Lokalitätsoptimierung durch statische Typanalyse in JavaParty

Diplomarbeit

Bernhard Haumacher

Januar 1998

Institut für Programmstrukturen und Datenorganisation,
Universität Karlsruhe

Betreuer: Dr. Michael Philippsen

Ich erkläre hiermit eidesstattlich, daß ich diese Diplomarbeit selbständig und ohne unzulässige Hilfe angefertigt habe.

Die verwendeten Quellen sind im Literaturverzeichnis vollständig angegeben

Karlsruhe, den 31. Januar 1998

Bernhard Haumacher

Zusammenfassung

JavaParty ist ein Cluster-Computing-Ansatz, bei dem mehrere virtuelle Java-Maschinen miteinander kooperieren, die auf unterschiedlichen Arbeitsstationen in einem Rechnernetzwerk ausgeführt werden. Ein System mit verteiltem Adreßraum kann nur dann Resultate erzielen, die mit einem symmetrischen Multiprozessorrechner vergleichbar sind, wenn Aktivitäten in unterschiedliche Adreßräume separiert sind und die Objekte des Programms eine gute Lokalität aufweisen. Ziel der Arbeit ist es, eine statische Analyse zu entwickeln, die eine Verteilungsstrategie für Objekte und Aktivitäten ohne Rückgriff auf Laufzeitinformationen generiert.

Wegen des hohen Maßes an Polymorphie in objektorientierten Sprachen erhält man nur durch Typinferenz besonders genaue Kontrollfluß-informationen, die zur Objektverteilung notwendig sind. Die Verbindung von Typinferenz und Lokalitätsoptimierung ist ein neuer vielver-sprechender Ansatz, um durch Vermeidung von expliziter Objekt-plazierung die Abstraktion eines virtuell gemeinsamen Adreßraums zu schaffen.

Mit der Implementation eines Typinferenzalgorithmus für Java und dessen Erweiterung auf Fragestellungen, die speziell bei der Verteilungsanalyse auftreten, konnte die Effektivität dieser Vorgehensweise gezeigt werden.

Inhaltsverzeichnis

1 Einleitung

Der Boom, den die Programmiersprache Java in den letzten Jahren erfahren hat, beruht nicht allein auf der Tatsache, daß Java mit seiner virtuellen Maschine eine systemunabhängige Plattform zur Verfügung stellt, mit deren Hilfe Programme über das Internet geladen und direkt ausgeführt werden können. Zur breiten Akzeptanz beigetragen hat auch die Eigenschaft von Java, Parallelität in Programmen einfach ausdrückbar zu machen und direkt auf Sprachebene zu unterstützen. Diese Eigenschaft ist neben der Verfügbarkeit von Bibliotheken zur Kommunikation zwischen mehreren vernetzten Rechnern eine wichtige Voraussetzung für die Entwicklung moderner Anwendungen.

Java eröffnet dem Programmierer durch Threads die Möglichkeit, Nebenläufigkeit in der Problemstellung in deren Lösung zu übernehmen, oder gezielt nebenläufige Problemlösungen zu entwickeln. Ein Geschwindigkeitsvorteil bei der Programmausführung gegenüber einer sequentiellen Lösung ist allerdings nur dann zu erwarten, wenn die virtuelle Maschine auf einem Rechner mit mehreren Zentralprozessoren zur Ausführung kommt und die Implementation der virtuellen Maschine die Verwendung mehrerer Prozessoren auch unterstützt. Andernfalls dient die Verwendung von mehreren Aktivitätssträngen in einem Java-Programm ausschließlich der Klarheit der Lösung, da dem Programmierer die Aufgabe abgenommen wird, ein inhärent paralleles Problem für einen einzelnen Prozessor explizit zu serialisieren.

JavaParty erweitert Java um eine verteilte Laufzeitumgebung, in der sich mehrere virtuelle Maschinen an der parallelen Lösung eines Problems beteiligen. Die einzelnen virtuellen Maschinen werden auf unterschiedlichen Arbeitsstationen ausgeführt, die untereinander durch ein Netzwerk gekoppelt sind. Jede einzelne virtuelle Maschine hat dabei ihren eigenen Adreßraum, kann aber Methoden von Objekten auf anderen virtuellen Maschinen über entfernten Methodenaufruf (*remote method invocation*, kurz RMI) ausführen. Hier kann durch die auf Thread-Ebene ausgedrückte Parallelität eines Programms auch dann eine Geschwindigkeitssteigerung erzielt werden, wenn die Zielplattform statt eines Mehrprozessorrechners mit gemeinsamem Adreßraum ein Workstation-Cluster ist. JavaParty hält die Entwicklung von Klassen, deren Objekte sich in der verteilten Umgebung befinden und deren Methoden von entfernten virtuellen Maschinen aus aufgerufen werden können, für den Programmierer transparent. Diese Klassen heißen Fernklassen und werden im Unterschied zu normalen Java-Klassen mit dem zusätzlichen Schlüsselwort *remote* deklariert. Instanzen von Fernklassen heißen entfernte Objekte, unabhängig davon auf welcher virtuellen Maschine sie angelegt sind. Bei der Übersetzung einer Fernklasse generiert der JavaParty-Übersetzer den dabei zusätzlich notwendigen Programmcode. Nach der Transformation der Fernklassen werden entfernte Methodenaufrufe mittels des zur Java-Klassenbibliothek gehörenden RMI-Pakets durchgeführt. Die bei Verwendung von RMI notwendige Fehlerbehandlung wird vor dem Programmierer verborgen und intern abgewickelt. Die Java-Objektsemantik wird für Fernklassen weitgehend beibehalten, so daß der Programmierer mit entfernten Objekten genauso umgehen kann wie mit normalen Java-Objekten.

Durch die Verteilung der Objekte und Aktivitätsstränge über die beteiligten virtuellen Maschinen alleine läßt sich allerdings noch keine Geschwindigkeitssteigerung erreichen. Nur geschickte Gruppierung der Objekte und der sie betreffenden Aktivitäten macht die in der Problemlösung steckende Parallelität auch nutzbar. Nur wenn Objekte, deren Methoden in unterschiedlichen Aktivitätssträngen abgearbeitet werden, auf unterschiedlichen Prozessorknoten liegen, besteht bei diesen Methoden überhaupt die Möglichkeit, sie echt parallel auszuführen. Objekte mit häufiger und teurer Kommunikation sollen dagegen auf demselben Prozessorknoten angelegt werden. Bei Aktivitätssträngen, die häufig Methoden derselben Objekte aufrufen, läßt sich durch Verteilung der Aktivitätsstränge auf unterschiedliche Prozessoren nur wenig Parallelität nutzbar machen, da die Plazierung eines Objekts immer den ausführenden Prozessor seiner Methoden bestimmt. Die Methode ein und desselben Objekts läßt sich also nie gleichzeitig auf zwei unterschiedlichen Prozessoren ausführen. Wenn man solche Aktivitätsstränge auf unterschiedlichen Prozessoren anlegt, kann sich die Ausführungszeit aufgrund teurer Kommunikation gegenüber der Ausführung auf einem einzelnen Prozessor sogar erhöhen.

JavaParty stellt bisher nur ein Konzept zur Verfügung, mit dem Klassen entwickelt werden können, deren Instanzen auf unterschiedlichen Prozessorknoten einer verteilten Umgebung existieren können und mit denen der Benutzer trotzdem fast genauso umgehen kann, als wären es normale Java-Klassen. Für die Verteilung der einzelnen Objekte und Aktivitäten auf die Prozessorknoten ist der Programmierer aber selbst verantwortlich. Eine solche manuelle Objektverteilung hat einige schwerwiegende Nachteile: Zum einen ist sie abhängig von der Zielplattform, für die das Programm übersetzt wird. Entscheidende Größen sind die zur Verfügung stehenden Prozessoren und der typische Kommunikations-Overhead, der bei einem entfernten Methodenaufruf anfällt. Die Verteilungsstrategie muß also für jede Zielplattform neu erarbeitet und das Programm entsprechend angepaßt werden. Zum anderen läßt es sich schwer vermeiden, daß der Programmcode durch eine Vielzahl von Anweisungen zur Objektplazierung „verschmutzt" wird. Gibt man dem Laufzeitsystem keine anders lautenden Instruktionen, verteilt es neu anzulegende Objekte nach einer zwar vom Programmierer wählbaren aber sehr grobkörnigen Strategie. Eine solche Strategie setzt sich z.B. aus Regeln der Form zusammen „alle Instanzen der Klasse C werden auf dem Prozessorknoten p angelegt".

Der Programmcode soll ausschließlich die Lösung der gestellten Aufgabe beschreiben. Die Korrektheit der Lösung ist aber vollkommen unabhängig von der Anordnung der Objekte, die an dieser Lösung beteiligt sind. Man möchte weder bei jeder einzelnen Objekterzeugung angeben, auf welchem Prozessorknoten dieses Objekt erzeugt werden soll, noch soll sich der Programmierer darum kümmern müssen, ob es bei einem Methodenaufruf günstiger ist, zuerst das Objekt zu migrieren und dann die Methode aufzurufen, oder den Aufruf direkt durchzuführen. Diese Arbeit beschäftigt sich damit, eine Verteilungsstrategie für Objekte und Aktivitäten automatisch zu generieren. Die Generierung der Verteilungsstrategie soll ohne Rückgriff auf Laufzeitinformationen allein aufgrund statischer Analyse des Programmtexts geschehen. Dadurch wird der Programmierer von der Aufgabe entbunden, sich neben der Problemlösung auch um die passende Objektverteilung zu kümmern. Bei einer Objektverteilung von Hand müßte der Programmierer die beim Methodenaufruf durch JavaParty verdeckten Adreßraumgrenzen wieder in seine Überlegungen einbeziehen. Wird die Verteilungsstrategie jedoch automatisch generiert, kann der verteilte Adreßraum während des gesamten Entwicklungsprozesses als virtuell gemeinsamer Adreßraum betrachtet werden.

Bei gemeinsamem Adreßraum kann die Zuordnung der Aktivitäten zu mehreren zur Verfügung stehenden Prozessoren durch das Betriebsystem vorgenommen werden. Sobald mehrere Aktivitäten zur Ausführung bereit sind, können diese auch parallel abgearbeitet werden. In der verteilten Umgebung kann ein Prozessor eine Aktivität nur dann ausführen, wenn sich das Objekt, dessen Methode abgearbeitet werden soll, in seinem Adreßraum befindet. Da die Plazierungsentscheidung aber vor der Erzeugung eines Objektes und damit auch vor dessen Benutzung fallen muß, setzt eine gute Plazierungsentscheidung globales Wissen über Aufrufhäufigkeiten und Beziehungen zwischen einzelnen Aktivitäten und Instanzen von Klassen des Programms voraus. Nur wenn vor der Erzeugung eines Objektes bekannt ist, welche Aktivität dieses Objekt hauptsächlich nutzen wird, können die Anzahl entfernter Methodenaufrufe minimiert und teure Objektmigration vermieden werden. In objektorientierten Programmen ist die Gewinnung solchen Wissens aber besonders schwierig, da starke Abhängigkeiten zwischen Kontroll- und Datenfluß bestehen. So spezifiziert ein Methodenaufruf nicht die konkret aufgerufene Funktion, sondern nur eine Menge möglicher Aufrufe. Bei einem Methodenaufruf *x.f(…)* sichert der deklarierte abstrakte Typ der Variable *x* nur das Vorhandensein einer Implementation der Methode *f* für jeden Aufruf zur Laufzeit zu. Vor jedem konkreten Aufruf zur Laufzeit findet die Auswahl der Implementation von *f* mittels der Klasse des Objekts statt, welches zu diesem Zeitpunkt in der Variablen *x* gespeichert ist. Die Verschiebung der Implementationsauswahl auf die Laufzeit ermöglicht Polymorphie und trägt dadurch zur besseren Wiederverwendbarkeit objektorientierten Codes bei. Die Vorhersage des globalen Kontrollflusses eines Programms wird allerdings erschwert und führt ohne tiefergreifende Analyse zu wenig aussagekräftigen Ergebnissen.

Will man zur Übersetzungszeit verläßliche Aussagen über den Kontrollfluß eines objektorientierten Programms treffen, muß man, wo immer möglich, die störende Polymorphie auflösen, um einen statischen interprozeduralen Kontrollfluß zu erhalten. An Methodenaufrufstellen *x.f(...)* benötigt man

daher eine möglichst präzise Vorhersage darüber, welche Objekte zur Laufzeit in der Variablen *x* gespeichert sein können. Von den Klassen dieser Objekte hängt der interprozedurale Kontrollfluß ab, der an diesem Methodenaufruf auftreten kann. Ein Typinferenzalgorithmus versucht eine möglichst genaue Vorhersage über die Objekte zu machen, die zur Laufzeit in einer Variable gespeichert sein können. Bei polymorph verwendetem Code geschieht das durch Zerlegung in monomorphe Versionen. In dieser Arbeit kommt daher ein Typinferenzalgorithmus zum Einsatz, um eine möglichst genaue statische Approximation an den interprozeduralen Kontrollfluß des Programms zu erzielen. Die Typinferenz wird in dieser Arbeit dahingehend erweitert, daß sich die Ergebnisse für eine darauf aufbauende Lokalitätsoptimierung einsetzen lassen.

Die Untersuchung des Aufrufgraphen der Aktivitäten des Programms ermöglicht anschließend eine Zuordnung von Objekten zu der Aktivität, welche die Objekte hauptsächlich benutzt. Diese Zuordnung beruht auf Schätzungen von Aufrufhäufigkeiten, Komplexitäten von Methoden und Strukturgrößen. Die aus den Zuordnungen gewonnenen Plazierungsinformationen dienen dazu, schon bei der Erzeugung eines Objekts die virtuelle Maschine gezielt auszuwählen, auf der das Objekt angelegt werden soll. Nur eine zur Übersetzungszeit durchgeführte statische Lokalitätsoptimierung kann diese Information liefern, da sich aus der Analyse des Programmtext Rückschlüsse auf die zukünftige Verwendung eines an einer bestimmten Stelle erzeugten Objekts ziehen lassen. Eine dynamisch zur Laufzeit durchgeführte Lokalitätsoptimierung kann erst nach der Erzeugung eines Objekts während dessen Verwendung feststellen, auf welcher virtuellen Maschine es am besten hätte angelegt werden sollen. Eine daraufhin durchzuführende Objektmigration auf diese Maschine kostet dann unnötig Zeit, die durch statische Analyse eingespart werden kann.

