots, e_{n'}$ die Typen der möglichen Exceptions sind
- $cls(m)$ die Klassen, in der m definiert ist

3.2 Points-To-Constraints aus Java-Bytecode

Wir beschränken uns im folgenden auf die Bytecode-Anweisungen, die an der Constraint-Erzeugung beteiligt sind. Dies Anweisungen zum Erzeugen von Objekten (`new`, `ldc`), für Zuweisungen (`astore`), für Methodenaufrufe (`invokestatic`, `invokevirtual`, `invokespecial`, `invokeinterface`), zum Zugriff auf Data-Members (`getstatic`, `putstatic`, `getfield`, `putfield`), zum Erzeugen von Arrays (`newarray`, `anewarray`, `multianewarray`) und Zugriff auf ihren Inhalt (`aaload`, `aastore`) und für Type-Casts (`checkcast`). Die restlichen Bytecode-Anweisungen erzeugen keine Constraints, ihre Auswirkungen auf den Stack sind aber für die Analyse wichtig. Wir lassen sie aus Platzgründen weg.

3.2.1 Objekte

Im Sinne unserer Analyse ist ein Objekt auf dem Heap belegter Speicherplatz. Dies geschieht in Java primär über die `new`-Anweisung. Unsere Analyse unterscheidet Objekte nach ihrem Ort ihrer Erzeugung, d.h. den verschiedenen `new`-Anweisungen innerhalb eines Programms. Ein Objekt wird daher identifiziert durch Methodennamen und die Adresse der erzeugenden Bytecode-Anweisung.

$I \equiv \texttt{new}\ A$	$S = [\ldots]$	
	$m.{<}adr(I){>} \in Obj$ $typ(m.{<}adr(I){>}) = A$	$S = [m.{<}adr(I){>}, \ldots]$

Im Gegensatz zu `new` in Java, schließt die `new` Bytecode-Anweisung nicht der Aufruf des Konstruktors mit ein. Java-Compiler generieren immer passende `invokespecial` Anweisungen nach den `new` Anweisungen.

Objekte für Strings innerhalb eines Java Programms können ebenfalls über einen sogenannten Konstanten-Pool erzeugt werden:

$$\frac{I \equiv \texttt{ldc}\ \ s \quad\Big|\quad S = [\ldots]}{\Big|\ \begin{array}{c} m.{<}adr(I){>} \in Obj \\ typ(m.{<}adr(I){>}) = \texttt{java.lang.String} \end{array}\ \Big|\ S = [m.{<}adr(I){>}, \ldots]}$$

Arrays sind ebenfalls Objekte, entsprechende Regeln werden zusammen mit den Regeln für Zugriffe auf Arrayinhalte definiert.

3.2.2 Explizite Zuweisungen

Neben Pointern und Objekten sind für die Points-To-Analyse die Zuweisungen zwischen ihnen wichtig. Eine Zuweisung zu einer Variablen drückt sich im Bytecode als Speichern eines Wertes in einem Register aus. Die Funktion *reg* führt die anfangs erwähnte Vereinfachung aus, sie liefert den Namen einer Variablen zurück.

$$\frac{I \equiv \texttt{astore}\ \ r \quad\Big|\quad S = [p, \ldots]}{(Register(m, r, b_{i+1}), p) \in Ass \quad\Big|\quad Register(m, r, b_{i+1}) \in Ptr \quad\Big|\quad S = [\ldots]}$$

3.2.3 Methoden-Aufrufe

Als erstes sollen die Aufrufe von statischen Methoden betrachtet werden. Im Grunde stellt ein Methodenaufruf nicht mehr da, als eine Reihe von Zuweisungen zwischen formalen und aktuellen Parametern der Methode, sowie einer Zuweisung für den Rückgabewert.

Den formalen Parametern werden bei Aufruf der Methode die Werte der Argumente zugewiesen. Der Rückgabewert ist etwas komplizierter. Für die Analyse brauchen wir bei Behandlung eines Methodenaufrufs etwas, das als Ergebnis auf den Stack gelegt werden kann. Dazu wird ein künstlicher Pointer verwendet, dem dann ebenfalls innerhalb der aufgerufenen Methode die Operanden der "return"-Anweisungen zugewiesen werden.

$$\frac{\begin{array}{c} I \equiv \texttt{invokestatic}\ \ m' \\ sig(m') = (t_1, \ldots, t_n) \rightarrow t/e_1, \ldots, e_{n'} \end{array} \ \Big|\ S = [p_n, \ldots, p_1, \ldots]}{\forall_{i=1}^{n} (par(m', i), p_i) \in Ass \ \Big|\ \begin{array}{c} m'.{<}\texttt{ret}{>} \in Ptr \\ typ(m'.{<}\texttt{ret}{>}) = t \end{array} \ \Big|\ S = [m'.{<}\texttt{ret}{>}, \ldots]}$$

$$\frac{I \equiv \texttt{areturn} \quad\Big|\quad S = [p, \ldots]}{(m.{<}\texttt{ret}{>}, p) \in Ass \ \Big|\ \Big|\ S = [\ldots],}$$

Eine Methode kann anstatt mit einer Return-Anweisung beendet zu werden, auch durch das Werfen einer Exception beendet werden. Ähnlich wie für Rückgabewerte, werden hier künstliche Pointer geschaffen, die auf die geworfenen Exceptions zeigen. Da jedoch der Kontrollfluß vom Typ einer geworfenen Exception beeinflußt wird und dieser statisch nicht immer genau bestimmt werden kann, sind hier mehrere Fälle zu unterscheiden. Ein Beispiel verdeutlicht dies:

```
class A extends Exception {}
class B extends Exception {}
class C extends B {}
class D extends Exception {}

class Test {
   void f() throws A,B,D { ... }
   void g() throws D {
      try {
          f();
      } catch(A a) {
          ...
      } catch(C c) {
```

```
            ...
        } catch(B b) {
            ...
        }
    }
}
```

In `f` können vier verschiedene Exceptions mit den Typen A, B, C oder D geworfen werden. Diese lassen sich in drei Fälle unterschieden, für jeden werden verschiedene Constraints generiert.

Der einfachste Fall ist, daß die Exception den Typ `A` hat. In diesem Fall wird sie sicher vom ersten `catch`-Block gefangen und eine Zuweisung von der Exception zur Variablen `a` ist notwendig.

$$
\frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{invokestatic}\ m' \\ sig(m') = (t_1,\ldots,t_n) \to t/e_1,\ldots,e_{n'} \\ \text{Handler erwartet } t' \\ \forall_{j=1}^{n'} e_j \le t' \end{array} & S = [p_n,\ldots,p_1,\ldots] \end{array}}{\begin{array}{c|c|c} \forall_{i=1}^{n}(par(m',i),p_i) \in Ass & \begin{array}{c} m'.\texttt{<exc/}e_j\texttt{>} \in Ptr \\ typ(m'.\texttt{<exc/}e_j\texttt{>}) = e_j \end{array} & S = [m'.\texttt{<exc/}e_j\texttt{>}] \end{array}}
$$

Die geworfene Exception kann aber auch die Typen B oder C haben. C ist nicht explizit in der Liste der möglichen Exception-Typen, ist aber als Untertyp von B zulässig. Bei der Analyse von `g` kann nicht statisch entschieden werden, ob die Exception den Typen B oder einen ihrer Untertypen hat, so daß ebenfalls nicht entschieden werden kann, welcher der beiden `catch`-Blöcke die Exception wirklich fängt. In diesem Beispiel kann der zweite Block die Exception fangen, wenn sie den Typ C hat, das dritte `catch` fängt sie auf jeden Fall. c darf also nur auf die Exception-Objekte zeigen, die wirklich vom Typ C sind. Genau wie bei dynamischer Bindung wird dies mit einem bedingten Constraint realisiert.