Zur Implementation der berechneten Verteilungsstrategie ist eine Programmtransformation nötig, welche die Verteilungshinweise für jede Objekterzeugung in Anweisungen an das Laufzeitsystem umsetzt. Der Programmierer kann dann Fernklassen wie im lokalen Fall definieren und verwenden. Der Übersetzter übernimmt die Transformation der Klassen für die Verwendung im verteilten Laufzeitsystem und implementiert gleichzeitig eine geeignete Verteilungsstrategie.

1.1 Hintergrund

Bevor ein Überblick über die vorliegende Arbeit gegeben wird, soll kurz auf verwandte Techniken hingewiesen und eine Einordnung benutzter Verfahren in die Literatur vorgenommen werden. Diese Arbeit untersucht Möglichkeiten statischer Typanalyse als Teil einer Lokalitätsoptimierung für verteiltes Rechnen. Typanalyse ist dabei ein sehr allgemeiner Begriff und soll hier näher spezifiziert werden. Bei dem Begriff Typanalyse unterscheidet man Typprüfung, Typrekonstruktion und Typinferenz. Die Begriffe Typrekonstruktion und Typinferenz werden in der Literatur oft synonym gebraucht, sie sollen hier aber verwendet werden, um zwei von der Intention her prinzipiell unterschiedliche Verfahren zu unterscheiden.

- Typprüfung ist Teil der Semantischen Analyse eines Compilers. Bei der Übersetzung einer statisch typisierten Sprache (wie C, Pascal, Java) stellt der Compiler während der Typprüfung sicher, daß Operatoren immer nur mit Argumenten kompatiblen Typs verwendet werden. Die statische Typisierung einer objektorientierten Sprache sichert also schon zur Übersetzungszeit u.a. zu, daß für alle Methodenaufrufe zur Laufzeit eine passende Implementation der Methode gefunden werden kann und daß Zugriffe auf Instanzvariablen nur mit solchen Objekten durchgeführt werden, die eine entsprechende Instanzvariable besitzen. Die Typprüfung gehört zu den Standardbestandteilen eines Übersetzers und läßt sich durch attributierte Grammatiken beschreiben [Wai84].
- Viele funktionale Sprachen (ML, Haskell, Gofer) verlangen vom Programmierer nicht, daß Typen von Bezeichnern des Programms explizit deklariert werden, da für das Typsystem dieser Sprachen ein Typrekonstruktionsalgorithmus existiert. Das Verfahren der Typrekonstruktion beruht auf Unifikation von Typen und berechnet für jeden Ausdruck seinen allgemeinsten Typ, sofern

dieser existiert. Das wohl bekannteste derartige Typsystem ist das Hindley/Milner System, welches in der Sprache ML zum Einsatz kommt [Mil78].

Deklarierte und durch Typrekonstruktion ermittelte Typen sind immer abstrakte Typen. Programmiersprachen verwenden abstrakte Typen, um die Anwendung von Operatoren auf Argumente, die nicht im Definitionsbereich des Operators liegen, schon zur Übersetzungszeit erkennen zu können und Typüberprüfungen zur Laufzeit, die eventuell zu Laufzeittypfehlern führen, nach Möglichkeit zu vermeiden. Typprüfung bzw. Typrekonstruktion sind demnach Teil der Programmverifikation, der durch den Übersetzer vollautomatisch durchgeführt werden kann.

- Im Gegensatz zur Typrekonstruktion, die den allgemeinsten (abstrakten) Typ bestimmt, bemüht man sich bei der Typinferenz konkrete Typen zu berechnen. Der konkrete Typ einer Variablen ist die Menge derjenigen Objekte, welche während aller denkbaren Programmabläufe in dieser Variablen gespeichert sein können.

In dieser Arbeit kommt speziell ein Typinferenzalgorithmus für objektorientierte Sprachen zum Einsatz, um das statische Programmverständnis zu verbessern und darauf aufbauend eine Lokalitätsoptimierung durchführen zu können. Im allgemeinen ist es nur möglich, eine Näherung für die konkreten Typen anzugeben, die den konkreten Typ als Teilmenge enthält. Die in der Literatur vorgeschlagenen Algorithmen zur Typinferenz bei objektorientierten Programmiersprachen unterscheiden sich dabei in der erreichbaren Genauigkeit [Pal91] [Oxh92] [Pal92] [Age95] [Ple96]. [Age94] ordnet die bekannten Algorithmen in eine Reihenfolge zunehmender Genauigkeit ein.

Die bei der Typinferenz gewonnenen Informationen sind allgemein bei der Optimierung objektorientierter Sprachen nützlich. Aus dem konkreten Typ einer Variablen, mit der ein Methodenaufruf stattfindet, läßt sich ablesen, ob für alle in dieser Variablen möglichen Objekte zur Laufzeit die selbe Implementierung der Methode ausgewählt wird. Ist dies der Fall, kann die Methodenauswahl schon während der Übersetzung durchgeführt (Vermeidung der dynamischen Methodenauflösung), oder der Code der Methode an Stelle des Aufrufs eingesetzt werden (Methoden Einbettung). [Dol97] schlägt darüber hinaus das automatische Einbetten ganzer Klassen in andere Klassen vor. Dies vermeidet die in objektorientierten Sprachen übliche Allokation vieler sehr kleiner Objekte.

In dieser Arbeit wird der Typinferenzalgorithmus aus [Ple96] verwendet. Die erreichte Trennschärfe reicht allerdings nicht aus, um die Ergebnisse direkt für die Verteilungsanalyse einsetzen zu können. Der Algorithmus versucht, durch Duplikation polymorph verwendeter Teile des Programms eine exaktere Typisierung zu erreichen und dadurch Kontrollflußunschärfen aufzulösen. Für die Verteilungsanalyse ist aber neben einer genauen Vorhersage des Kontrollflusses auch wichtig, die Verwendung von verschiedenen Instanzen der selben Klasse in unterschiedlichen Aktivitäten zu erkennen. Durch die Erweiterung des Typinferenzalgorithmus wird diese Unterscheidung auch dann möglich, wenn die Instanzen der Klasse in beiden Aktivitäten monomorph verwendet werden und der ursprüngliche Algorithmus sie daher nicht weiter unterteilen würde.

1.2 Überblick

Kapitel 2 stellt zuerst die Prinzipien des einfachen objektorientierten Typinferenzalgorithmus aus [Pal91] vor und motiviert anhand von Beispielen, in denen verschiedene Formen von Polymorphie vorkommen, warum die mit diesem Algorithmus erzielten Ergebnisse für eine Verteilungsanalyse nicht ausreichen. Die größte Genauigkeit der in 1.1 erwähnten Typinferenzalgorithmen erreicht der Algorithmus aus [Ple96]. Die Beispiele aus Kapitel 2 lassen erkennen, daß mindestens diese Genauigkeit für eine Verteilungsanalyse notwendig ist, da dieser Algorithmus sowohl polymorph verwendete Methoden als auch Datenpolymorphie adäquat behandelt.

Der Rest von Kapitel 2 liefert eine formale Darstellung dieses Typinferenzalgorithmus, die so in [Ple96] nicht enthalten ist. Die Verteilungsanalyse kann nicht losgelöst von der Typinferenz betrachtet werden, da sowohl Verteilungsanalyse als auch Typinferenz auf den selben Prinzipien beruhen. So werden während der Typinferenz zur Auflösung polymorpher Strukturen Objekte

ausgehend von den Stellen ihrer Erzeugung in Klassenkonturen unterteilt. Diese Klassenkonturen bilden dann die Einheiten, die für die Objektverteilung zur Zuweisung an unterschiedliche virtuelle Maschinen zur Verfügung stehen. Die Erweiterungen des Typinferenzalgorithmus in Kapitel 6 zielen dann darauf ab, die erreichte Genauigkeit des Algorithmus durch weitergehende Unterteilung dieser Klassenkonturen zu erhöhen, um damit die Ergebnisse der Typinferenz direkt für die Verteilungsanalyse einsetzen zu können. Außerdem wird bei der Formalisierung der Typinferenz auf spezifische Eigenschaften der Sprache Java eingegangen.

Nachdem in Kapitel 3 die Grundzüge von JavaParty und die daraus resultierenden Strategien für die Objektverteilung vorgestellt wurden, wird in Kapitel 4 der eigentlich zweite Teil der Verteilungsanalyse beschrieben. Die Grundlage für diesen Schritt bildet das Programm, das zum Zwecke der schärferen Typisierung durch Transformation aus dem ursprünglichen Programm hervorgegangen ist. Diese während der Typinferenz durchgeführten Transformationen extrahieren aus dem Ausgangsprogramm erst die für die Verteilung zur Verfügung stehenden Objekte, was besonders bei den Erweiterungen in Kapitel 6 deutlich wird. Daher kann die Typinferenz mit ihren Erweiterungen schon als erster Teil der Verteilungsanalyse betrachtet werden. Der zweite Teil der Verteilungsanalyse in Kapitel 4 schlägt Abschätzungen vor, die Verwendungen von Instanzen der durch die Typinferenz erzeugten Klassenkonturen des transformierten Programms in unterschiedlichen Aktivitäten widerspiegeln. Diese Abschätzungen werden schließlich zu expliziten Verteilungshinweisen verdichtet.

Da die während der Typinferenz zur schärferen Typisierung eingeführten Konzepte wie Methoden- und Klassenkonturen kein legales Java-Programm mehr ergeben, muß sich an Typinferenz und Verteilungsanalyse eine Programmtransformation anschließen, welche die aus der Typinferenz hervorgegangenen Strukturen verknüpft mit den Verteilungshinweisen wieder in ein legales Java-Programm zurückverwandelt. Diese Programmtransformation wird in Kapitel 5 vorgestellt. Sie ist einerseits eng mit den während der Typinferenz eingeführten Konzepten verknüpft, muß andererseits aber die Eigenschaften von Java, insbesondere das statische Typsystem, speziell berücksichtigen.

Kapitel 8 diskutiert die erzielten Resultate, während Kapitel 9 einen Ausblick auf noch mögliche und notwendige Erweiterungen gibt.

2 Der Typinferenzalgorithmus

2.1 Das Typsystem von Java

Java ist eine objektorientierte Sprache. Alle Laufzeitobjekte sind Instanzen einer Klasse oder Werte eines Basistyps (*boolean, byte, short, char, int, long, float, double*). Mit Ausnahme der Klasse *Object*, der Wurzel der Klassenhierarchie, ist jede Java-Klasse Unterklasse genau einer anderen Klasse. Die Einschränkung auf einfache Erblichkeit kann durch die Definition von Schnittstellen ausgeglichen werden. In einer Schnittstelle sind nur Methodensignaturen vereinbart. Implementiert eine Klasse eine Schnittstelle, stellt sie eine Implementation für die in der Schnittstelle deklarierten Methoden bereit. Instanzen dieser Klasse können dann an allen Stellen verwendet werden, an denen ein Objekt mit dieser Schnittstelle erwartet wird.

Das Typsystem von Java ist statisch. Zur Übersetzungszeit läßt sich nach Überprüfung der deklarierten Typen zusichern, daß für jeden textuellen Methodenaufruf in der Klasse des betroffenen Laufzeitobjekts eine passende Implementation für die gerufene Methode zur Verfügung steht. Die in Deklarationen verwendeten Typen sind Basistypen, Klassen- oder Schnittstellentypen. Eine Variable des Klassentyps *A* speichert zur Laufzeit Instanzen der Klasse *A* oder einer Unterklasse *B* von *A*. Variablen vom Schnittstellentyp *I* können zur Laufzeit Instanzen der Klassen *Ci* oder deren Unterklassen aufnehmen, welche die Schnittstelle *I* implementieren. Da sich die aufgerufene Methode nach der Klasse des Laufzeitobjekts und nicht nach dem definierten Typ richtet, ist es notwendig, die deklarierten Typen und die Klassen der Objekte, die sich zur Laufzeit in einer Variable befinden können, zu unterscheiden. Mit *konkretem Typ* einer Variablen soll im folgenden die Menge der Klassen der Laufzeitobjekte bezeichnet werden, die diese Variable zur Laufzeit aufnehmen kann. Eine erste Approximation des konkreten Typs läßt sich aus dem definierten Typ und der Klassenhierarchie ableiten. Ist der definierte Typ die Klasse *A*, umfaßt der konkrete Typ höchstens die Klasse *A* mit allen ihren Unterklassen. Handelt es sich beim definierten Typ um einen Schnittstellentyp *I*, enthält der konkrete Typ höchstens alle Klassen C_i, die die Schnittstelle *I* implementieren mit allen deren Unterklassen. Eine Variable eines Basistyps enthält zur Laufzeit immer einen Wert ihres deklarierten Typs.

Wie man sich leicht vorstellen kann, liefert diese Approximation des konkreten Typs keine sinnvollen Werte zur Bestimmung des interprozeduralen Kontrollflusses eines Programms. Für jeden Methodenaufruf ist man an einer möglichst eng begrenzten Menge von Methoden interessiert, die zur Laufzeit an dieser Stelle aufgerufen werden können. Ist die Menge dieser Aufrufziele einelementig, so heißt der Kontrollfluß des Programms an diesem Methodenaufruf statisch. Bei Methodenaufrufen für Variablen eines Schnittstellentyps *I* sind alle Methoden mit passender Signatur aller Klassen, welche die Schnittstelle *I* implementieren sowie deren Unterklassen, mögliche Aufrufziele. Diese Menge von Methoden ist in der Regel keine präzise Vorhersage für die zur Laufzeit an dieser Stelle tatsächlich durchgeführten Methodenaufrufe.

2.2 Der einfache Typinferenzalgorithmus

Aus dem abstrakten Typ eines Ausdrucks läßt sich, wie in 2.1 beschrieben, eine erste Approximation seines konkreten Typs angeben. Der in [Pal91] vorgestellte einfache Typinferenzalgorithmus verfeinert obige Approximation.