$$
\frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{invokestatic}\ m' \\ sig(m') = (t_1,\ldots,t_n) \to t/e_1,\ldots,e_{n'} \\ \text{Handler erwartet } t' \\ \forall_{j=1}^{n'} t' < e_j \end{array} & S = [p_n,\ldots,p_1,\ldots] \end{array}}{\begin{array}{c|c|c} \forall_{i=1}^{n}(par(m',i),p_i) \in Ass & \begin{array}{c} (t')m'.\texttt{<exc/}e_j\texttt{>} \in Ptr \\ typ((t')m'.\texttt{<exc/}e_j\texttt{>}) = t' \end{array} & S = [(t')m'.\texttt{<exc/}e_j\texttt{>}] \end{array}}
$$

$$
o \in Pt(m'.\texttt{<exc/}e_j\texttt{>}) \Rightarrow \frac{\begin{array}{c|} \begin{array}{c} I \equiv \texttt{invokestatic}\ m' \\ sig(m') = (t_1,\ldots,t_n) \to t/e_1,\ldots,e_{n'} \\ \text{Handler erwartet } t' \\ \forall_{j=1}^{n'} t' < e_j \\ typ(o) \le t' \end{array} \end{array}}{\begin{array}{c|c|c} ((t')m'.\texttt{<exc/}e_j\texttt{>}, o) \in Ass & & \end{array}}
$$

Der letzte Fall ist schließlich, daß die Exception den Typ D hat und überhaupt nicht innerhalb der Methode gefangen wird. Dafür wird eine Zuweisung zwischen den Exception-Variablen der aufgerufenen und der aufrufenden Methode generiert. Da wiederum der genaue Typ nicht bekannt ist und statisch nicht entschieden werden kann, welche Zuweisungen möglich sind, wird auch hier ein bedingtes Constraint generiert.

$$
\frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{invokestatic}\ m' \\ sig(m') = (t_1,\ldots,t_n) \to t/e_1,\ldots,e_{n'} \\ \forall_{j=1}^{n'} e_j \text{wird nicht in } m \text{ gefangen} \\ sig(m) = (t'_1,\ldots,t'_{n''}) \to t'/e'_1,\ldots,e'_{n'''} \\ \forall_{k=1}^{n'''} e'_k \le e_j \end{array} & S = [p_n,\ldots,p_1,\ldots] \end{array}}{\begin{array}{c|c|c} (m'.\texttt{<exc/}e_j\texttt{>}, m.\texttt{<exc/}e'_k\texttt{>}) \in Ass\} & & \end{array}}
$$

$$
o \in Pt(m.\texttt{<exc/}e_k\texttt{>}) \Rightarrow \quad
\begin{array}{c|}
I \equiv \texttt{invokestatic}\ m' \\
sig(m') = (t_1, \ldots, t_n) \rightarrow t/e_1, \ldots, e_{n'} \\
\forall_{j=1}^{n'} e_j \,\text{wird nicht in } m \text{ gefangen} \\
m \text{ wirft Exception vom Typ } e_k \\
sig(m) = (t'_1, \ldots, t'_{n''}) \rightarrow t'/e'_1, \ldots, e'_{n'''} \\
\forall_{k=1}^{n'''} e_j < e'_k \\
typ(o) \leq e_j \\
\hline
(m'.\texttt{<exc/}e_j\texttt{>}, o) \in Ass\} \quad | \quad |
\end{array}
$$

Für Methoden, die nicht als static deklariert wurden und somit der dynamischen Bindung unterliegen, brauchen wir einen Pointer, um deren This-Pointer zu repräsentieren. Ein weiterer Hilfspointer ist notwendig, da die dynamische Bindung während der Code-Analyse noch nicht aufgelöst werden kann, somit die aufgerufene Funktion nicht bekannt ist, aber trotzdem ein Zeiger für den Rückgabewert benötigt wird. Um die dynamische Bindung auflösen zu können, muß dabei auf die Points-To Menge des Pointers, von dem der Aufruf ausgeht, zugegriffen werden.

$$
o \in Pt(q) \Rightarrow \quad
\begin{array}{c|c|c}
\begin{array}{c}
I \equiv \texttt{invokevirtual}\ m' \\
sig(m') = (t_1, \ldots, t_n) \rightarrow t/e_1, \ldots, e_{n'} \\
m'' = LookupVirtual(o, m')
\end{array} & S = [p_r \ldots p_1, q, \ldots] & \\
\hline
\begin{array}{c}
\forall_{i=1}^{n} (par(m'', i), p_i) \in Ass, \\
(m''.\texttt{<this>}, q) \in Ass, \\
(m'.\texttt{<ret/}q\texttt{>}, m''.\texttt{<ret>}) \in Ass
\end{array} &
\begin{array}{c}
m'.\texttt{<ret/}q\texttt{>} \in Ptr \\
typ(m'.\texttt{<ret/}q\texttt{>}) = t \\
m.\texttt{<this>} \in Ptr \\
typ(m.\texttt{<this>}) = cls(m)
\end{array} &
S = [m'.\texttt{<ret/}q\texttt{>}, \ldots]
\end{array}
$$

Die Funktion *LookupVirtual* entspricht dem Algorithmus der JVM Spezifikation für die Bytecode-Anweisung `invokevirtual`. Die Regel für die Bytecode-Anweisungen `invokespecial` und `invokeinterface` ist bis auf den verwendeten Lookup-Algorithmen analog zu `invokevirtual`. Genau wie die Regeln für den Rückgabewert können die Regeln für Exceptions für dynamisch gebundene Funktionsaufrufe erweitert werden.

Unter Umständen kann die dynamische Bindung bereits zum Zeitpunkt der Analyse in statische umgewandelt werden (z.B. für Klassen oder Methoden, die als "final" deklariert wurden, oder für Methoden mit Zugriffsberechtigung "private"). In diesem Fall kann für `invokevirtual` die gleiche Regel wie für `invokestatic` benutzt werden, wobei eine zusätzliche Zuweisung für den This-Pointer notwendig ist.

3.2.4 Data-Members

Data-Members (*fields*) müssen bei der Analyse nach der Art der Instantiierung unterschieden werden. Von einem Data-Member mit dem Attribut "static" existiert jeweils eine einzige Instanz. Diese kann mit zwei verschiedenen Bytecode-Anweisungen gelesen oder beschrieben werden, und ist prinzipiell vergleichbar mit einer globalen Variablen bei der Analyse von C-Programmen.

Der Bytecode erlaubt es, auf ein Data-Member über den Namen einer Unterklasse Bezug zu nehmen. D.h. der Static-Lookup muß nicht vom Compiler, sondern braucht erst zur Laufzeit durchgeführt werden[3]. Um die Bezeichner für ein Data-Member eindeutig zu machen, verwenden wir nur den Namen der Klasse, in der es definiert wurde. Die JVM Spezifikation definiert diesen Algorithmus (§5.4.3.2), der im folgenden als *LookupField* benutzt wird.

$$
\begin{array}{c|c}
\begin{array}{c}
I \equiv \texttt{getstatic}\ f \\
f' = LookupField(f)
\end{array} & S = [\ldots] \\
\hline
f' \in Ptr & S = [f', \ldots]
\end{array}
$$

[3]Verschiedene Java Compiler zeigen in dieser Hinsicht unterschiedliches Verhalten

$$\frac{\begin{array}{c|c} I \equiv \texttt{putstatic}\ f & \\ f' = \mathit{LookupField}(f) & S = [p, \ldots] \end{array}}{\begin{array}{c|c|c} (f', p) \in \mathit{Ass} & & S = [\ldots] \end{array}}$$

Data-Members ohne "static"-Attribut haben genau eine Instanz pro erzeugtem Objekt der Klasse. Entsprechend werden sie für die Points-To-Analyse modelliert: Für jedes Objekt wird für jedes enthaltende Feld ein Pointer in der Analyse erzeugt. Auf Data-Members wird zur Laufzeit immer über Pointer zugegriffen, deshalb können die Zugriffe wiederum erst mit Hilfe bekannter Points-To Beziehungen vollständig aufgelöst werden.

$$\frac{\begin{array}{c|c} I \equiv \texttt{getfield}\ f & S = [p, \ldots] \\ f' = \mathit{LookupField}(f) & p \in \mathit{Ptr} \end{array}}{\begin{array}{c|c|c} & \begin{array}{c} p.f' \in \mathit{Ptr} \\ \mathit{typ}(p.f') = \mathit{typ}(f') \end{array} & S = [p.f', \ldots] \end{array}}$$

$$o \in Pt(p) \Rightarrow \quad \frac{\begin{array}{c|c} I \equiv \texttt{getfield}\ f & \\ f' = \mathit{LookupField}(f) & \end{array}}{\begin{array}{c|c|c} (p.f', o.f') \in \mathit{Ass} & \begin{array}{c} o.f' \in \mathit{Ptr} \\ \mathit{typ}(o.f') = \mathit{typ}(f') \end{array} & \end{array}}$$

$$\frac{\begin{array}{c|c} I \equiv \texttt{putfield}\ f & S = [v, p, \ldots] \\ f' = \mathit{LookupField}(f) & p \in \mathit{Ptr} \end{array}}{\begin{array}{c|c|c} (p.f', v) \in \mathit{Ass} & \begin{array}{c} p.f' \in \mathit{Ptr} \\ \mathit{typ}(p.f') = \mathit{typ}(f') \end{array} & S = [\ldots] \end{array}}$$

$$o \in Pt(p) \Rightarrow \quad \frac{\begin{array}{c|c} I \equiv \texttt{putfield}\ f & S = [v, p, \ldots] \\ f' = \mathit{LookupField}(f) & p \in \mathit{Ptr} \end{array}}{\begin{array}{c|c|c} (o.f', v) \in \mathit{Ass} & \begin{array}{c} o.f' \in \mathit{Ptr} \\ \mathit{typ}(o.f') = \mathit{typ}(f') \end{array} & \end{array}}$$

3.2.5 Arrays

Die Arrays der JVM sind mehr als nur ein großer Speicherbereich und Syntax-Unterstützung für den Zugriff wie z.B. in C/C++. Der erste Unterschied ist, daß alle Arrays (auch Arrays von Basis-Datentypen) Objekte sind, der zweite, daß sie maximal eine Dimension haben. Mehrdimensionale Arrays sind aus eindimensionalen zusammengesetzt. Mit dem Java Statement

```
new int[5][2];
```

wird also nicht nur ein Objekt, sondern 6 erzeugt (5 *int* Arrays mit 2 Elementen und ein *int*[] Array mit 5 Elementen, die mit den anderen belegt sind).

Unsere Analyse ignoriert die Indizes und Größen-Angaben bei Arrays, so daß in diesem Fall nur zwei Objekte unterschieden werden können: eines vom Typ *int*[] und eines vom Typ *int*[][]. Mit drei verschiedenen Bytecode-Anweisungen können Arrays erzeugt werden, eine für Arrays von Basis-Datentypen, eine für Arrays von Objekt-Typen und eine für Arrays von Arrays.

$$\frac{\begin{array}{c|c} I \equiv \texttt{newarray}\ t & S = [c, \ldots] \end{array}}{\begin{array}{c|c|c} & \begin{array}{c} m.{<}\mathit{adr}(I){>} \in \mathit{Obj} \\ \mathit{typ}(m.{<}\mathit{adr}(I){>}) = t[] \end{array} & S = [m.{<}\mathit{adr}(I){>}, \ldots] \end{array}}$$

$$\frac{\begin{array}{c|c} I \equiv \texttt{anewarray}\ t & S = [c, \ldots] \end{array}}{\begin{array}{c|c|c} & \begin{array}{c} m.{<}\mathit{adr}(I){>} \in \mathit{Obj} \\ \mathit{typ}(m.{<}\mathit{adr}(I){>}) = t[] \end{array} & S = [m.{<}\mathit{adr}(I){>}, \ldots] \end{array}}$$

$$\begin{array}{c|c}
I \equiv \texttt{multianewarray}\ t\ n & S = [c_n, \ldots, c_1, \ldots] \\
\hline
\begin{array}{c} \forall_{i=0}^{n-1} m. < adr(I) \underbrace{[] \ldots []}_{n} > \in Obj \\ typ(\forall_{i=0}^{n-1} m. < adr(I) \underbrace{[] \ldots []}_{n} >) = A \underbrace{[] \ldots []}_{n+1} \end{array} & S = [m.\langle adr(I)\rangle, \ldots]
\end{array}$$

Für den Inhalt jedes Arrays werden wie bei Data-Members aufgrund der in der Analyse vorhandenen Objekte zusätzliche Pointer erzeugt. Auch hier geschieht der Zugriff über Pointer und kann nur mit den Ergebnissen der Points-To Analyse aufgelöst werden. Bei schreibenden Zugriff auf ein Array findet zusätzlich eine Typprüfung statt, ob das neue Element zum Typ des Arrays paßt.

$$\begin{array}{c|c}
I \equiv \texttt{aaload}\ f & \begin{array}{c} S = [c, p, \ldots] \\ p \in Ptr, typ(p) = t[] \end{array} \\
\hline
\begin{array}{c} p[] \in Ptr \\ typ(p[]) = t \end{array} & S = [p[], \ldots]
\end{array}$$

$$o \in Pt(p) \Rightarrow \begin{array}{c|c}
I \equiv \texttt{aaload}\ f & \begin{array}{c} p \in Ptr \\ typ(o) = t[] \end{array} \\
\hline
(p[], o[]) \in Ass & \begin{array}{c} o[] \in Ptr \\ typ(o[]) = t \end{array}
\end{array}$$

$$\begin{array}{c|c}
I \equiv \texttt{aastore}\ f & \begin{array}{c} S = [v, c, p, \ldots] \\ p \in Ptr, typ(p) = t[] \end{array} \\
\hline
(p[], v) \in Ass & \begin{array}{c|c} \begin{array}{c} p[] \in Ptr \\ typ(p[]) = t \end{array} & S = [\ldots] \end{array}
\end{array}$$

$$o \in Pt(p) \Rightarrow \begin{array}{c|c}
I \equiv \texttt{aastore}\ f & \begin{array}{c} S = [v, c, p, \ldots] \\ p \in Ptr \\ typ(o) = t[] \\ o[] \leq typ(v) \end{array} \\
\hline
(o[], v) \in Ass & \begin{array}{c} o[] \in Ptr \\ typ(o[]) = t \end{array}
\end{array}$$

3.2.6 Type-Casts

In C und C++ haben Typ-Casts wenig Auswirkungen auf die Points-To-Analyse, in Java ist dies anders, wie ein Beispiel zeigt:

```
...
   Object o=new Object();
   A a=(A)o;
   Object p=a;
...
```

Bei einem analogen C++ Fragment könnte zur Laufzeit `p` auf das erzeugte Objekt zeigen. In Java würde dagegen beim Cast nach `A` ein Laufzeitfehler erzeugt werden. Um unsere Analyse so genau wie möglich zu halten, versuchen wir, dies in der Points-To-Analyse nachzubilden.