Jeder Variablen und jedem Programmausdruck *e* wird eine Typvariable T_e zugeordnet, die den konkreten Typ als Ergebnis der Analyse aufnimmt. Anweisungen und Ausdrücke des Programms werden dann in Mengenungleichungen zwischen diesen Typvariablen übersetzt. Die Mengenungleichungen simulieren dabei den Datenfluß während des Programmablaufs zur Laufzeit, sie abstrahieren aber von der Reihenfolge der Anweisungen und vom lokalen Kontrollfluß innerhalb einer Metho-

de. Eine Ungleichung $T_b \supseteq T_a$ bedeutet, daß zur Laufzeit das Ergebnis des Ausdrucks *a* an *b* zugewiesen werden kann.

Die in Abbildung 1 aufgeführten Regeln erzeugen für alle Programmausdrücke eine Typvariable und generieren für alle relevanten Programmkonstrukte die entsprechenden Ungleichungen. Die Lösung des resultierenden Mengenungleichungssystems läßt sich durch Fixpunktiteration beginnend mit leeren Typvariablen als Startwerten berechnen. Eine Ungleichung $T_b \supseteq T_a$ wird dadurch erfüllt, daß die Werte der rechten Seite T_a an die linke Seite T_b weitergereicht werden. Man führt diese Operation solange für alle Ungleichungen durch, bis sich der Inhalt von keiner Typvariablen mehr ändert.

Derartige Mengenungleichungssysteme treten bei Optimierungen im Übersetzerbau häufig auf und deren Lösung ist unter dem Stichwort „Datenflußanalyse“ bekannt. Klassische Anwendungen von Datenflußanalyse sind z.B. die Berechnung von Informationen über „erreichende Definitionen“, „lebendige Variablen“ oder „verfügbare Ausdrücke“. Bei Optimierungen innerhalb einer Funktion setzt man diese Informationen ein, um Codeverschiebungen durchführen zu können oder die mehrfache Berechnung desselben Ausdrucks zu vermeiden [Wai84].

(var)	für Variable x ist T_x zugeordnete Typvariable	x :: T_x
(assign)	für Zuweisung x = e mit x :: T_x und e :: T_e	$T_x \supseteq T_e$
(new)	für new A(p_1,..., p_n) und die zugehörige Konstruktordefinition A(P_1 p_1', ..., P_n p_n') {...} mit p_1' :: T_{p_1}', ..., p_n' :: T_{p_n}' und this :: T_{this_A} innerhalb der Konstruktordefinition	new A(p_1,..., p_n) :: T_{new}, $T_{new} \supseteq \{A\}$, $T_{this_A} \supseteq \{A\}$, $T_{p_1}' \supseteq T_{p_1}$,..., $T_{p_n}' \supseteq T_{p_n}$
(apply)	für x.f(p_1,..., p_n) und die Deklarationen der durch den Aufruf erreichbaren Methoden A f(P_1 p_1', ..., P_n p_n'){...} mit p_1' :: T_{p_1}', ..., p_n' :: T_{p_n}' und this :: T_{this_f}, result :: T_{result_f} innerhalb der Methodendeklaration	x.f(p_1,..., p_n) :: T_{apply}, $T_{this_f} \supseteq T_x$, $T_{p_1}' \supseteq T_{p_1}$,..., $T_{p_n}' \supseteq T_{p_n}$ und $T_{apply} \supseteq T_{result_f}$
(return)	für return e mit e :: T_e innerhalb der Methode f	$T_{result_f} \supseteq T_e$
(coerce)	für (A)e mit e :: T_e	(A)e :: T_c, T_c = {t ∈ T_e | t Unterklasse von A}

Abbildung 1
Regeln zur einfachen Datenflußanalyse

Instanzvariablen werden in Abbildung 1 noch nicht gesondert behandelt. Sie werfen besondere Probleme auf, die aber im weiteren Verlauf ausführlich untersucht werden. Die Regeln aus Abbildung 1 dienen lediglich zur Motivation, um an kleinen Programmbeispielen zu zeigen, daß eine tiefergreifende Analyse notwendig ist, um eine konkrete Typisierung zu berechnen, die zu einem statischen Kontrollfluß führt.

Besonders zu beachten ist in der Regel (apply), daß von dem Methodenaufruf *x.f(...)* mehrere Methoden erreicht werden können. Die Menge der erreichbaren Methodendeklarationen ist von den Klassen in der Typvariablen T_x abhängig. Für jede Klasse $c \in T_x$ ist diejenige Methode *f'* durch den Methodenaufruf erreichbar, die zur Laufzeit angesprungen wird, wenn sich im Parameter *x* eine Instanz der Klasse *c* befindet. Statische Methoden, denen beim Aufruf keine Instanz der Klasse mitübergeben wird, führen beim Aufruf immer zu einem statischen Kontrollfluß. Sie können entweder separat behandelt werden oder man führt, wie weiter unten noch beschrieben wird, zu Analyse-

zwecken eine Variable *voidVar* ein, die bei statischen Methodenaufrufen an Stelle von *x* und $this_f$ verwendet wird. Aufrufe einer statischen Methode *f* werden dann als *voidVar.f()* modelliert.

Die Selektion der Aufrufziele in Abhängigkeit von den Werten im Parameter *x* ist das Analyseäquivalent zur dynamischen Methodenauflösung während der Programmausführung. An dieser Stelle tritt die Abhängigkeit zwischen Daten- und Kontrollfluß deutlich zu Tage. Anhand des Beispielprogramms aus Abbildung 2 sollen die Fälle von Polymorphie erläutert werden, die eine Verfeinerung der Analyse notwendig machen.

Das Programm deklariert zwei Klassen *W1* und *W2* die beide die Schnittstelle *Runnable* implementieren. Instanzen beider Klassen können somit in Variablen des Schnittstellentyps *Runnable* gehalten werden. Die Hauptfunktion *main()* erzeugt zuerst jeweils eine Instanz der Klasse *W1* und *W2* und speichert diese in die Variablen *r* und *s*. Danach werden die in der Schnittstelle *Runnable* deklarierten Methoden *W1.run()* und *W2.run()* für diese beiden Instanzen auf unterschiedliche Arten aufgerufen.

Im ersten Aufruf von *serialize(r, r)* wird zweimal *W1.run()* ausgeführt, im zweiten Aufruf *serialize(s, s)* dagegen zweimal *W2.run()*. Diese beiden Aufrufe von *serialize()* werden zuerst betrachtet, um die Auswirkungen der polymorphen Verwendung der Methode auf die Typisierung bzw. auf das Ergebnis der Kontrollflußanalyse zu veranschaulichen.

Eine andere Form von Polymorphie tritt im weiteren Verlauf des Programms auf. Mit *r* und *s* als Träger der Aktivität werden zwei Threads instanziert und gestartet, die die beiden Methoden *W1.run()* und *W2.run()* parallel ausführen. Die Klasse *java.lang.Thread* ist in Auszügen mit abgedruckt, um zu zeigen, wie der Start der neuen Aktivität abläuft. Dem Konstruktor von *Thread* wird der Träger der Aktivität als Parameter übergeben und in der Instanzvariablen *Thread.r* gespeichert. Die Methode *start()* erzeugt den neuen Aktivitätsstrang, der seine Ausführung mit der Methode *Thread.run()* beginnt. Dort wird die Instanzvariable *Thread.r* wieder gelesen und die *run()* Methode des in ihr gespeicherten Objekts aufgerufen. Die neue Aktivität endet, wenn die Ausführung von *run()* zurückkehrt.

```
class W1 implements Runnable {
  public void run() {…}
}
class W2 implements Runnable {
  public void run() {…}
}

class Thread {
  Runnable r;

  Thread(Runnable r) {
    this.r = r;
  }
  void run() {
    if (r != null) r.run();
  }
  native void start() {
    // ruft run() in einem neuen Thread auf
  }
}
⋮
public class Main {
  static void serialize(
    Runnable r1, Runnable r2
  ) {
    r1.run();  r2.run();
  }
  ⋮
  public static void main(String args[]) {
    Runnable r = new W1();
    Runnable s = new W2();
    serialize(r, r); serialize(s, s);
    (new Thread(r)).start();
    (new Thread(s)).start();
  }
}
```

Abbildung 2
Parametrischer Polymorphismus und Datenpolymorphie

2.3 Parametrischer Polymorphismus

In Abbildung 3 ist die Datenflußanalyse für die beiden Methoden *main()* und *serialize()* durchgeführt. Die konkreten Typen sind als hochgestellte Mengen von Klassen an alle Ausdrücke und Variablen der Funktionen annotiert. Man erkennt, daß die beiden Aufrufe von *serialize()* monomorph sind, daß sich aber die monomorphen konkreten Typen *{W1}* und *{W2}* der beiden Aufrufe zu einem polymorphen Typ *{W1, W2}* in den Parametern der Methode *serialize()* mischen. Diese Form von Polymorphie heißt hier *parametrischer Polymorphismus*.

Die Mehrdeutigkeit im konkreten Typ der Parameter *r1* und *r2* bewirkt, daß an der Stelle des Methodenaufrufs *r1.run()* innerhalb von *serialize()* nur noch sicher ist, daß eine der beiden Metho-

den *W1.run()* oder *W2.run()* aufgerufen wird. Die Information geht verloren, daß der erste Aufruf von *serialize()* ausschließlich die Methode *W1.run()* benutzt und der zweite nur *W2.run()*. In diesem Beispiel fällt der Informationsverlust nicht weiter ins Gewicht, da beide Aufrufe von *serialize()* im selben Aktivitätsstrang durchgeführt werden. Man muß sich aber vor Augen halten, daß die konkrete Typisierung dazu verwendet werden soll, einen statischen Kontrollfluß zu erzeugen, damit Rückschlüsse auf eine gute Objektgruppierung aus den Aufrufhäufigkeiten einzelner Methoden untereinander gezogen werden können.

```
void serialize(
  Runnable r1{W1, W2},
  Runnable r2{W1, W2}
) {
  r1{W1, W2}.run();  r2{W1, W2}.run();
}

void main() {
  Runnable r{W1} = new W1(){W1};
  Runnable s{W2} = new W2(){W2};
  serialize(r{W1}, r{W1});
  serialize(s{W2}, s{W2});
  (new Thread(r{W1})){Thread}.start();
  (new Thread(s{W2})){Thread}.start();
}
```

Abbildung 3
Ergebnis der Datenflußanalyse
Anotierte konkrete Typen

Erfolgten die beiden Aufrufe von *serialize()* aus zwei unterschiedlichen Aktivitätssträngen heraus, würde es durchaus eine Rolle spielen, in welchem Aufruf die Methode *W1.run()* und in welchem die Methode *W2.run()* verwendet wird. Aufgrund der in diesem Beispiel errechneten konkreten Typen ließe sich dann nicht mehr nachvollziehen, welcher Aktivitätsstrang die Methode *W1.run()* und welcher die Methode *W2.run()* durchläuft, und damit welcher dabei die Klasse *W1* und welcher die Klasse *W2* verwendet. Die Information ist über beide Aufrufe verschmiert und läßt sich nicht zur Gewinnung einer Verteilungsstrategie für Objekte und Aktivitäten nutzen.

Ein Inferenzalgorithmus für konkrete Typen, der solche durch parametrischen Polymorphismus entstehenden Unschärfen vermeidet, wird in [Age95] vorgestellt. Die Grundidee dabei ist, daß für verschiedene polymorphe Verwendungen einer Methode unterschiedliche Versionen dieser Methode angelegt werden. Eine solche Version einer Methode heißt Kontur oder Schablone. Jede Kontur einer Methode hat strukturell den gleichen Rumpf; für die Parameter und lokalen Variablen der Kontur werden aber ebenfalls unterschiedliche Versionen in jeder einzelnen Kontur erzeugt. Die nur noch monomorphe Verwendung einer Methodenkontur bewirkt dann, daß die Methodenkonturparameter ebenfalls monomorph werden, da sich in ihnen nur noch die Werte aus den Argumenten von monomorphen Aufrufen mischen. Die Trennung der konkreten Typen bleibt im Rumpf der Methodenkonturen erhalten, da für jede lokale Variable in jeder Kontur eine separate Version angelegt wird. Weiter unten heißen solche Versionen von Variablen auch Variablenkonturen.

```
void serialize①(
  Runnable r1{W1},
  Runnable r2{W1}
) {
  r1{W1}.run();
  r2{W1}.run();
}
void serialize②(
  Runnable r1{W2},
  Runnable r2{W2}
) {
  r1{W2}.run();
  r2{W2}.run();
}

void main() {
  Runnable r{W1} = new W1(){W1};
  Runnable s{W2} = new W2(){W2};
  serialize①(r{W1}, r{W1});
  serialize②(s{W2}, s{W2});
  (new Thread(r{W1})){Thread}.start();
  (new Thread(s{W2})){Thread}.start();
}
```

Abbildung 4
Selektion verschiedener Methodenkonturen

Aufrufen an verschiedenen Stellen des Programms mit unterschiedlichen Argumenttypen werden unterschiedlichen Konturen zugeordnet. In Abbildung 4 wurden für die beiden monomorphen Aufrufe von *serialize()* unterschiedliche Konturen ① und ② selektiert. In diesem Beispiel bleiben die Aufrufe eindeutig, da ihre Argumente monomorph sind. Die Selektion unterschiedlicher Konturen bewirkt in diesem Fall, daß innerhalb der Konturen *serialize*① und *serialize*② die Aufrufziele *r1* und *r2* der Aufrufe von *run()* monomorph werden. Daraus ergibt sich in diesem Teil des Beispiels ein statischer Kontrollfluß.

Hätte ein Methodenaufruf selbst polymorphe Parameter, würden für diesen einen Aufruf mehrere Konturen selektiert. Aus einem Methodenaufruf mit polymorphen Argumenten wird dann ein Methodenaufruf mit mehrdeutigem Ziel (unterschiedlichen Konturen) aber monomorphen Parametern der einzelnen Methodenkonturen.