Für das Ergebnis eines Type-Casts wird ebenfalls ein eigenständiger Pointer erzeugt. Für diesen Pointer werden dann für alle Objekte, auf die der ursprüngliche Pointer zeigen kann und deren Typ paßt, Zuweisungen gemacht. Bereits zum Zeitpunkt der Analyse kann man jedoch einige Casts als unnötig verwerfen und den Ursprungspointer als Ergebnis des Casts verwenden.

$$\begin{array}{c|c}
I \equiv \texttt{checkcast}\ t & \begin{array}{c} S = [p, \ldots] \\ (p = null \lor typ(p) \leq t) \end{array} \\
\hline
 & S = [p, \ldots]
\end{array}$$

$$\frac{I \equiv \texttt{checkcast}\ t \;\Big|\; \begin{array}{c} S = [p, \ldots] \\ \neg(p = null \vee typ(p) \leq t) \end{array}}{\Big|\; \begin{array}{c} (t)p \in Ptr \\ typ((t)p) = t \end{array} \;\Big|\; S = [(t)p, \ldots]}$$

$$o \in Pt(p) \Rightarrow \frac{I \equiv \texttt{checkcast}\ t \;\Big|\; \begin{array}{c} S = [p, \ldots] \\ \neg(p = null \vee typ(p) \leq t) \\ typ(o) \leq t \end{array}}{((t)p, o) \in Ass \;\big|\; \big|}$$

3.3 Points-To-Constraints für Analyse-Verfahren

Die im vorigen Abschnitt präsentierten Regeln für die Generierung von Constraints für die Points-To-Analyse sind unabhängig vom verwendeten Points-To-Verfahren. Alle Zuweisungen werden als Paare in der Menge Ass gespeichert. Um das Mengen-System zu komplettieren müssen also noch Constraints angegeben werden, die diese Zuweisungspaare in Elemente der Points-To-Mengen umsetzen. Die einfachste Regel dabei ist:

$$(p, o) \in Ass, o \in Obj \Rightarrow o \in Pt(p)$$

Durch diese Regel wird sichergestellt, daß bei einer Zuweisung eines Objektes an einen Pointer dieser in der Points-To-Menge des Pointers enthalten ist.

In Ahnlehnung an unsere Implementation werden wir unser Constraint-System im folgenden KABA nennen.

3.3.1 KABA/Andersen

Die Regeln für eine Zuweisung zwischen zwei Pointern unterscheiden sich bei den beiden Verfahren. Bei Andersen wird in so einem Fall die Points-To-Menge der rechten Seite eine Untermenge der Points-To-Menge der linken Seite der Zuweisung:

$$(p, q) \in Ass, q \in Ptr, o \in Pt(q) \Rightarrow o \in Pt(p)$$

3.3.2 KABA/Steensgaard

Steensgaard dagegen erzwingt die Gleichheit der beiden Points-To-Mengen. Wie bereits im Beispiel gezeigt, hat dies Points-To-Beziehungen zur Folge, die nicht typkorrekt sind. Allerdings kann man mit einer zusätzlichen Bedingung die Typkorrektheit garantieren. Dadurch werden die Regeln unsymmetrisch und unterscheiden sich deutlich vom Steensgaard-Verfahren für C/C++.

$$(p, q) \in Ass, q \in Ptr, o \in Pt(q) \Rightarrow o \in Pt(p)$$

$$(p, q) \in Ass, q \in Ptr, o \in Pt(p), typ(o) \leq typ(q) \Rightarrow o \in Pt(q)$$

3.3.3 KABA / One Level Flow Analysis

Das[6] erweiterte Steensgaards Algorithmus zur One Level Flow Analysis. Dabei werden durch eine Zuweisung Knoten nicht mehr direkt verschmolzen, sondern nur die Knoten für die dereferenzierten Varianten der zugewiesenen Pointer. Da Java keine Pointer auf Pointer kennt, ist dies Verfahren nicht direkt auf Java übertragbar. Es kann jedoch auf Data-Members und Arrays angewendet werden. Ein entsprechendes Constraint für ein Data-Member f würde so aussehen:

$$(p, q) \in Ass \Rightarrow Pt(p.f) = Pt(q.f)$$

Für Arrays ist es ähnlich anwendbar, jedoch werden die Constraints komplexer, da hier die Typ-Korrektheit extra berücksichtigt werden muß. Da dies gegenüber dem Original-Algorithmus für C jedoch stark eingeschränkt ist, ist zu beschweifeln, daß der Geschwingdingskeits bzw. Genauigkeitsgewinn ähnlich groß sein wird.

4 Whole-Program-Analysis

Ein generelles Problem bei der Analyse von Programmen besteht darin, daß niemals der vollständige Quell-Code analysiert werden kann. Meistens sind Bibliotheksquellen nicht verfügbar oder nicht analysierbar (z.B. durch massiven Einsatz von Assembler). In Java sind manche API Funktionen "native" und somit ist nahezu immer eine gemischte Java/C oder Java/C/Assembler Analyse notwendig. Eine solche Analyse ist zwar theoretisch nicht unmöglich, aber extrem aufwendig.

Für die Points-To-Analyse ergibt sich daraus ein Problem, da sich der nicht-analysierte Code auf die Points-To-Beziehungen in analysiertem Code auswirken kann. Ein kleines Beispiel zeigt dies:

```
class A {
   static native Object f();
}
...
   o=A.f();
...
```

In diesem Fall kann die Methode `f` nicht analysiert werden und gibt einen Pointer zurück. Worauf zeigt nun der Pointer `o`? Für die weitere Analyse ist dieses Wissen unbedingt notwendig, z.B. um dynamische Bindung aufzulösen. Die Möglichkeiten reichen von einer leeren Points-To-Menge bis zu einer mit allen Objekten, die einen passenden Typ haben. Dabei ist es nicht schlimm, wenn die Annahmen über o zu konservativ sind, dies verschlechtert zwar die Analyse-Ergebnisse, bewahrt aber die Korrektheit. Nimmt man für o eine leere Points-To-Menge an, kann dies jedoch die Korrektheit der ganzen Analyse zerstören, wenn z.B. ein dynamisch gebundener Funktionsaufruf überhaupt nicht aufgelöst wird.

Man kann diesem Problem begegnen, indem man für aufgerufene Methoden, deren Quell-Code nicht analysiert wird, sogenannte Stubs programmiert, die das Verhalten der Original-Methode im Bezug auf die Points-To-Analyse simulieren. Dies erfordert jedoch einen hohen Aufwand, da die exakte Semantik jeder einzelnen Methode bekannt sein muß. Da schon die Standard Java-APIs über mehrere Tausend Methoden verfügen, scheitert dieser Ansatz schnell an dem dafür notwendigen Aufwand.

Wir machen statt dessen eine konservative Approximation. Wir nehmen an, daß nicht-analysierter Code mit dem ihm übergebenen Pointern und Objekten alles technisch mögliche tut und daß jedes Objekt, das irgendwo an nicht-analysierten Code übergeben wird, an jeder anderen Stelle wieder auftauchen kann, als ob der gesammte nicht-analysierte Code über eine globale Variable kommuniziert. Diese Abschätzung verschlechtert zwar die Qualität der Analyse-Ergebnisse, ist aber der einzige Weg, die Korrektheit der Analyse sicherzustellen, ohne manuell Informationen für den nicht-analysierten Code in die Analyse einzubringen.

Die Approximation gliedert sich in zwei Teile: Es muß zuerst festgestellt werden, wo Objekte an nicht-analysierten Code übergeben werden und von diesem zurückgegeben werden, dann muß festgestellt werden, was der nicht-analysierte Code mit diesen Objekten anfangen kann.

Unanalysierter Code wird in der Points-To-Analyse als globale Variable modelliert. Dazu wird ein Pointer $unanalysed$ geschaffen. Jede Zuweisung $(unanalysed, p)$ bedeutet, daß p an nicht-analysierten Code übergeben wird. Für Pointer, die von unanalysiertem Code an den analysierten übergeben werden, hat man zusätzlich noch eine Typschranke. Dafür werden weitere Pointer *unanalysed/t* geschaffen, die jeweils auf alle Objekte zeigen, auf die nicht-analysierter Code Zugriff hat, und deren Typ ein Untertyp von t ist. Das folgende Constraint erzeugt die entsprechenden Zuweisungen:

$$o \in Pt(unanalysed) \wedge typ(o) \leq t \Rightarrow o \in Pt(unanalysed/t)$$

4.1 Datenfluß von/zu unanalysiertem Code

Für Datenfluß zwischen analysiertem und nicht-analysiertem Code gibt es zwei verschiedene Möglichkeiten: Funktionsaufrufe und statische Data-Members. Wird eine Funktion von nicht-analysiertem Code aufgerufen, werden ihre Parameter an den nicht-analysierten Code übergeben. Der Rückgabe-Wert ist dann eine Variable für den nicht analysierten Code mit entsprechendem Typ. Dies wird durch die folgenden Constraints ausgedrückt:

$$\frac{\begin{array}{c} m \text{ ist nicht-analysiert,} \\ sig(m) = (t_1, \ldots, t_n) \rightarrow t/e_1, \ldots, e_{n'} \end{array}}{\forall_{i=1}^{n}(unanalysed, m.{<}par(m,i){>}) \in Ass}$$

$$\frac{m \text{ ist nicht-analysiert, } sig(m) = (t_1, \ldots, t_n) \rightarrow t/e_1, \ldots, e_{n'}}{(m'.{<}\texttt{ret}{>}, unanalysed/t) \in Ass}$$

Für Exceptions gelten Constraints analog zu den Rückgabewerten. Ein statisches Data-Member ist wie eine globale Variable. Nicht analysierter Code kann ihren Wert auslesen oder neu schreiben. Allerdings kann dies sinnvoll durch die Zugriffsrechte auf die Variable eingeschränkt werden: ein Data-Member, das "private static" ist, kann nur gelesen oder verändert werden, wenn die Klasse, in der es definiert ist, selbst unanalysierten Code enthält. Ein Data-Member, das "final" ist, kann nur gelesen und nicht geschrieben werden, solange die entsprechenden Initialisierungs-Routinen der Klasse analysiert sind. In ähnlicher Weise kann man für die anderen Attribute den Zugriff von unanalysiertem Code einschränken. Für die folgenden Constraints ignorieren wir diese Details und unterscheiden nur zwischen lesbar und schreibbar:

$$\frac{f \text{ ist static, } f \text{ ist lesbar von nicht-analysiertem Code}}{(unanalysed, f) \in Ass}$$

$$\frac{f \text{ ist static, } f \text{ ist schreibbar von nicht-analysiertem Code}}{(f, unanalysed/typ(f)) \in Ass}$$

4.2 Einfluß von nicht analysiertem Code

Weiterhin muß man wissen, was nicht analysierter Code mit den ihm übergebenen Objekten machen kann.

- Methoden aufrufen (eingeschränkt durch Zugriffsrechte)
- Data-Members lesen/schreiben (eingeschränkt durch Zugriffsrechte)
- Arrayinhalte lesen/schreiben
- Objekte erzeugen

Als Parameter für die Methodenaufrufe kann dabei *unanalysed* mit passendem Typen gewählt werden, der Rückgabewert wird ebenfalls *unanalysed* zugewiesen.

$$o \in Pt(unanalysed/t) \Rightarrow \frac{\begin{array}{c} m \text{ ist Methode in } t, \\ m \text{ bekannt im nicht-analysierten Code}, \\ sig(m) = (t_1, \ldots, t_n) \rightarrow t/e_1, \ldots, e_{n'} \\ m' = LookupVirtual(o, m'), \end{array}}{\begin{array}{c} \forall_{i=1}^{n}(par(m', i), unanalysed/t_i) \in Ass, \\ (m'.\texttt{<this>}, o) \in Ass, \\ (unanalysed, m'.\texttt{<ret>}) \in Ass \end{array}}$$

In diesem Constraint fällt die Einschränkung auf, daß die aufgerufene Methode im nicht-analysierten Code bekannt sein muß. Ein Beispiel erläutert dies:

```
class A {
   String toString() { ... }
   String toString2() { ... }
}
...
   System.out.println(new A());
...
```

In diesem Beispiel wird das durch `new A()` erzeugte Objekt an nicht-analysierten Code übergeben. Der kann nur die Methode `toString` aufrufen, weil sie aus `Object` stammt und in `A` überschrieben wurde. Die Methode `toString2` ist in `A` neu und kann deshalb nicht innerhalb des nicht-analysierten Codes bekannt sein und von dort (von der Reflection API einmal abgesehen) nicht aufgerufen werden.

Constraints für Data-Members und Arrays werden wie Methoden analog zu den entsprechenden Regeln für Bytecode generiert, nur ist das Bezugsobjekt hier immer *unanalysed*.

Der letzte Punkt ist etwas aufwendiger. Es gibt mehrere Möglichkeiten in Java, Objekte nicht mit `new`, sondern durch Aufruf von API-Funktionen zu erzeugen. Dazu gibt es innerhalb der Reflection-API die (Meta-)Klasse `java.lang.Class`, deren Objekte Klassen repräsentieren. Mit so einem Objekt kann man eine Instanz der entsprechenden Klasse erzeugen. Zugriff auf diese Objekte bekommt man z.B. aus einer existierenden Instanz oder durch Angabe des Klassennamens als String. Damit ist es möglich, in einem Programm Objekte von Klassen zu erzeugen, die zum Zeitpunkt der Analyse nicht bekannt waren oder nicht mit analysiert wurden. Dies kann natürlich die Ergebnisse der Points-To-Analyse verfälschen, weil z.B. dynamische Bindung nicht mehr richtig aufgelöst wird.

Man muß davon ausgehen, daß der nicht analysierte Code von allen bekannten Klassen Objekte erzeugt (wir nennen diese Objekte *uacreated*).

$$\frac{t \text{ kommt in der Analyse vor}}{uacreated/t \in Obj, typ(uacreated/t) = t, (unanalysed, uacreated/t) \in Ass}$$

Da es möglich ist, Objekte von Klassen nachzuladen, die während der Analyse überhaupt nicht berücksichtigt wurden, reicht dies aber nicht aus.

```
class A {
   private Object f;
   void set(Object f1) {
      f=f1;
   }
   Object get() {
      return new String("foo");
   }
   protected Object g() {
      return f;
   }
```

```
}
...
   A a=... /* von unanalysiertem Code erzeugt */
   a.set(new String("bar"));
   o=a.get();
```

In diesem Beispiel ist es einfach, zu sehen, worauf `o` am Ende zeigen kann, nämlich auf den String mit dem Inhalt `foo`. Was passiert aber, wenn der nicht analysierte Code gar kein Objekt vom Typ A, sondern eines vom Klasse `B` erzeugt, die wie folgt aussieht:

```
class B extends A {
   Object get() {
      return g();
   }
}
```

In diesem Fall würde `o` auf den String `bar` zeigen. Um dieses Problem zu umgehen, muß man für die Analyse zu jeder analysierten Klasse eine Unterklasse erzeugen, die alle Methoden überschreibt, und nur aus nicht-analysiertem Code bestellt. Das würde in dem Beispiel dazu führen, daß der Aufruf von g nach dem oben beschriebenen Mechanismus erkannt und korrekte Points-To-Informationen berechnet würden.

4.3 Bessere Ergebnisse für Reflection-API

Innerhalb der Reflection-API gibt es einige Methoden, die in der JDK-Implementation native sind, also nicht analysierter Code wären. Diese Methoden stellen elementare Funktionalität, wie das Zugriff auf Array-Elemente oder Data-Members, Aufruf von Methoden oder Erzeugen von Objekten zur Verfügung. Dabei kann man in manchen Fällen bessere Ergebnisse in Hinblick auf die Points-To-Analyse bekommen, wenn man sie detailiert analysiert, anstelle sie der Approximation für nicht-analysierten Code zu überlassen.