Polymorphie in den Argumenten von Funktionsaufrufen wird aufgelöst, indem für jedes Tupel aus dem Kreuzprodukt der konkreten Typen der Argumente eine Kontur für die zu analysierende Funktion angelegt wird. Hat ein Parameter einer Methode einen polymorphen d.h. mehrelementigen konkreten Typ, so bedeutet das, daß sich bei einem realen Aufruf zur Laufzeit in dem Parameter eine Instanz einer Klasse aus seinem konkreten Typ befinden kann. Der Aufruf selbst ist monomorph, d.h. zu jedem Aufrufzeitpunkt sind die Klassen der Objekte in allen Parametern der Methode eindeutig. Die Analyse einer Methode ist also auch dann vollständig, wenn jede mögliche Kombination von Klassen in ihren Parametern einzeln untersucht wird.

Jede einzelne Kontur wird separat analysiert. Kontrollflußunschärfen, die durch parametrischen Polymorphismus im Programm entstehen, lassen sich dadurch auflösen. Die Behandlung von parametrischem Polymorphismus alleine genügt allerdings für eine ausreichende Trennschärfe nicht, wie aus der Analyse der restlichen Teile des Programms in Abbildung 2 hervorgeht.

2.4 Datenpolymorphie

Objektorientierte Sprachen erlauben außer der polymorphen Verwendung von Methoden auch sog. Datenpolymorphie, welche die Speicherung von Objekten unterschiedlichen Typs in Instanzvariablen ermöglicht.

```
class Thread {
  Runnable r{W1, W2};

  Thread①(Runnable r{W1}) {
    this.r{W1, W2} = r{W1};
  }
  Thread②(Runnable r{W2}) {
    this.r{W1, W2} = r{W2};
  }

  void run() {
    if (r{W1, W2} != null) r{W1, W2}.run();
  }
}
⋮
class Main {
  ⋮
  void main() {
    Runnable r{W1} = new W1(){W1};
    Runnable s{W2} = new W2(){W2};
    serialize①(r{W1}, r{W1});
    serialize②(s{W2}, s{W2});
    (new Thread①(r{W1})){Thread}.start();
    (new Thread②(s{W2})){Thread}.start();
  }
}
```

Abbildung 5
Datenpolymorphie

Abbildung 5 zeigt die Ergebnisse von Datenflußanalyse und Methodenkonturselektion für die noch nicht betrachteten Teile des Programms aus Abbildung 2. In der Hauptfunktion *main()* erkennt man, daß auch der Konstruktor der Klasse *Thread* polymorph verwendet wird. Auch hierbei handelt es sich um parametrischen Polymorphismus. Dieser wird ebenfalls durch die Selektion von zwei verschiedenen Konturen *Thread①()* und *Thread②()* für den Konstruktor aufgelöst. Der Parameter *r* des Konstruktors wird dadurch in beiden Konturen monomorph. Diese beiden monomorphen Parameter *r* in beiden Konturen des Konstruktors der Klasse *Thread* werden jetzt der Instanzvariablen *Thread.r* zugewiesen. Dabei mischen sich die konkreten monomorphen Typen *{W1}* und *{W2}* dieser beiden Parameter in der Instanzvariable *Thread.r*. Die Selektion der unterschiedlichen Methodenkonturen *Thread①()* und *Thread②()* verhindert zwar, daß sich die konkreten Typen der Argumente beim Aufruf des Konstruktors mischen, kann aber einen unscharfen konkreten Typ der Instanzvariablen *Thread.r* nicht verhindern, da diese Variable zur Klasse *Thread* und nicht zu einer ihrer Methoden gehört. Sie wird also beim Anlegen mehrerer Konturen einer Methode nicht vervielfacht.

Das Anlegen von zwei Konturen für die beiden Konstruktoraufrufe mit *{W1}* und *{W2}* als konkreten Parametertypen reicht also nicht aus, um zu verhindern, daß sich die konkreten Typen beim Speichern in die Instanzvariable *Thread.r* zu *{W1, W2}* mischen. Bei der Analyse der Methode

Thread.run() wird diese Instanzvariable wieder gelesen und ihre Implementation von *run()* aufgerufen. Da die Analyse hier aber den konkreten aber unscharfen Typ *{W1, W2}* vorfindet, kann nicht mehr entschieden werden, ob die Methode *W1.run()* oder die Methode *W2.run()* aufgerufen wird. Der anschließenden Analyse des interprozeduralen Kontrollflußgraphen bleibt damit verborgen, daß bei der Abarbeitung des ersten Threads ausschließlich *W1.run()* und bei Abarbeitung des zweiten nur die Methode *W2.run()* aufgerufen wird.

Für die Gruppierung der Objekte und Aktivitäten ist dieses Ergebnis nutzlos, da nicht erkennbar wird, daß die Aktivität, die durch den ersten *Thread* repräsentiert wird, eine Instanz der Klasse *W1* benutzt und die zweite Aktivität eine Instanz der Klasse *W2*. In jedem zu analysierenden Programm würde die Gruppierung von Objekten und Aktivitäten schon bei dem Versuch scheitern, herauszufinden, welche Aktivität überhaupt welche Klassen als Träger verwendet. Dies liegt daran, daß schon der Start einer Aktivität mit dem Speichern des Trägers der Aktivität in eine Instanzvariable verbunden ist. Aus den Ergebnissen der Datenflußanalyse und der Auflösung von parametrischem Polymorphismus läßt sich also noch keine Information zur Gruppierung von Objekten und Aktivitäten gewinnen.

In [Ple96] wurde ein Algorithmus beschrieben, der parametrischen Polymorphismus ähnlich obigem Prinzip behandelt und gleichzeitig auch Datenpolymorphie durch eine iterative Verfeinerung der Analyse berücksichtigt. Das Prinzip dieses Algorithmus soll hier verwendet werden, um für Java-Programme eine konkrete Typisierung zu erzeugen, mit deren Hilfe sich eine interprozedurale Kontrollflußanalyse ausreichender Trennschärfe für automatische Objektverteilung durchführen läßt. Im folgenden wird der Typinferenzalgorithmus unter Berücksichtigung von Java-spezifischen Eigenschaften erklärt. Kapitel 6 stellt Erweiterungen dieses Typinferenzalgorithmus vor, die sich im Laufe dieser Arbeit als notwendig herauskristallisiert haben, um die Ergebnisse der Typinferenz als Ausgangspunkt für die Berechnung einer Objektverteilung einsetzen zu können.

```
class Thread❶ {
  Runnable r{W1};
  Thread①(Runnable r{W1}) {
    this{Thread❶}.r{W1, W2} = r{W1};
  }
  void run() {
    if (r{W1} != null) r{W1}.run();
  }
}
class Thread❷ {
  Runnable r{W2};
  Thread②(Runnable r{W2}) {
    this{Thread❷}.r{W2} = r{W2};
  }
  void run() {
    if (r{W2} != null) r{W2}.run();
  }
}
⋮
class Main {
  ⋮
  void main() {
    Runnable r{W1} = new W1(){W1};
    Runnable s{W2} = new W2(){W2};
    serialize①(r{W1}, r{W1});
    serialize②(s{W2}, s{W2});
    (new Thread①(r{W1})){Thread❶}.start();
    (new Thread②(s{W2})){Thread❷}.start();
  }
}
```

Abbildung 6
Selektion von Klassenkonturen

Der iterative Typinferenzalgorithmus baut auf der in 2.2 vorgestellten einfachen Typinferenz und der Selektion von Methodenkonturen für monomorphe Verwendungen von Methoden auf. Alle nach diesem ersten Analyseschritt verbleibenden Unschärfen sind auf Datenpolymorphie zurückzuführen. Die Analyse verfolgt diese verbleibenden Unschärfen zurück zu ihren Ursachen und behebt diese. Dabei werden nicht nur von Methoden sondern auch von Klassen Konturen angelegt, um monomorphe Benutzungen von Instanzvariablen der Klassen voneinander zu trennen. Die iterative Verfeinerung mit anschließender Neuberechnung der konkreten Typen wird solange wiederholt, bis entweder alle Unschärfen aufgelöst sind, oder nur noch solche zurückbleiben, die von der Analyse nicht aufgelöst werden können.

Einen Eindruck vom Ziel der Typanalyse vermittelt Abbildung 6. Die polymorph verwendete Klasse *Thread* ist in zwei Klassenkonturen *Thread*❶ und *Thread*❷ geteilt. Damit verwendet die Analyse für beide Konturen der Klasse getrennte Sätze von Instanzvariablen. Der erste erzeugte Thread hat

die Kontur *Thread*❶. Ihre Instanzvariable *r* erhält dadurch den eindeutigen konkreten Typ *{W1}*. Entsprechend wird an der zweiten Stelle ein Objekt der Kontur *Thread*❷ mit der konkret typisierten Instanzvariablen *r :: {W2}* erzeugt. Damit ist der interprozedurale Kontrollfluß an allen Methodenaufrufstellen statisch, da alle konkreten Typen einelementig sind. Die Analyse des Kontrollflusses kann danach feststellen, daß *Thread*❶ ausschließlich die Klasse *W1* benutzt und entsprechend *Thread*❷ nur die Klasse *W2*.

Die nächsten Abschnitte gliedern sich wie folgt: Zuerst wird ein abstraktes Java definiert, das nur noch solche Konstrukte enthält, die für die Typinferenz relevant sind und auf die der iterative Algorithmus bei der Verfeinerung Bezug nimmt. Im Anschluß wird die Umsetzung dieses abstrakten Java in Datenflußungleichungen beschrieben. Die Lösung dieser Datenflußgleichungen ist eine konkrete Typisierung des Programms. Anhand dieser Typisierung werden Unschärfen im interprozeduralen Kontrollfluß erkannt und durch Verfeinerung der Analyse aufgelöst. Diese Verfeinerung ist ein iterativer Prozeß, bei dem durch Anlegen von neuen Konturen die Struktur des Programms verändert wird. Das wirkt sich auf die Datenflußungleichungen aus, und führt in der nächsten Iteration zu einer genaueren neuen Typisierung des Programms.

Zugriffe auf Felder werden dabei so behandelt, als ob immer das selbe Element angesprochen würde. Unschärfen, die durch Speicherung von Objekten unterschiedlichen Typs in ein Feld entstehen, können daher nicht aufgelöst werden. Gleiches gilt für rekursive Datenstrukturen (z.B. Listen), die als Behälter für Objekte unterschiedlichen Typs dienen. Auf Unschärfen, die nicht aufgelöst werden können wird in 2.11 näher eingegangen.

2.5 Transformation in abstraktes Java

Die Operationen des Typinferenzalgorithmus, die Selektion der verschiedenen Sorten von Konturen und das Teilen von Konturen bei Bedarf, stehen in enger Beziehung zu den Strukturen des Programms bestehend aus Klassen, Methoden und deren Anweisungen. Zur Erläuterung des Typinferenzalgorithmus benötigen wir eine Repräsentation des Programms, die einerseits von der konkreten Syntax und den für die Typinferenz unwichtigen Details abstrahiert, es andererseits aber erlaubt, Operationen auf einzelnen Identitäten des Programms zu formulieren. Die Identitäten des Programms sind die in ihm benutzten Klassen, deren Methoden, die lokalen Variablen von Methoden, die statischen Variablen der Klassen, Instanzvariablen und die einzelnen Anweisungen der Methoden. Diese Identitäten sollen als Mengen vorliegen, die durch Relationen untereinander verknüpft werden.

C = Menge aller benutzten Klassen
$\forall$ A, B $\in$ C: B $\leqslant$ A $:\Leftrightarrow$ B ist Unterklasse von A $\Leftrightarrow$ (B = A $\vee$ B erweitert A)

V = Menge aller Variablen des Programms (lokale, globale Variablen und Instanzvariablen)

F = Menge aller Methoden

Der Rumpf einer Methode wird zerlegt in eine Menge von Anweisungen. Dabei werden nur Anweisungen betrachtet, die für die Analyse relevant sind, die also Kanten im Datenflußgraphen verursachen. Ist eine Anweisung verantwortlich für einen Datenfluß zwischen Variablen, so verursacht sie Kanten im Datenflußgraphen des Programms. Für die Datenflußanalyse ist später nur das Vorhandensein einer Kante, nicht aber die Reihenfolge wichtig, mit der sie in den Datenflußgraphen eingefügt wurden. Daher spielt die Reihenfolge von Anweisungen für die Analyse keine Rolle. Das Ergebnis der Datenflußanalyse ist die transitive Hülle aller Datenflüsse, die durch Anweisungen des Programms verursacht werden. Aus einem ähnlichen Grund sind Strukturen, die den lokalen Kontrollfluß einer Methode betreffen, für die Datenflußanalyse unwichtig. Das Ergebnis der Analyse soll nicht dafür verwendet werden, mittels Konstantenfaltung statische Sprungvorhersagen für den lokalen Kontrollfluß in einer Methode zu generieren. Die Datenflußanalyse betrachtet bei allen Kontrollstrukturen im Rumpf einer Methode das Eintreten aller Fälle für möglich. Eine *if-then-else* Struktur verursacht daher immer die Datenflüsse, die zur Berechnung der Sprungbedingung notwen-

dig sind und sowohl die Datenflüsse, die im positiven Zweig der Struktur, als auch die, die im negativen Zweig entstehen. In der abstrakten Repräsentation des Programms treten daher keine Kontrollstrukturen mehr auf. Schleifen können auch unberücksichtigt bleiben, da das Ergebnis der Datenflußanalyse ohnehin die transitive Hülle aller möglichen Datenflüsse enthält.

Eine einzelne Anweisung einer Methode verknüpft in der Regel mehrere Variablen miteinander. Die Variablen sind Behälter für die Werte der Datenflußanalyse. Sie bzw. ihre Konturen sind die Knoten im Datenflußgraphen. In der abstrakten Repräsentation seien daher alle zusammengesetzten Ausdrücke in eine lineare Abfolge von Anweisungen aufgelöst, die zur Berechnung des Ausdrucks notwendig sind. Dabei müssen eventuell neue lokale Variablen eingeführt werden, die im Programm nicht deklariert wurden, um die entstehenden Zwischenergebnisse aufzunehmen.