Stellvertretend soll hier die Erzeugung von Objekten mit zur Laufzeit festgelegten Typen beschrieben werden. Das folgende Programm dient als Beispiel:

```
class A {
   void f() { ... }
}

class B extends A {
   void f() { ... }
}

class Main {
   void main() throws IllegalAccessException, InstantiationException {
      A a=new A();
      Class c=a.getClass();
      Object o=c.newInstance();
      Aa a2=(A)a;
      a2.f();
   }
}
```

Was dieses Programm tut, ist einfach zu beschreiben: Es erzeugt ein Objekt der Klasse `A`, erzeugt dann mit Hilfe des zu `A` zugehören `Class`-Objektes eine neue Instanz von `A` und ruft für dieses Objekt schließlich `f` auf.

Sowohl `getClass` als auch `newInstance` sind nicht analysierbarer Code, d.h. `c` zeigt bei einer Analyse auf *uacreated/java.lang.Class* und die leicht zu ersehende Zusatz-Information, daß es sich um das `Class`-Objekt für `A` handelt, geht vorloren. Daher würde die Approximation annehmen, daß o auf *uacreated/A*, *uacreated/B* und das erste `A` Objekt zeigen kann und die dynamische Bindung von `a2.f()` könnte nicht mehr eindeutig zu `A.f` aufgelöst werden, es müßte ebenso `B.f` angenommen werden.

Deshalb ist es sinnvoll, die Methode `getClass()` gesondert zu behandeln, und für sie genauere Rückgabewerte zu erstellen: Wir kreieren für jede bekannte Klasse mit Typ t im Programm ein individuelles Objekt *class/t*. Bei Aufruf von `getClass` wissen wir nach Auflösung der dynamischen Bindung, welches dieser Objekte wird zurückgeben müssen. Im Beispiel würde `c` als nur auf *class/A* zeigen, da das `a` aus `a.forClass()` wiederum nur auf ein `A`-Objekt zeigen kann. Diese Information kann nun von einer speziellen `newInstance` Methode aufgegriffen werden, die anstelle von *unanalysed/java.lang.Objekt*, wie sie es die Approximation für nicht-analysierten Code tun würde, nur die Objekte in *unanalysed* zurückliefert, deren Typ wirklich A ist. Damit würden `o` und `a2` auf *uacreated/A* und das `A`-Objekt aus dem Programm zeigen, und die dynamische Bindung von `a2.f()` kann eindeutig aufgelöst werden.

Sicherlich behandelt diese Methode nur wenige Spezialfälle, dafür läßt sie sich ebenso auf Methoden-Aufrufe und Data-Member Zugriff der Reflection-API anwenden und ist nicht besonders teuer.

Dieses Vorgehen ist vergleichbar mit dem Erstellen von Stubs, allerdings erstellen wir nur wenige Sonderfälle, um die Analyse zu verbessern, notwendig für die Korrektheit ist es nicht. Zudem ist dieser Mechanismus mächtiger als Stubs.

5 Implementation

Obwohl unsere Analyse als Mengen-Constraint-System beschrieben ist, benutzt unsere Implementation einen "traditionellen" Points-To-Graphen zur Speicherung der Points-To-Informationen. Ein Knoten in diesem Graphen repräsentiert einen oder mehrere Pointer und speichert die Menge von Objekten, auf die diese Pointer zeigen. Eine Zuweisung von einem Objekt zu einem Pointer verändert direkt diese Menge. Zuweisungen zwischen Pointern werden von den beiden Algorithmen unterschiedlich behandelt.

Unser Algorithmus gliedert sich in zwei Phasen: Zuerst wird der Graph mit allen Pointern erzeugt und die statischen Zuweisungen hinzugefügt. Mit Hilfe dieser dann existierenden Points-To-Informationen wird die Fixpunkt-Iteration gestartet, in der aus den bedingten Constraints ggf. weitere Zuweisungen erzeugt werden. Diese wird solange wiederholt, bis keine neuen Zuweisungen mehr erzeugt werden und der Fixpunkt erreicht ist.

Ein vollständiger Durchlauf über alle Constraints dauert sehr lange. Um nicht in jeder Iteration alle bedingten Constraints prüfen zu müssen, wird eine Liste von Pointern bzw. Knoten geführt, deren Points-To-Beziehungen sich während des letzten Durchlaufes verändert haben. Nur diese werden in nächsten Iteration betrachtet.

5.1 Andersen-Verfahren

Die Implementation des Andersen-Verfahrens zieht eine Kante zwischen zwei Pointern, wenn eine Zuweisung zwischen den beiden existiert. Entlang dieser Kante können dann Objekte durch die Points-To-Mengen propagiert werden, so daß in jedem Knoten die vollständige Points-To Menge der zugehörigen Pointer gespeichert ist. Die Liste von veränderten Pointern für die Iteration zu führen ist in diesem Fall einfach.

Alle Points-To-Mengen explizit auszurechnen ist leider mit einem enormen Speicherbedarf verbunden. Selbst für kleine Programme kann man feststellen, daß diese Implementation sehr schlecht skaliert.

Eine Verbesserungsmöglichkeit besteht darin, nicht mehr die vollständige Points-To-Menge in einem Knoten zu speichern, sondern nur noch die Objekte, die den entsprechenden Zeigern direkt zugewiesen wurden und bei Bedarf den Graphen zu traversieren und die vollständige Menge zu errechnen. Dies hat neben dem deutlich geringeren Speicherbedarfs den Vorteil, daß keine Points-To-Mengen berechnet werden, die später gar nicht gebraucht werden. Dafür ist der Aufwand höher, für die Iteration festzustellen, welche Pointer verändert wurden. Wie sehr sich der Mehr-Aufwand auswirkt, hängt letztendlich vom Zugriffsmuster auf die Analyse-Ergebnisse ab.

5.2 Zyklenerkennung

Beim Traversieren des Graphen muß bereits darauf geachtet werden, jeden Knoten nur einmal zu besuchen, um Schleifen zu verhindern. Inspiriert von [8] haben wir dies zu einer (im Vergleich allerdings primitiven) Zyklenerkennung ausgebaut, die Zyklen verschmilzt, wenn sie beim Abfragen der Points-To-Mengen erkannt werden.

Wie [14] haben wir ebenfalls eine Zyklensuche mit Hilfe von stark verbundenen Komponenten implementiert, die zwischen Aufbau des Graphen durch statische Zuweisungen und der Constraint-Iteration stattfindet.

In beiden Fällen sind die Ergebnisse enttäuschend, da jeweils kaum Zyklen gefunden werden. Dies liegt an der Typ-Korrektheit des Points-To-Graphen, die für einen Zyklus immer erzwingt, daß alle beteiligten Pointer den gleichen Typ haben. Denkbar ist auch, daß der Wegfall von Pointern auf Pointer in Java hier eine Rolle spielt.

Sofern es für auf der Points-To-Analyse aufbauende Verfahren genutzt werden kann (wie in unserem Fall), ist das Berechnen von Äquivalenzklassen ein Mehrwert der Points-To-Analyse, der sich eventuell im Ganzen amortisieren kann, auch wenn sie die Points-To-Analyse nicht unbedingt beschleunigt.

5.3 Steensgaard-Verfahren

Auch bei der Implementierung des Steensgaard-Verfahrens repräsentiert ein Knoten ein oder mehrere Pointer und speichert die Objekte, auf die diese Pointer zeigen können. Anstelle eine Kante zwischen zwei Knoten zu ziehen, verschmilzt eine Zuweisung allerdings die beteiligten Knoten. War die Zuweisung ein Upcast, werden dadurch ggf. inkorrekte Points-To-Beziehungen erzeugt. Dies macht eine zusätzliche Typprüfung für alle Objekte der Points-To-Menge für einen bestimmten Pointer notwendig, zudem können die Pointer eines Knotens nicht direkt als Äquivalenzklasse betrachtet werden.

Die Vereinigungs-Operation benutzt für die Menge der Pointer der beider Knoten den Union/Find-Algorithmus. Für die Objekt-Mengen kann dieser Algorithmus nicht angewendet werden, da hier potentielle Duplikate entfernt werden müssen, sonst wären die Points-To-Mengen Multi-Mengen. Dadurch dauert die Iteration entsprechend länger, weil die Points-To-Mengen größer sind, zudem wird ein explizierter Test auf das Erreichen des Fixpunktes notwendig.

Durch das Verschmelzen von Knoten werden manche Constraints redundant, das dolgende Beispiel zeigt dies:

```
class A {}
class B extends A {}
...
```

```
   A a;
   B b;
...
   a=b;
   b=(B)a;
```

Durch die beiden Zuweisungen werden die Pointer `A`, `B` und `(B)a` in einem Knoten verschmolzen. Dadurch wird das Constraint

$$o \in Pt(\mathtt{a}) \wedge typ(o) \leq B \Rightarrow ((\mathtt{B})\mathtt{a}, o) \in Ass$$

redundant. Redundante Constraints könnte man finden und eliminieren, aber dazu ist uns kein Verfahren bekannt, das schnell genug ist, um dadurch einen Vorteil zu erlangen.

Ein weiteres Beispiel zeigt, daß das Verschmelzen von Knoten mit weiteren Problemen verbunden ist:

```
class Printer {
   static print(Object o) {
      ...
   }
}
class A {
   f() {
      g();
      Printer.print(this);
   }
   g() {
   }
}
class B extends A {
   f() {
      Printer.print(this);
   }
   g() {
   }
}
...
   A a;
   a=new A();
   a.f();
   a=new B();
   a.f();
```

Durch die beiden Methodenaufrufe von `print` werden die Pointer für `A.f().this`, `B.f().this` und `o` verschmolzen. Durch die `f`-Aufrufe zeigen diese Pointer nun auf die mit `new` erzeugten Objekte `A` und `B`. Dies führt dazu, daß der Aufruf von `g` in `A.f()` u.a. dynamisch zu `B.f()` aufgelöst wird, was zur Laufzeit nicht passieren kann.

Es stellt sich die Frage, ob es überhaupt sinnvoll ist, ein so aufwendiges Constraint-System, wie wir es vorgestellt haben, mit dem Steensgaard-Verfahren zu verwenden. Abhängig ist dies sicherlich von der Verwendung der Analyse-Ergebnisse. Ist nur eine grobe Abschätzung verlangt, kann der Steensgaard-Algorithmus sicherlich zur Anwendung kommen. Für präzise Ergebnisse insbesondere in Bezug auf dynamische Bindung scheint es nicht brauchbar.

	LOC	Klassen	PTR	Gewinn	direkt	transitiv	collapse
JLex	7823	26	7047	32.63%	54.86s	56.66s	71.96s
Mars	2903	19	4271	29.70%	29.74s	22.57s	26.46s
graph	7045	32	8418	31.24%	53.12s	45.54s	54.22s
hanoi	4958	45	7471	44.58%	23.47s	38.89s	47.51s
jEdit	11862	108	18599	27.08%	359.57s	487.69s	1111.30s
jas	5436	127	12590	15.03%	412.10s	606.97s	706.15s

Tabelle 1: Beispiele für Andersens Algorithmus

6 Fallstudien

Wir haben unsere Analyse auf mehrere Beispiele verschiedener Größe angewandt. Tabelle 1 zeigt die Ergebnisse für die verschiedenen Varianten von Andersens Algorithmus, Tabelle 2 für Steensgaards[4]. Da die Größe von Points-To-Mengen an sich keine Aussagekraft hat, geben wir die Genauigkeit der Analyse-Ergebnisse als Verbesserung gegenüber der naiven Points-To-Analyse an:

$$Pt(p) = \{o \in Obj | typ(o) \leq typ(p)\}$$

6.1 Qualität der Analyse-Ergebnisse

Für alle Beispiele ist die Genauigkeit des Andersen-Verfahrens deutlich höher als die des Steensgaard-Verfahrens. Dabei errechnet das Andersen-Verfahren zwischen 15 und 45% Prozent weniger Points-To-Beziehungen als die naive Analyse, das Steensgaard-Verfahren zwischen nahezu 0 und 15% bessere. Der Unterschied beider Verfahren schwankt zwischen 15 und 30%. Es ist eine Tendenz erkennbar, daß die Unterschiede größer werden, wenn die Ergebnisse für Andersen besser werden. Ein Zusammenhang zur Größe des Programms ist nicht erkennbar.

Wir sehen durch diese Ergebnisse unsere oben genannten Bedenken, Steensgaards Verfahren auf Java zu übertragen bestätigt.

6.2 Zeit-Dauer der Analyse

Vergleicht man die Ergebnisse für Andersen und Steensgaard fällt sofort ins Auge, daß Steensgaard Algorithmus deutlich langsamer ist, obwohl dieser für C linearer und Andersens kubischen Zeitaufwand haben. Das liegt an der Iteration über die bedingten Constraints, die länger dauert, je schlechter die bisherigen Ergebnisse der Analyse sind.

Die Andersen-Varianten mit direkter Speicherung der Points-To-Informationen und mit Traversieren des Points-To-Graphen sind für die kleinen Beispiele nahezu gleich schnell, für die größere Beispiele wird das Traversieren des Graphen langsamer. Hier liegt der wesentliche Unterschied im Speicherbedarf.

Die Variante, die während der Berechnung im Points-To-Graphen kollabiert, ist allerdings in jedem Fall langsamer. Dies liegt daran, daß nach unserem Constraint-System extrem wenig Zyklen in den Points-To-Graphen existieren, so daß der Gewinn durch das Kollabieren den Aufwand der Suche nicht aufwiegt. In der Praxis wird sich diese Variante nicht lohnen, da die Zyklen mit Hilfe von stark verbundenen Komponenten [14] schneller gefunden werden können.

[4]Alle Beispiele wurden auf einer Sun Enterprise 450 errechnet

	LOC	PTR	Gewinn	HashSet	BitSet
JLex	7823	7047	10.74%	302.99s	274.36s
Mars	2903	4271	8.56%	99.24s	96.99s
graph	7045	8418	6.95%	292.80s	291.27s
hanoi	4958	7471	14.68%	141.89s	137.37s
jEdit	11862	18599	13.03%	2148.02s	2069.66s
jas	5436	12590	0.61%	925.49s	990.97s

Tabelle 2: Beispiele für Steensgaard Algorithmus

Variant	PTR	Gewinn	Andersen
Normal	7455	44.62%	46.43s
Fields/Objects	6325	35.60%	27.50s
Reached Methods	3767	19.97%	33.77s
Casts/Methods	7492	44.18%	80.10s

Tabelle 3: Variationen für Hanoi Beispiel

6.3 Variationen

Unser Algorithmus hat verschiedene Möglichkeiten mit denen Genauigkeit und Laufzeit beeinflußt werden können. Tabelle 3 zeigt, wie sich diese Möglichkeiten auf das Hanoi Beispiel auswirken[5].