(assign)	$l_1 = l_2$ oder: assign(l_1, l_2)
(coerce)	$l_1 = (A)\ l_2$ oder: coerce(l_1, A, l_2)
(load)	$l_1 = l_2.a$ oder: load(l_1, l_2, a)
(store)	$l_1.a = l_2$ oder: store(l_1, a, l_2)
(apply)	$l_r = l_t.f(l_1,..., l_n)$ oder: apply(l_r, l_t, f, l_1,..., l_n)
(new)	l_1 = new A oder: new(l_1, A)

Abbildung 7
Abstraktes Java für Analysezwecke

Abbildung 7 charakterisiert die verschiedenen Formen von Anweisungen, aus denen sich der Rumpf einer Methode zusammensetzt. Die verschiedenen Arten von Variablen in *V* werden dabei unterschiedlich behandelt. Instanzvariablen sind immer voll qualifiziert, der Zugriff auf sie erfolgt ausschließlich über die Anweisungen (load) und (store). Im späteren Verlauf der Analyse ist dies wichtig, da bei Unschärfen in Instanzvariablen alle Anweisungen innerhalb des gesamten Programms identifiziert werden müssen, die für Datenflüsse von oder zu dieser unscharfen Instanzvariable verantwortlich sind. Statische Klassenvariablen existieren im Programm nur ein einziges Mal und können (sofern der Zugriff auf sie nicht durch *access-modifier* eingeschränkt ist) überall verwendet werden. Sie werden daher wie globale Variablen des Programms behandelt und können in den Anweisungen von Abbildung 7 an allen Stellen auftreten, an denen lokale Variablen einer Methode stehen können. Alle l_i stehen für lokale Variablen der Methode. In der abstrakten Repräsentation gibt es keine Konstanten. Eine String-Konstante wird beispielsweise behandelt wie eine lokale Variable mit expliziter Konstruktion des String-Objekts in einer (new)-Anweisung. Basistypen erhalten ohnehin eine Sonderbehandlung. In Java ist es nicht möglich, Methoden auf Basistypen aufzurufen. D.h. ein Ausdruck der Form *x.foo()* ist nicht erlaubt, wenn der Typ von *x* ein Basistyp ist. Basistypen spielen daher bei Unsicherheiten im interprozeduralen Kontrollfluß, die durch Polymorphie in solchen Aufrufen entstehen, keine Rolle. (Die Typinferenz beschäftigt sich aber ausschließlich mit der Auflösung solcher interprozeduralen Kontrollflußunschärfen.) Außerdem sind Basistypen für die Objektverteilung uninteressant, da sie sowieso in Java Wertsemantik besitzen. Objekte mit Wertsemantik werden bei Prozeduraufrufen als vollständige Kopie, nicht als Referenz auf ein Objekt übergeben. Es macht daher keinen Sinn, sich Gedanken über eine Plazierungsstrategie für Objekte mit Wertsemantik zu machen; sie wechseln ohnehin bei jedem Methodenaufruf, bei dem sie als Parameter übergeben werden, den Adreßraum. In JavaParty haben auch Instanzen von lokalen Klassen Wertsemantik. Auch sie wechseln bei der Parameterübergabe als vollständige Kopie den Adreßraum. Anders als Basistypen können aber Instanzen von lokalen Klassen Referenzen auf entfernte Objekte in ihren Instanzvariablen enthalten oder entfernte Objekte in ihren Methoden erzeugen. Aus diesem Grund müssen lokale Klassen, trotzdem voll in die Analyse einbezogen werden mit dem einzigen Unterschied, daß bei der Erzeugung einer Instanz einer lokalen Klasse keine Information darüber generiert wird, auf welcher Maschine diese Instanz angelegt werden soll. Instanzen von lokalen Klassen werden immer im Adreßraum ihres Erzeugers angelegt. Werte eines Basistyps hingegen sind für die Analyse uniform durch eine Konstante *simpleValue* repräsentiert.

In einem Methodenaufruf $l_t.f(l_1,..., l_n)$ wird neben den Argumenten $l_1...l_n$ auch das Objekt, das sich in der Variablen l_t befindet und dessen Methode *f* aufgerufen wird, an die Methode übergeben. Zu diesem Zweck hat eine nicht-statische Methode *f* einen zusätzlichen Parameter $this_f$, der beim Methodenaufruf dieses Objekt aufnimmt. Aufrufe einer statischen Methode *g* werden dargestellt als *voidVar.g(…)*. *voidVar* ist dabei eine nur für Analysezwecke eingeführte globale Variable.

In Abbildung 7 taucht keine *return*-Anweisung auf. Jede Methode *f* mit einem von *void* verschiedenen Ergebnistyp besitzt statt dessen eine weitere ausgezeichnete Variable $result_f$. *Return*-Anweisungen in *f* werden übersetzt in Zuweisungen an $return_f$. Methoden mit Ergebnistyp *void* verwenden *voidVar* als Ergebnisvariable. Dadurch läßt sich die Analyse aller Methodentypen in Java mit demselben Formalismus darstellen.

Die Anweisung (new) ist keine legale Java-Anweisung, da eine Objekterzeugung immer an den Aufruf eines Konstruktors gekoppelt ist. Für die Analyse wird diese Verbindung aufgebrochen, ein *new*-Ausdruck $new\ A(p_1,..., p_n)$ wird zerlegt in den Vorgang der argumentlosen Objekterzeugung $l_1 = new\ A$ und die Objekt-Initialisierung $l_1.<init_A>(p_1,..., p_n)$. $<init_A>$ ist dabei der Name des Konstruktors der Klasse *A*. Der einzige Unterschied zum normalen Methodenaufruf ist, daß beim Aufruf eines Konstruktors keine dynamische Methodenauflösung durchgeführt werden muß. Bei der dynamischen Methodenauflösung wird die Implementation der Methode aufgrund der Klasse des Objekts ausgewählt, dessen Methode aufgerufen wird. Beim Aufruf des Konstruktors direkt nach der Objekterzeugung ist dies aber nicht nötig, da die Klasse *A* des erzeugten Objekts immer mit dem Namen des aufgerufenen Konstruktors $<init_A>$ übereinstimmt.

Eine Methode wird dann wie folgt durch einen eindeutigen Namen *f* und eine Reihe von Relationen dargestellt, welche die Methode in Beziehung zu ihren Parametern, Variablen und Anweisungen setzen:

$\forall f \in F$:

$result(f) \in V$	die Ergebnisvariable $result_f$ von f oder voidVar
$this(f) \in V$	die this-Variable $this_f$ in f oder voidVar
$params(f) \in V^n$	die Parameter $(p_1,\ldots, p_n)$ von f
code(f) = Menge von Anweisungen einer Bauart aus Abbildung 7	

In Abbildung 8 ist in Teilen die Übersetzung des Beispielprogramms aus Abbildung 2 in die abstrakte Repräsentation durchgeführt, die als Ausgangspunkt für die sich anschließende Typinferenz dient. Man beachte den Unterschied zwischen der Relation *this(.)* und der *this*-Variablen einer Methode $this_f$. Die Relation *this(.)* verknüpft die Methode run_{Thread} mit ihrer Variablen $this_{run_{Thread}}$, die Methode $main_{Main}$ dagegen mit der Variablen *voidVar*, da es sich bei $main_{Main}$ um eine statische Methode handelt. Eine Variable $this_{main_{Main}}$ existiert nicht. Entsprechendes gilt für die Relation *result(.)*.

```
C = {Object, Runnable, W1, W2, Thread, Main}

F = {
  <init_Object>,
  run_Runnable,
  <init_W1>,          run_W1,
  <init_W2>,          run_W2,
  <init_Thread>,      run_Thread,     start_Thread,
  serialize_Main,     main_Main
}

V = {
  voidVar, SimpleVar,                 // Analyse-Variablen
  this_<init_w1>, this_<init_w2>,     // this-Variablen
```

```
  this<init_Thread>, this_run_Thread,
  p1, p2, p3,                          // Parameter der Methoden (umbenannt wegen
                                       // Namenskonflikten)
  r, s, tmp1, tmp2, tmp3, tmp4,        // lokale und temporäre Variablen
  Thread.r                             // Instanzvariable der Klasse Thread
}

result(<init_W1>) = voidVar
this(<init_W1>) = this<init_w1>
params(<init_W1>) = ()
code(<init_W1>) = {
  apply(voidVar, this<init_w1>, <init_Objekt>) // Aufruf des Konstruktors der erweiterten
                                               // Klasse Object (Wurzel der Klassenhierarchie)
}

result(<init_Thread>) = voidVar
this(<init_Thread>) = this<init_Thread>
params(<init_Thread>) = (p1)
code(<init_Thread>) = {
  apply(voidVar, this<init_w1>, <init_Objekt>),
  store(this<init_Thread>, Thread.r, p1)       // Speichern des Parameters p1 in der
                                               // Instanzvariablen Thread.r
}

result(run_Thread) = voidVar
this(run_Thread) = this_run_Thread
params(run_Thread) = ()
code(run_Thread) = {
  load(tmp1, this_run_Thread, Thread.r),       // Laden der Instanzvariable Thread.r
  assign(tmp2, SimpleVar),                     // Auswertung der Bedingung der Abfrage
  apply(voidVar, tmp1, run_Runnable)           // Ausführung des Rumpfes der Abfrage
}

result(serialize_Main) = voidVar
this(serialize_Main) = voidVar       // statische Methode, ohne this-Parameter
params(serialize_Main) = (p2, p3)
code(serialize_Main) = {
  apply(voidVar, p2, run_Runnable)   // Aufruf der Methode run() auf den Parametern
  apply(voidVar, p3, run_Runnable)   // run_Runnable ist die statisch aufgerufene Methode
                                     // der Schnittstelle Runnable. Die Analyse wählt die
                                     // richtigen Methoden aus, wenn die konkreten Typen
                                     // von p1 und p2 bekannt sind.
}

result(main_Main) = voidVar
this(main_Main) = voidVar                          // statische Methode, ohne this-Parameter
params(main_Main) = ()
code(main_Main) = {
  new(r, W1),                                      // Erzeugen eines Objekts der Klasse W1
  apply(voidVar, r, <init_W1>),                    //   Aufruf des zugehörigen Konstruktors
  new(s, W2),                                      // Erzeugen eines Objekts der Klasse W2
  apply(voidVar, s, <init_W1>),                    //   Aufruf des zugehörigen Konstruktors
  apply(voidVar, voidVar, serialize_Main, r, r),   // Zweimaliger Aufruf von serialize
  apply(voidVar, voidVar, serialize_Main, s, s),
  new(tmp3, Thread)                                // Erzeugen eines neuen Thread
  apply(voidVar, tmp3, <init_Thread>, r),          //   Aufruf des Threadkonstruktors,
                                                   //   Parameter r
  apply(voidVar, tmp3, start_Thread),              //   Starten des neu erzeugten Threads
  new(tmp4, Thread)                                // Erzeugen eines neuen Thread
  apply(voidVar, tmp4, <init_Thread>, s),          //   Aufruf des Threadkonstruktors,
                                                   //   Parameter s
  apply(voidVar, tmp4, start_Thread)               //   Starten des neu erzeugten Threads
}
```

Abbildung 8
Übersetzung in die abstrakte Repräsentation

2.6 Klassen-, Methoden- und Variablenkonturen

Wie in Abschnitt 2.4 ausführlich erläutert wurde, muß man zur Analyse von polymorphen Programmstrukturen sowohl von Methoden als auch von Klassen des Programms Konturen anlegen, die eine monomorphe Verwendung der Methoden bzw. Instanzvariablen repräsentieren. Durch die separate Analyse der einzelnen Konturen können Unschärfen im Kontrollfluß eines Programms aufgelöst werden. Aus der Notwendigkeit der Zerlegung einer Klasse in monomorphe Konturen folgt, daß Mengen von Klassen als konkreter Typ einer Variablen zu ungenau sind. Daher wird als konkreter Typ einer Variablen die Menge von Klassenkonturen verwendet, denen die Objekte angehören, die sich zur Laufzeit in dieser Variablen befinden können.

Von einer Klasse *A* des Programms können nach Bedarf Konturen $A^{❶}$, $A^{❷}$,… angelegt werden. Zu Beginn der Analyse existiert für jede Klasse *A* nur eine Kontur $A^{❶}$. Diese Kontur wird an allen (new)-Anweisungen erzeugt, die diese Klasse instanzieren. Im weiteren Verlauf wird diese Kontur dann bei Bedarf geteilt, um Unschärfen in Instanzvariablen der Klasse aufzulösen.

Für eine Methode *f* werden sukzessive bei Bedarf Konturen $f^{①}$, $f^{②}$,… angelegt, um verschiedene monomorphe Verwendungen dieser Methode getrennt untersuchen zu können. Stößt die Analyse auf einen Methodenaufruf, werden die hierfür benötigten Konturen angelegt, oder schon angelegte passende Konturen selektiert.

Da sich mit den Methodenkonturen die lokalen Variablen einer Methode und mit den Klassenkonturen die Instanzvariablen dieser Klasse vervielfältigen, ist im folgenden nur noch die Rede von Variablenkonturen. Eine Variablenkontur ist eine Version einer Variable in einer sie umgebenden Methoden- oder Klassenkontur. Für eine lokale Variable *v* einer Methode *f* wird für jede Methodenkontur $f^{①}$, $f^{②}$,… eine eigene Kontur $v^{f^{①}}$, $v^{f^{②}}$,… angelegt. Entsprechendes gilt für Instanzvariablen von Klassen: Für eine Instanzvariable *A.b* einer Klasse *A* existiert für jede Kontur $A^{❶}$, $A^{❷}$,… dieser Klasse eine Variablenkontur $A^{❶}.b$, $A^{❷}.b$,…. Statische Klassenvariablen spielen die Rolle globaler Variablen und existieren deshalb nur genau einmal, weswegen für sie auch nur genau eine Kontur gleichen Namens angelegt wird.

Nicht die Variablen des Programms, sondern deren Konturen bilden jetzt die Knoten des Datenflußgraphen. Sie enthalten also die Datenflußwerte und für sie werden die konkreten Typen bestehend aus Mengen von Klassenkonturen berechnet.