Die Variantionen unterscheiden sich dabei wie folgt:

- Fields/Objects
  Data-Members werden nicht mehr pro Objekt individuell behandelt, sondern nur noch pro Klasse, d.h. es wird immer die Regel für statische Data-Members angewendet. Dabei wird die Genauigkeit geringer. Da weniger Constraints auszuwerten sind, wird die Analyse auch schneller. Für größere Programme erwarten wir einen umgekehrten Effekt. Wie bei Steensgards Algorithmus wird die geringere Genauigkeit die Iteration verlangsamen.
- Reached Methods
  Ausgehen von den `main`-Methoden werden nur die Methoden zur Analyse hinzugefügt, von denen bekannt ist, daß sie wirklich aufgerufen werden. Für das Hanoi-Beispiel reduziert sich dabei der Points-To-Graph auf die Hälfte seiner ursprünglichen Größe, damit verbunden ist eine Laufzeit-Verkürzung. Es werden jedoch für weniger Pointer Points-To-Mengen berechnet, so daß die Genauigkeiten nicht vergleichbar sind.
- Casts/Methods
  Bei dieser Variation werden Typcasts nicht durch Constraints behandelt, sondern für Casts künstliche Methoden generiert, die den gleichen Effekt haben, aber anstelle eines eingenen Constraints das für dynamisch gebundene Methodenaufrufe verwenden. Dies führt allerdings dazu, daß es pro Typ einen Pointer gibt, der auf alle Objekte dieses Typs zeigt, die gecastet werden und somit eine zusätzliche Ungenauigkeit einführt. Dies könnte durch eine Kontext-sensitive Analyse verhindert werden, in unserem Fall war dies mit einem speziellen Constraint leichter realisierbar.

7 Ähnliche Arbeiten

Rountev/Milanova/Ryder [15] haben unabhängig von uns Andersens Algorithmus auf Java angewandt und als Mengen-Inklusions-System beschrieben. Ihre Arbeit ist unserer sehr ähnlich, ist

[5]Die Berechnungen wurden auf einem PC/Athlon-800 durchgeführt. Dabei wurde eine andere Version des JDK als bei den vorherigen Beispiel verwendet, was die abweichenden Zahlen für "normal" erklärt

jedoch auf Andersens Algorithmus beschränkt und hat keine Approximation für nicht-analysierten Code und die Effekte der Reflektion-API. Ihre Implementation benutzt Soot [6] als Frontend und BANE [1] als Constraint-Solver. Da BANE zahlreiche Optimierungen [8, 21] enthält, mag ihre Implementation gerade im Hinblick auf Speicherbedarf unserer überlegen sein. Rountev et al. beschreiben nicht, wie genau sie mit Type-Casts und Arrayinhalten umgehen, so daß es möglich ist, daß diese nicht mit der gleichen Genauigkeit wie bei uns behandelt werden, wodurch ihre Points-To-Graphen mehr Zyklen enthalten und die Optimierungen in diesem Gebiet besser greifen, dafür aber ungenauere Ergebnisse liefern. Hier sind noch genauere Vergleiche hinsichtlich der Präzision notwendig.

Für C gibt es zahlreiche Points-To-Algorithmen [2, 20, 16, 11, 12, 7, 13, 9], mit verschiedenen Verhältnissen von Genauigkeit zu Aufwand.

Ähnliche Techniken kommen bei der Escape-Analysis[3, 5, 22] zur Anwendung, die untersucht, ob Objekte den Gültigkeitsbereich ihrer Definition verlassen. Dabei ist Diese Analyse mehr auf einzelne Methoden als auf ganze Programme ausgerichtet.

8 Zusammenfassung und Zukünftige Arbeiten

Wir haben einen Framework präsentiert, der es möglich macht, eine Klasse von bisher auf C angewandten Algorithmen zur Points-To-Analyse auf Java zu übertragen. Wir haben dazu gezeigt, wie aus Java-Bytecode ein Constraint-System, das Points-To-Mengen berechnet, erzeugt werden kann. Mit einer Implementierung haben wir verschiedene Java-Programme damit analysiert und ausgewertet.

Die analysierten Programme haben dabei unsere Vermutungen bestätigt, daß es nicht sinnvoll ist, unser Constraint-System mit Steensgaards Algorithmus einzusetzen; es scheint zweifelhaft, ob dieser Algorithmus überhaupt für Java sinnvoll eingesetzt werden kann, ohne nur eine geringe Verbesserung gegenüber einer naiven Points-To-Analyse zu liefern.

Dabei bietet unser Constraint-System noch Möglichkeiten zur Verbesserung.

Der Code einer Methode kann in Single-Static-Assignment Form transformiert werden, dadurch wird aus der Fluß-insensitiven Analyse teilweise eine Fluß-sensitive (vgl. [10]). Für Fluß-sensitive Verfahren werden jedoch Threads in Java noch extra zu berücksichtigen sein.

Unser Constraint-System sollte sich mit Algorithmen aus [12], [7] oder [4] zu einer Context-sensitiven Analyse ausbauen lassen. Dies wird die Genauigkeit insbesonderen in Hinblick auf unsere nachfolgende Analyse [18, 19, 17] erhöhen, da es dann möglich ist, Objekte nicht nur nach dem Ort ihrer Erzeugung zu unterscheiden.

Um die Performance zu verbessern und dem Scale-Up entgegen zu treten, scheint es sinnvoll zu sein, Points-To Graphen auf Methodenebene zu erstellen und diese zu optimieren. Im Rahmen von Escape-Analysis[3, 5, 22] ist ermittelt worden, daß viele Java-Objekte die Methode, in der sie erzeugt wurden, nie verlassen. Es sollte möglich sein, Constraints für lokale Pointer und Objekte auf Methoden-Ebene zu behandeln und damit den globalen Points-To-Graphen zu verkleinern, um dadurch die Analyse schneller und weniger Speicheraufwendig zu machen. In [13] wird dies für C gezeigt, es sollte sich auf Java übertragen lassen.

Literatur

[1] A. Aiken, M. Faehndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. *Lecture Notes in Computer Science*, 1473:78–??, 1998.

[6] `http://www.sable.mcgill.ca`

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[3] B. Blanchet. Escape analysis for object oriented languages. application to Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–34, 1999.

[4] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 133–146, New York, NY, USA, 1999. ACM Press.

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. *ACM SIGPLAN Notices*, 34(10):1–19, Oct. 1999.

[6] M. Das. Unification-base pointer analysis with directional assignments. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programmming Design and Implementation (PLDI)*, pages 35–46, Vancouver, Canada, June 2000.

[7] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256. ACM, ACM, June 1994.

[8] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998. *ACM SIGPLAN Notices* 33(6).

[9] J. S. Foster, M. Fähndrich, and A. Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical Report CSD-97-964, University of California, Berkeley, Aug. 5, 1997.

[10] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 97–105, Montreal, Canada, 17–19 June 1998.

[11] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[12] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1992.

[13] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In O. Nierstrasz and M. Lemoine, editors, *Proceedings of the 7th European Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engeneering*, volume 24.6 of *Software Engineering Notes (SEN)*, pages 199–215, N. Y., Sept. 6–10 1999. ACM Press.

[14] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programmming Design and Implementation (PLDI)*, pages 47–56, Vancouver, Canada, June 2000.

[15] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated inclusion constraints. Technical Report DCS-TR-417, Department of Computer Science, Rutgers University, July 2000.

[16] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In ACM, editor, *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Paris, France, 15–17 January 1997*, pages 1–14, New York, NY, USA, 1997. ACM Press.

[17] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*. to appear.

[18] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering: FSE-6*, pages 99–110. ACM Press, 1998.

[19] G. Snelting and F. Tip. Reengineering of class hierarchies using concept analysis. Technical Report MIP-9910, Universität Passau, Fakultät für Mathematik und Informatik, December 1999.

[20] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, FL, January 1996.

[21] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95, Boston, Massachusetts, Jan. 19–21, 2000.

[22] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, Oct. 1999.