CC = Menge aller Klassenkonturen
- $\forall$ cc $\in$ CC: class (cc) $\in$ C — cc ist Kontur der Klasse class(cc)
- $\forall$ A $\in$ C: $\exists$ $A^{❶}$ $\in$ CC: class($A^{❶}$) = A — jede Klasse A hat standardmäßig eine Kontur $A^{❶}$

VC = Menge aller Variablenkonturen
- $\forall$ vc $\in$ VC:
  - var(vc) $\in$ V — vc ist Kontur der Variablen var(vc)
  - value(vc) $\subseteq$ CC — der berechnete konkrete Typ der Variablenkontur vc

FC = Menge aller Methodenkonturen
- $\forall$ fc $\in$ FC:
  - fun(fc) $\in$ F — fc ist Kontur der Funktion fun(fc)
  - result(fc) $\in$ VC — die Kontur der Ergebnisvariable result(fun(fc))
  - params(fc) = $(p_0, p_1, \ldots, p_n) \in VC^{n+1}$ — die Parameter der Methodenkontur, wobei p_0 die Kontur von this(fun(fc)) ist

Die einzelnen Methodenkonturen *fc* $\in$ *FC* einer Methode *f* $\in$ *F* mit *fun(fc)* = *F* erzeugen strukturell denselben Datenflußgraphen. Die Anweisungen der Methodenkontur *fc* sind von derselben Bauart

wie die ihrer Methode *f*. Für jede vorkommende Variable wird nur die zu der Kontur *fc* passende Variablenkontur ausgewählt. Anweisungen einer Methodenkontur werden daher immer als Tupel aus Methodenkontur und Anweisung der zugehörigen Funktion dargestellt.

$\forall$ f $\in$ F: $\forall$ fc $\in$ FC, fun(fc) = f: $\forall$ cmd $\in$ code(f): (fc, cmd) ist Anweisung der Methodenkontur fc

So bezieht sich *assign(l_1, l_2) $\in$ code(f)* in der Kontur *fc* von *f* auf die Variablenkonturen l_1^{fc} und l_2^{fc}. l_1^{fc} bedeutet dabei die in *fc* gültige Version der Variablen l_1. Handelt es sich bei l_1 um eine globale (statische) Variable, so existiert von dieser, wie oben erwähnt, nur eine Kontur und es gilt:

l_1 ist globale Variable $\Rightarrow \forall$ fc, fc' $\in$ FC: $l_1^{fc} = l_1^{fc'}$

l_1 ist lokale Variable $\Rightarrow \forall$ fc, fc' $\in$ FC: l_1^{fc} und $l_1^{fc'}$ sind zwei verschiedene Konturen von l_1

Instanzvariablen treten nur in Anweisungen (load) und (store) auf. Bei ihnen ist die Konturselektion nicht abhängig von der Methodenkontur, in der die Anweisung steht. Beim Laden und Speichern von Instanzvariablen werden für ein Ausdruck *x.a* von allen Klassenkonturen in *value(x)* die zugehörigen Instanzvariablenkonturen selektiert. Dies geschieht unabhängig davon, ob *x* die Kontur einer lokalen oder globalen Variable ist. In *fc* $\in$ *FC* bezieht sich die Anweisung *store(l_1, a, l_2)* mit *value(l_1^{fc}) = {$A^{❶}$, $A^{❷}$, $B^{❶}$}* auf die Instanzvariablenkonturen $A^{❶}.a$, $A^{❷}.a$, $B^{❶}.a$.

```
class Thread {
   ⋮
  void run() {
     ⋮
     tmp{W1, W2} = this{Thread}.r;
     tmp{W1, W2}.run();
     ⋮
  }
  ⋮
}
```

Abbildung 9
Polymorpher Methodenaufruf

Bevor jetzt formal die Erzeugung der Datenflußungleichungen und die Konturselektion für Methoden beschrieben wird, muß noch ein ergänzendes Attribut für Methodenkonturen eingeführt werden. Abbildung 9 ruft noch einmal die Methode run_{Thread} aus dem Beispielprogramm aus Abbildung 2 in Erinnerung. Der Methodenaufruf, welcher ursprünglich die Methode *run()* der Instanzvariablen *Thread.r* aufruft, ist entsprechend den Regeln aus Abbildung 7 in das Laden der Instanzvariable und den Methodenaufruf zerlegt. Die konkreten Typen sind in hochgestellten Klammern an die Variablen(-konturen) annotiert. Da bisher von jeder Funktion erst eine Kontur existiert, werden hier Funktionen und deren Konturen nicht weiter unterschieden. Für den Aufruf $tmp^{\{W1, W2\}}.run()$ werden zwei Methodenkonturen $run_{W1}(this_{run_{W1}})$ und $run_{W2}(this_{run_{W2}})$ selektiert, da der *this*-Parameter *tmp* polymorph ist. Es handelt sich in diesem Fall also um zwei Konturen unterschiedlicher Methoden, was aber für die folgenden Erklärungen keine Rolle spielt. Wichtig ist nur, daß in der einen Kontur der Fall „Parameter hat konkreten Typ {W1}“ und in der anderen der Fall „Parameter hat konkreten Typ {W2}“ abgedeckt wird.

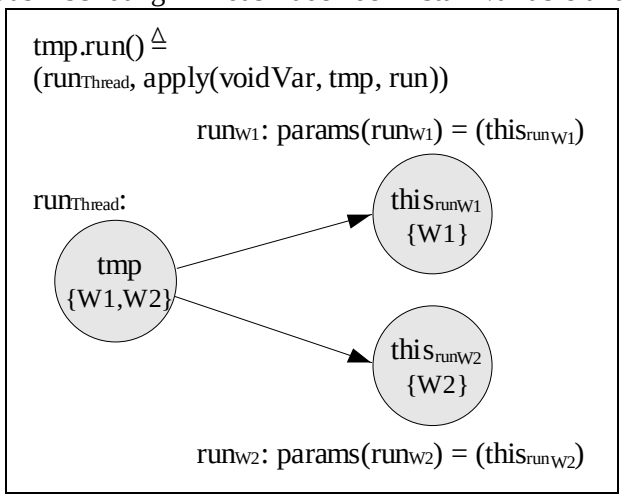

Abbildung 10
Polymorpher Methodenaufruf

Der zugehörige Datenflußgraph, der die Parameterübergabe an dieser Aufrufstelle beschreibt, ist in Abbildung 10 dargestellt. Man erkennt, daß die Datenflußwerte (die konkreten Typen) nicht in gleicher Weise über beide Kanten weitergeleitet werden. Über die Kante, die in der Variablen $this_{run_{W1}}$ mündet, darf nur der Wert *W1* fließen, über die Kante nach $this_{run_{W2}}$ nur der Wert *W2*. Ansonsten würde der Effekt nicht erreicht, einen polymorphen Methodenaufruf in mehreren monomorphen Konturen zu behandeln. Aus diesem Grund erhält jeder Parameter einer Methodenkontur eine ihm zugeordnete Beschränkungsgleichung, die das Fließen von Datenflußwerten in diesen Parameter verhindert, die in dieser Methodenkontur nicht behandelt werden sollen.

Eine Beschränkungsgleichung ist eine Funktion, die Klassenkonturen $cc \in CC$ auf Elemente aus *{true, false}* abbildet. Eine Klassenkontur *cc* kann nur dann in einen Parameter *p* einer Methodenkontur fließen, wenn die *p* zugeordnete Beschränkungsfunktion $r \in R$ den Wert $r(cc) = true$ liefert.

$R = CC \rightarrow \{true, false\}$ Beschränkungsfunktionen auf Datenflußwerten (Klassenkonturen)

Jede Methodenkontur *fc* erhält jetzt zusätzlich zu dem Tupel ihrer Parameter *params(fc)* ein weiteres Tupel mit Beschränkungsgleichungen für diese Parameter *restrict(fc)*. Jede dieser Beschränkungsfunktionen verhindert das Fließen von in der Methodenkontur nicht behandelten Datenflußwerten in den entsprechenden Parameter. Damit können Methodenkonturen gezielt Klassenkonturen für ihre Parameter aussuchen, die in ihnen behandelt werden sollen.

$\forall\, fc \in FC, (p_0,\ldots, p_n) = params(fc)$:
  $restrict(fc) = (r_0, r_1,\ldots, r_n) \in R^{n+1}$ die Beschränkungen der Methodenparameter
  $\forall\, i \in \{0,\ldots, n\}$:
    $\forall\, cc \in value(p_i)$: $r_i(cc) = true$

Mit folgenden Definitionen lassen sich dann die notwendigen Beschränkungsfunktionen für das Beispiel aus Abbildung 10 angeben. Es handelt sich um einige abkürzende Schreibweisen, die Anwendung von $r \in R$ auf Mengen und die Erweiterung auf Tupel.

Für $A \in C$ sei
  $r_A(cc) := (class(cc) = A)$
  $r_{\leqslant A}(cc) := (class(cc) \leqslant A)$

Für $cc \in CC$ sei
  $r_{cc}(cc') := (cc = cc')$

Für $vals \subseteq CC$ und $r \in R$ sei $r(vals) := \{v \in vals \mid r(v) = true\}$ die Trägermenge von r.

Für $vals_i \subseteq CC$ sei
  $(r_0,\ldots, r_n)$ mit $(r_0,\ldots, r_n)(vals_0 \times \ldots \times vals_n) := r_0(v_0) \times \ldots \times r_n(v_n)$
    die Erweiterung auf Mengen von n-Tupeln.

Die Beschränkungsfunktionen für das Beispiel aus Abbildung 10 sehen dann folgendermaßen aus: $restricts(run_{W1}) = (r_{W1})$, die Beschränkungsfunktion, die nur Klassenkonturen der Klasse *W1* zuläßt, und $restrict(run_{W2}) = (r_{W2})$. Im weiteren Verlauf der Analyse ist es dann möglich, Beschränkungen von Methodenkonturparametern zu verfeinern und nur noch das Fließen einzelner Klassenkonturen in die Methodenkonturparameter zuzulassen.

2.7 Der Datenflußgraph

Die Knoten des interprozeduralen Datenflußgraphen $DFG = (VC, E)$ sind die Variablenkonturen des Programms. Für jede Anweisung *cmd* einer Funktion *f* in einer ihrer Konturen *fc* existieren Kanten $(vs, vt) \in E$, wenn sie einen Datenfluß von *vs* nach *vt* verursacht, $vs, vt \in VC$ sind dabei Variablen-

konturen. Da die Art des Datenflusses von der speziellen Anweisung abhängt (z.B. Beschränkungsfunktionen beim Methodenaufruf) wird diese Anweisung *(fc, cmd)* an die verursachten Kanten annotiert: *(fc, cmd) ∈ cmds(vs, vt)*

DFG = (VC, E) interprozeduraler Datenflußgraph
E = VC × VC Kanten von DFG

CMD = $\cup_{f \in F}$ code(f)
$\forall$ e ∈ E: cmds(e) ⊆ FC × CMD die für die Kante verantwortlichen Anweisungen

Der Aufbau des Datenflußgraphen und die Berechnung der Datenflußwerte seiner Knoten kann nicht voneinander getrennt werden. Die Selektion der Konturen beim Methodenaufruf hängt von den Datenflußwerten in den Parametern ab. Andererseits werden für jede selektierte Kontur beim Methodenaufruf mehrere Datenflußkanten zur Parameterübergabe und Rückgabe des Resultats erzeugt. Ähnliches gilt für den Zugriff auf eine Instanzvariable. Eine Anweisung *x.a = b* kann in Abhängigkeit der Werte in der Variablen *x* ebenfalls mehrere Datenflußkanten verursachen. Von jeder Klassenkontur *cc ∈ values(x)* wird die zugehörige Instanzvariablenkontur *cc.a* selektiert und eine Kante *(b →cc.a)* in den Datenflußgraphen eingefügt.

Aufbau und Auswertung der Datenflußungleichungen muß also gleichzeitig erfolgen, da beides voneinander abhängt. Dazu benutzt man eine Fixpunktiteration; ausgehend von dem leeren Datenflußgraphen und einem Aufruf der main-Methode des Programms werden so lange neue Kanten eingefügt und die dadurch entstehenden Datenflußgleichungen ausgewertet, bis sich nichts mehr ändert. Für jede Anweisung einer Methodenkontur werden die entstehenden Datenflußkanten in den Datenflußgraphen eingefügt, die resultierenden Datenflußungleichungen ausgewertet und die Methodenkonturselektion für apply-Anweisungen innerhalb dieser Methodenkontur durchgeführt. Gleiches geschieht darauf für alle selektierten Methodenkonturen. Der Datenflußgraph und alle Werte in seinen Knoten sind dann vollständig bestimmt, wenn sich keine Änderung mehr ergibt.

Der folgende Abschnitt beschreibt detailliert die durch die verschiedenen Anweisungen erzeugten Datenflußkanten und die Methodenkonturselektion.

2.8 Erzeugung der Datenflußungleichungen

Für jede Sorte von Anweisungen des in 2.5 abstrakten Java wird im folgenden untersucht, welche Datenflußkanten sie in den Datenflußgraphen einfügen und wie die durch diese Datenflußkanten repräsentierten Datenflußungleichungen auszuwerten sind.

2.8.1 Instanzierung

In der Regel (new) entstehen keine Datenflußkanten. Die Variablenkonturen, die das Ergebnis einer new-Anweisung aufnehmen, enthalten einen initialen Datenflußwert. Alle Klassenkonturen, die durch die Knoten des Datenflußgraphen fließen, gehen von Variablenkonturen von new-Anweisungen aus. Daher heißt eine solche Variablenkontur auch Erzeugungspunkt. Zu Beginn der Analyse gibt es zu jeder Klasse genau eine Klassenkontur. Die Ergebnisvariable einer new-Anweisung enthält diese initiale Kontur jener Klasse, welche sie zur Laufzeit instanziert. Die Klassenkonturen in Ergebnisvariablen von new-Anweisungen werden im Verlauf der Analyse nur noch verändert, wenn Klassenkonturen geteilt werden, ansonsten bleiben sie unverändert. Wird später die Kontur einer Methode geteilt, hat das keine Auswirkungen auf die Klassenkonturen ihrer new-Anweisungen. Alle Konturen, in die die Methodenkontur geteilt wird, enthalten nach der Teilung an korrespondierenden new-Anweisungen dieselben Datenflußwerte wie die geteilte Methodenkontur.

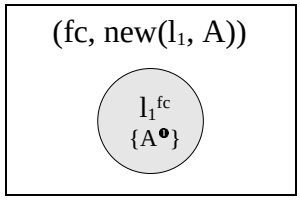

Abbildung 11
Objekt Instanzierung

Abbildung 11 zeigt die Variablenkontur der Anweisung l_1 = *new A* in einer Methodenkontur *fc*. Diese new-Anweisung instanziert die Klasse *A*. Ihre Variablenkontur l_1^{fc} enthält deshalb den Datenflußwert $A^{❶}$, die initiale Kontur der Klasse *A*.

(new) $\forall$ f $\in$ F: $\forall$ cmd $\in$ code(f), cmd = new(l_1, A): $\forall$ fc $\in$ FC, fun(fc) = f:
$A^{❶}$ sei die standardmäßig für die Klasse A angelegte Kontur
value(l_1^{fc}) = {$A^{❶}$}

2.8.2 Zuweisung

Die Anweisung (assign) erzeugt eine einzelne Datenflußkante, duch die die Datenflußwerte der verbundenen Knoten ungehindert fließen können.

(assign) $\forall$ f $\in$ F: $\forall$ cmd $\in$ code(f), cmd = assign(l_1, l_2):
$\forall$ fc $\in$ FC, fun(fc) = f:
value(l_1^{fc}) $\supseteq$ value(l_2^{fc})
(l_2^{fc}, l_1^{fc}) $\in$ E

Abbildung 12 veranschaulicht die Datenflußkante, die durch eine Anweisung $l_1 = l_2$ innerhalb der Funktionskontur *fc* entsteht. Die betroffenen Variablenkonturen sind l_1^{fc} und l_2^{fc}. In l_2^{fc} werden die Datenflußwerte *values*(l_2^{fc}) = {$A^{❶}$, $B^{❷}$} angenommen. Diese werden über die durch die Anweisung induzierte Kante ungehindert nach l_1^{fc} weitergeleitet.

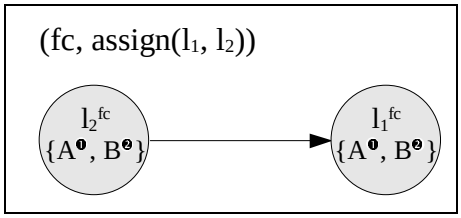

Abbildung 12
Zuweisung

2.8.3 Typkonvertierung

Der Fluß von Werten über die durch *coerce*(l_1, *A*, l_2) induzierte Kante ist durch eine Bedingung $r_{\leqslant A} \in R$ eingeschränkt. In Abbildung 13 sieht man, daß nur der Wert $A^{❶}$ über die durch *coerce*(l_1, *A*, l_2) verursachte Kante weitergeleitet wird, während der Wert $B^{❷}$ blockiert wird.

Die Typkonvertierung ist eine Zusicherung, daß zur Laufzeit in l_2 nur Instanzen von Unterklassen von *A* gespeichert sind. Befindet sich zur Laufzeit eine Instanz einer Klasse *B* mit $B \not\leqslant A$ in l_2, so wird das Programm mit einem Laufzeitfehler abgebrochen. Bei der Berechnung der Datenflußwerte für das Ergebnis l_1 einer Typkonvertierung l_1 = *(A)* l_2 müssen daher solche Klassenkonturen *B* mit $\neg r_{\leqslant A}(B)$ aus dem Datenfluß an dieser Stelle herausgefiltert werden.

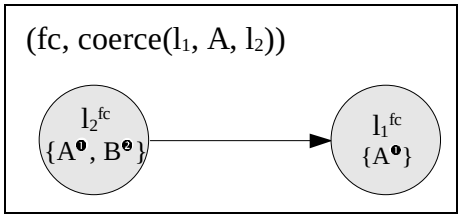

Abbildung 13
Typkonvertierung

(coerce) $\forall$ f $\in$ F: $\forall$ cmd $\in$ code(f), cmd = coerce(l_1, A, l_2): $\forall$ fc $\in$ FC, fun(fc) = f:
value(l_1^{fc}) $\supseteq$ $r_{\leqslant A}$(value(l_2^{fc})) = {cc $\in$ value(l_2^{fc}) | class(cc) $\leqslant$ A}
(l_2^{fc}, l_1^{fc}) $\in$ E

Zum einen ist klar, daß ein Datenflußwert der Klasse *B* mit $B \not\leqslant A$ an dieser Stelle aus dem Datenfluß herausgefiltert werden muß: Ließe man einen solchen Wert die Typkonvertierung passieren, hätte dies zur Folge, daß sich in Variablen nach der Typkonvertierung Datenflußwerte befinden könnten, die nicht mit dem definierten Typ dieser Variablen kompatibel wären. Das ist unzulässig; konkrete

Typen müssen immer eine bessere Beschreibung möglicher Laufzeitobjekte in den Variablen des Programms liefern, als die definierten Typen das tun. Die Verletzung dieser Bedingung würde im weiteren Verlauf der Analyse Probleme beim Aufruf von Methoden und dem Zugriff auf Instanzvariablen verursachen.

Zum anderen kann man sich fragen, wie es in einem Programm, das zur Laufzeit an dieser Stelle nie einen Typfehler verursachen wird, möglich ist, daß solche inkompatiblen Datenflußwerte überhaupt erst die Variablenkontur l_2^{fc} erreichen. Im allgemeinen liegt das an der polymorphen Verwendung von Behälterklassen, die noch nicht adäquat durch Teilen ihrer Klassenkonturen modelliert sind. Wird eine Behälterklasse mit einem Element des deklarierten Typs *Object* im Programm an unterschiedlichen Stellen einmal zur Speicherung von Objekten der Klasse *A* und einmal zur Speicherung von Objekten der Klasse *B* verwendet, so treten in der Regel an den Programmstellen, an denen das Element wieder ausgelesen wird, Typkonvertierungen auf.

Da die Analyse den Behälter zu Beginn aber noch nicht in zwei unterschiedliche Klassenkonturen geteilt hat, besitzt die Instanzvariable des Behälters den unscharfen konkreten Typ *{A, B}*, der beim Auslesen der Instanzvariable an die Variable weitergegeben wird, die das Ergebnis der Leseoperation aufnimmt. Das Programm geht bei der Durchführung der Typkonvertierung korrekterweise davon aus, daß diese nach dem Lesen der Instanzvariable des Behälters erfolgreich verläuft, da in das betreffende Behälterobjekt nur Objekte der Klasse *A* gespeichert wurden. Die Analyse hat bis jetzt aber alle Verwendungen der Behälterklasse betrachtet und alle auftretenden konkreten Typen in ihren Instanzvariablen gemischt.

```
class A { … }
class B { … }

class Pair {
  Object fst;
  Object snd;

  Pair(Object fst, Object snd) {
    this.fst = fst;
    this.snd = snd;
  }
}
   ⋮
  Pair p = new Pair(new A(), new A());
  Pair q = new Pair(new B(), new B());
  A a = (A) p.fst;
  B b = (B) q.snd;
   ⋮
```

Abbildung 14
Polymorpher Behälter

Abbildung 14 gibt ein Beispiel für eine solche Verwendung einer Behälterklasse. Die Klasse *Pair* wird einmal verwendet, um Instanzen von *A* und einmal Instanzen von *B* zu speichern. Die notwendige Typkonvertierung beim Lesen der Elemente von *Pair* verläuft immer erfolgreich, die Analyse findet aber in beiden Instanzvariablen von *Pair* den unscharfen konkreten Typ *{A, B}* vor. Erst im weiteren Verlauf der Analyse kann diese versuchen, die Klassenkontur des Behälters zu teilen, um die einzelnen monomorphen Verwendungen des Behälters zu trennen.

2.8.4 Laden einer Instanzvariable

In Abbildung 15 sind die Datenflußkanten eingezeichnet, die von einer Anweisung $l_1 = l_2.a$ in einer Methodenkontur *fc* verursacht werden. Dabei wird angenommen, daß sich in l_2^{fc} die Werte $value(l_2^{fc}) = \{B^{❶}, B^{❷}, D^{❶}\}$ befinden. l_2^{fc} ist die zur Variablen l_2 gehörenden Variablenkontur in *fc*. Der definierte Typ von l_2 sei dabei der Klassentyp *B* und $D \leqslant B$ sei eine Unterklasse von *B* (eine Schnittstelle ist hier nicht möglich, da die Definition von Instanzvariablen in Schnittstellen nicht erlaubt ist).

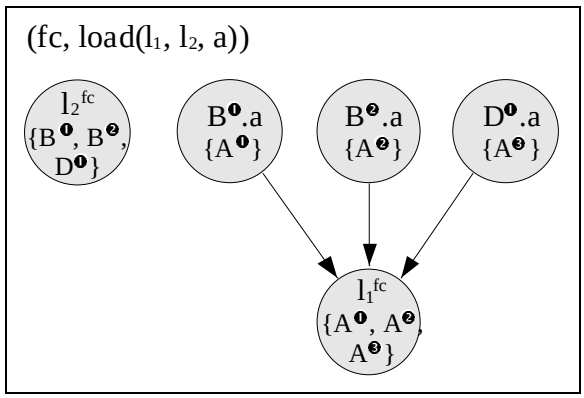

Abbildung 15
Laden einer Instanzvariablen

Das Laden einer Instanzvariablen bezieht sich immer auf das Objekt, das in der Variablen gespeichert ist, von welcher die Instanzvariable selektiert

wird (hier l_2^{fc}). Während der Analyse beschreibt der konkrete Typ von l_2^{fc} die Menge der Klassenkonturen, von welchen zur Laufzeit Instanzen in l_2 gespeichert sein können. Die Ladeanweisung selektiert dann von allen Klassenkonturen in *value*(l_2^{fc}) die entsprechende Instanzvariable und fügt Datenflußkanten von diesen Instanzvariablen zur Ergebnisvariable der Ladeanweisung (l_1^{fc}) in den Datenflußgraphen ein. Dabei mischen sich die Werte aus allen betroffenen Instanzvariablen in der Ergebnisvariablen der Anweisung. Im Beispiel aus Abbildung 15 werden von den Klassenkonturen $B^{❶}$, $B^{❷}$ und $D^{❶}$ aus *value*(l_2^{fc}) die zugehörigen Instanzvariablen $B^{❶}.a$, $B^{❷}.a$ und $D^{❶}.a$ selektiert. Von diesen drei Instanzvariablen wird dann jeweils eine Kante zur Ergebnisvariablen l_1^{fc} der Anweisung eingefügt.

Die Anzahl der durch die Anweisung *load*(l_1, l_2, *a*) in der Methodenkontur *fc* erzeugten Datenflußkanten hängt also von der Anzahl der Klassenkonturen B^C in der Variablenkontur l_2^{fc} ab. Wie in obigem Beispiel gezeigt, bezieht sich der Zugriff auf jede Instanzvariablenkontur $B^C.a$, deren zugehörige Klassenkontur B^C sich in der Variablen l_2^{fc} befindet. $B^C.a$ soll dabei die zur Klassenkontur B^C gehörende Variablenkontur der Instanzvariable *B.a* sein. B^C soll andeuten, daß es sich um eine Kontur der Klasse *B* handelt. Es gilt zu beachten, daß sich nicht nur mehrere Konturen einer Klasse in *value*(l_2^{fc}) befinden können, sondern auch Konturen unterschiedlicher Klassen. Die Menge möglicher Klassen hängt vom deklarierten Typ der Variable l_2 ab. Ist der deklarierte Typ von l_2 der Klassentyp *B*, so können sich Klassen $D \in C$ mit $D \preccurlyeq B$ in der Kontur l_2^{fc} der Variablen l_2 befinden. Alle diese Klassen *D* besitzen dieselbe (ererbte) Instanzvariable *B.a* unabhängig davon, ob sie eine neue Instanzvariable gleichen Namens deklarieren, die *B.a* verdeckt. Die nachfolgenden Formeln sind modulo solcher Namenskonflikte zu betrachten.

(load) $\forall$ f $\in$ F: $\forall$ cmd $\in$ code(f), cmd = load(l_1, l_2, a): $\forall$ fc $\in$ FC, fun(fc) = f:

$\forall$ $B^C \in$ value(l_2^{fc}):

value(l_1^{fc}) $\supseteq$ value(B^C.a),

(B^C.a, l_1^{fc}) $\in$ E

2.8.5 Speichern in eine Instanzvariable

Das für eine (load)-Anweisung Gesagte gilt in gleicher Weise auch für die (store)-Anweisung mit dem einen Unterschied, daß sich die Richtung der Datenflußkanten umkehrt. Der Effekt für die im Datenflußgraphen fließenden Werte ändert sich dabei ebenfalls. Bei der (load)-Anweisung mischen sich alle Werte der betroffenen Instanzvariablenkonturen in der Kontur der lokalen Variable, die das Ergebnis der Anweisung aufnimmt. Bei der (store)-Anweisung werden die (möglicherweise) unscharfen Werte in der zu speichernden Variablenkontur (l_2^{fc}) an alle betroffenen Instanzvariablenkonturen weitergeleitet.

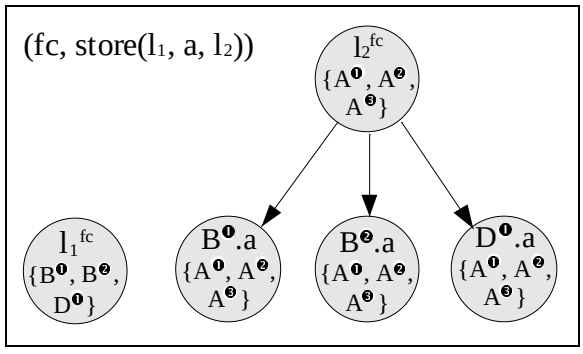

Abbildung 16
Speichern in eine Instanzvariable

Abbildung 16 nimmt an, daß eine Anweisung $l_1.a = l_2$ in der Funktionskontur *fc* vorliegt. Die Variablenkontur l_1^{fc} enthalte dabei die Werte $value(l_1^{fc}) = \{B^{❶}, B^{❷}, D^{❶}\}$. Von diesen Klassenkonturen werden jetzt die jeweiligen Instanzvariablenkonturen $B^{❶}.a$, $B^{❷}.a$ und $D^{❶}.a$ selektiert. Durch Einfügen von drei Kanten in den Datenflußgraphen ausgehend von der Variablenkontur l_2^{fc} werden daraufhin die Werte $value(l_2^{fc}) = \{A^{❶}, A^{❷}, A^{❸}\}$ in der zu speicherten lokalen Variablen l_2^{fc} an die selektierten Instanzvariablenkonturen weitergeleitet.

In den Abbildungen 14 und 15 ist davon abzusehen, daß zu Beginn der Analyse von jeder Klasse erst eine Kontur existiert. Klassenkonturen werden erst im Verlauf der Analyse geteilt, die Abbildungen dienen lediglich zur Veranschaulichung der Datenflußkanten.

(store) $\forall f \in F: \forall cmd \in code(f), cmd = store(l_1, a, l_2): \forall fc \in FC, fun(fc) = f:$

$\forall B^c \in value(l_1^{fc}):$

$value(B^c.a) \supseteq value(l_2^{fc}),$

$(l_2^{fc}, B^c.a) \in E$

2.8.6 Methodenaufruf

Der Datenfluß beim Methodenaufruf wird in Abbildung 17 anhand einer Anweisung *cmd = apply(l_r, l_0, g, l_1)* innerhalb der Methodenkontur *fc* beispielhaft dargestellt. Die mehr an Java angelehnte Schreibweise dieser Anweisung wäre $l_r = l_0.g(l_1)$. Das Argument l_1^{fc} enthält den mehrdeutigen konkreten Typ $value(l_1^{fc}) = \{B^{❶}, D^{❶}\}$, das *this*-Argument l_0^{fc} ist eindeutig. Für diesen Aufruf *(fc, cmd)* der Methode *g* wurden zwei Methodenkonturen $g^{①}$ und $g^{②}$ selektiert. Diese selektierten Konturen beschreibt die Menge $selected(fc, cmd) = \{g^{①}, g^{②}\}$, sie bleibt zwischen den nachfolgenden Verfeinerungsschritten der Analyse erhalten, da die Verfeinerungsschritte unter anderem auf dieser Menge aufbauen. Die Regeln der Konturselektion werden weiter unten ausführlich besprochen. Für dieses Beispiel ist nur wichtig, daß die Methodenkonturen immer so selektiert werden, daß die Parameter der selektierten Methodenkonturen monomorph in Bezug auf ihre Klasse bleiben. Dies wird durch entsprechende Beschränkungsgleichungen für die Parameter erreicht. Da die zu den Klassenkonturen in $value(l_1^{fc})$ gehörenden Klassen nicht eindeutig sind, unterscheiden sich die Beschränkungsfunktionen $restrict(g^{①}) = (r_A, r_B)$ und $restrict(g^{②}) = (r_A, r_D)$ an dieser Stelle. Die Funktionsergebnisse der beiden Methodenkonturen vereinigen sich anschließend wieder in der Kontur der Ergebnisvariablen l_r.

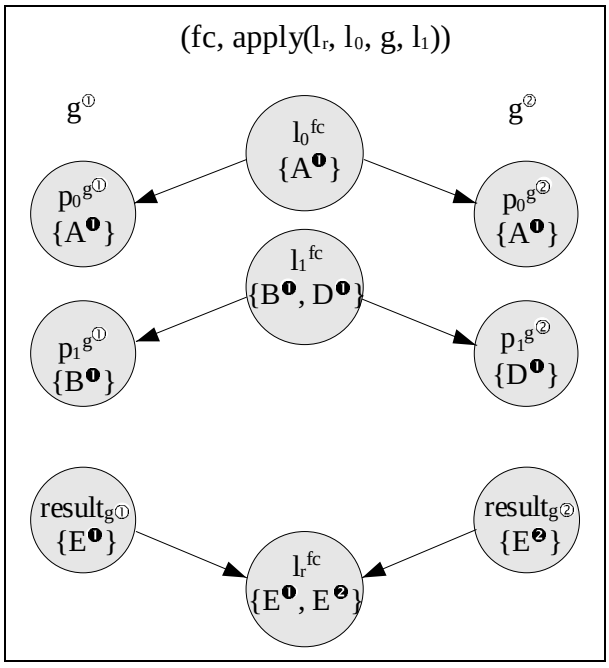

Abbildung 17
Methodenaufruf

(apply) $\forall f \in F: \forall cmd \in code(f), cmd = apply(l_r, l_0, g, l_1, \ldots, l_n): \forall fc \in FC, fun(fc) = f:$
$\forall gc \in selected(fc, cmd):$

(i) $(p_0^{gc}, \ldots, p_n^{gc}) = params(gc)$

(ii) $r = (r_0, \ldots, r_n) = restrict(gc)$

(iii) $vr^{gc} = result(gc)$

(iv) $(p_0^{gc}, \ldots, p_n^{gc}) \leftarrow r(value(l_0^{fc}) \times \ldots \times value(l_n^{fc}))$

(v) $value(l_r^{fc}) \supseteq value(vr^{gc})$

(vi) $(l_0^{fc}, p_1^{gc}), \ldots, (l_n^{fc}, p_n^{gc}) \in E$

(vii) $(vr^{gc}, l_r^{fc}) \in E$

(viii) $\Pi_{i=0\ldots n}\, value(l_i^{fc}) \subseteq \cup_{gc \in selected(fc, cmd)} \Pi_{i=0\ldots n}\, value(p_i^{gc})$

(i)-(iii) verkürzen lediglich die Schreibweise. (iv) beschreibt die Verteilung der Wertkombinationen $VALUES_{(fc, cmd)} := value(l_0^{fc}) \times \ldots \times value(l_n^{fc})$ der Argumente an die Konturen der Methodenparameter. Im Beispiel aus Abbildung 17 ist $VALUES_{(fc, cmd)} = \{(A^{\bullet}, B^{\bullet}), (A^{\bullet}, D^{\bullet})\}$. Jede dieser Wertkombinationen muß nur an eine Methodenkontur weitergeleitet werden (jeder einzelne reale Aufruf ist monomorph). Mit '$\leftarrow$' anstatt '$\subseteq$' soll zum Ausdruck kommen, das die Kombinationen in $VALUES_{(fc, cmd)}$ sequentiell an die selektierten Konturen *selected(fc, cmd)* verteilt werden, so daß sichergestellt ist, daß eine Kombination nicht an mehrere Methodenkonturen zur Bearbeitung zugewiesen wird.

(v) beschreibt die Rückgabe des Funktionsergebnisses, (vi)-(vii) geben die resultierenden Datenflußkanten an. Jede Parameterkombination muß mindestend in einer Kontur behandelt werden, damit die Analyse vollständig ist. (viii) verlangt daher, daß jede Kombination aus den Werten der Argumente des Aufrufs als Parameterkombination einer selektierten Methodenkontur auftreten muß.

2.8.7 Selektion von Methodenkonturen

Der vorhergehende Abschnitt 2.8.6 beschreibt nur einen Aspekt des Methodenaufrufs beim Aufbau des Datenflußgraphen nämlich die Erzeugung der Datenflußkanten und die Weiterleitung der Datenflußwerte über diese Kanten. Dies geschieht für einen Methodenaufruf *(fc, cmd)* in Abhängigkeit der Menge *selected(fc, cmd)*, der für diesen Aufruf selektierten Methodenkonturen. Dieser Abschnitt beschreibt wie man für einen Methodenaufruf *(fc, cmd)* diese Menge *selected(fc, cmd)* erhält.

Wie in 2.3 beschrieben, soll Polymorhie in den Parametern von Methoden aufgelöst werden. Dies geschieht durch geeignete Konturselektion beim Aufruf von Methoden. Die Parameter einer Methodenkontur sollen nur Konturen einer einzigen Klasse als Werte aufnehmen. Enthält ein Argument Konturen unterschiedlicher Klassen, so werden für diesen Aufruf mehrere Konturen selektiert. Beim Methodenaufruf aus Abbildung 17 werden zwei Parameter übergeben (l_0 und l_1). Der Parameter l_0 enthält eine Kontur der Klasse *A*, der Parameter l_1 enthält Konturen der Klassen *B* und *D*. Um diesen polymorhen Aufruf (es befinden sich zwei Konturen unterschiedlicher Klassen im Parameter l_1) in monomorphen Konturen behandeln zu können, müssen mindestens so viele Konturen selektiert werden, wie das Kreuzprodukt der Klassen der Argumentwerte Elemente umfaßt. In diesem Fall wäre das Kreuzprodukt der Klassen die Menge *{(A, B), (A, D)}*; es müssen also mindestens zwei Methodenkonturen selektiert werden.

Die Selektion von Methodenkonturen geschieht in zwei Schritten. Zuerst wird versucht, die Parameterkombinationen mit schon existierenden Methodenkonturen abzudecken. Erst im zweiten Schritt werden dann für die verbleibenden Kombinationen neue Konturen erzeugt. Die neu erzeugten

Konturen erhalten für ihre Parameter immer Beschränkungsgleichungen, die diese auf Konturen einer einzigen Klasse einschränken. Im späteren Verlauf der Analyse können diese Einschränkungen dann weiter verschärft werden, was zu mehr selektierten Methodenkonturen pro Aufruf führt. Dieses Teilen von Methodenkonturen wird im Abschnitt 2.10.5 behandelt.

Die Menge der noch an keine Methodenkontur zugewiesenen Parameterkombinationen sei $VALUES_{(fc,\ cmd)}$. Die Selektion von schon existierenden Methodenkonturen ist unter (i) beschrieben; Gibt es unter den schon existierenden Methodenkonturen mehrere, die auf eine noch zu verteilende Parameterkombination passen, wird zuerst diejenige selektiert, zu der die wenigsten Parameterkombinationen passen, die also die schärfsten Parameterbeschränkungen hat. Dies geschieht mit der Begründung, daß schon existierende spezialisierte Methodenkonturen auch verwendet werden sollen. Dies verhindert, daß im Verlauf der Analyse viele gleiche Versionen einer solchen spezialisierten Version erzeugt werden.

(i) $\exists$ gc' $\in$ FC, gc' $\notin$ selected(fc, cmd): $\exists$ (cc, v_1,…, v_n) $\in$ $VALUES_{(fc, cmd)}$:
fun(gc') = g', g' = dispatch(g, class(cc))
$\Rightarrow$ füge gc' zu selected(fc, cmd) hinzu, verteile passende Parameterkombinationen an gc'

$\forall$ g $\in$ F: $\forall$ A $\in$ C:
dispatch(g, A) = die (evtl. überschriebene) Version der Methode g in der Klasse A

Solange (i) zutrifft werden schon existierende Konturen selektiert, die zu der Klasse eines Wertes im *dispatch*-Parameter l_0 passen. Gibt es keine passenden Methodenkonturen mehr, werden neue Methodenkonturen erzeugt:

(ii) $\exists$ (cc_0, cc_1,…, cc_n) $\in$ $VALUES_{(fc, cmd)}$: g' = dispatch(g, class(cc_0)),
gc' = neue Funktionskontur von g' mit
fun(gc') = g'
restrict(gc') = ($r_{class(cc_0)}$,…, $r_{class(cc_n)}$)
füge gc' zu selected(fc, cmd) hinzu, verteile passende Parameterkombinationen an gc'

Für eine noch nicht verteilte Parameterkombination $(cc_0,\ldots, cc_n)$ wird eine neue Kontur angelegt, deren Parameter auf die Klassen $class(cc_0)\ldots class(cc_n)$ eingeschränkt werden. Diese neue Kontur *gc'* wird der Menge der für den Aufruf selektierten Konturen *selected(fc, cmd)* hinzugefügt. Gibt es unter den noch nicht verteilten Parameterkombinationen solche, die zu den Beschränkungen der neu angelegten Kontur passen, so werden auch diese an die neue Methodenkontur zugewiesen. Dieser Vorgang wird so lange wiederholt, bis alle Parameterkombinationen aus $VALUES_{(fc, cmd)}$ verteilt sind. Dadurch, daß neue Methodenkonturen für ihre Parameter immer mindestens Einschränkungen auf Klassen erhalten, ist sichergestellt, daß rein parametrischer Polymorphismus schon im ersten Durchgang ohne Verfeinerung der Analyse aufgelöst wird. Zur Auflösung von Datenpolymorphie kann es bei Verfeinerung der Analyse notwendig werden, die Parameter von Methodenkonturen weiter einzuschränken.

2.9 Unschärfen

Die Lösung der initialen Datenflußungleichungen, wie in 2.8 beschrieben, erreicht eine Trennschärfe wie der Algorithmus aus [Age95], da beim Methodenaufruf immer mindestens so viele Methodenkonturen selektiert werden, daß alle Parameter monomorph bleiben. Dabei wird für Methodenaufrufe mit polymorphen Argumenten für jede Parameterkombination eine eigene Methodenkontur selektiert. Jede Klasse ist aber bis jetzt nur durch eine Kontur repräsentiert. Demzufolge ist für jede Instanzvariable auch erst eine Kontur angelegt. Wird eine Klasse in unterschiedlichen Kontexten mit verschiedenen Typen verwendet, sind die konkreten Typen ihrer Instanzvariablen eine Mischung aller konkreten Typen aus allen ihren Verwendungen.

Im folgenden heißt eine Variablenkontur vc, die mehrere Datenflußwerte enthält, *unscharf*. Unschärfen lassen sich, je nachdem welche Sorten von Datenflußwerten man betrachtet und wie genau man diese Datenflußwerte unterscheidet, in mehrere Kategorien unterteilen. Die einzigen bisher eingeführten Datenflußwerte sind die Klassenkonturen, die den konkreten Typ einer Variablenkontur bilden. In einer Variablenkontur *x* tritt eine *Konturunschärfe* auf, wenn ihr konkreter Typ *value(x)* mehrere Klassenkonturen enthält. Die Variablenkonturen *x* mit *value(x)* = $\{A^{\bullet}, A^{\bullet\bullet}\}$ und *y* mit *value(y)* = $\{A^{\bullet}, B^{\bullet}\}$ sind beide unscharf bezüglich der in ihrem konkreten Typ enthaltenen Klassenkonturen. Man spricht von einer *Klassenunschärfe*, wenn sich auch die Klassen der Konturen des konkreten Typs unterscheiden. Die einzige Klasse, von welcher Konturen im konkreten Typ von *x* auftreten ist *A*. Daher ist *x* scharf bezüglich der Klassen im konkreten Typ. Nicht so *y*; der konkrete Typ der Variablenkontur *y* enthält die Klassen *A* und *B*, damit tritt in *y* eine Klassenunschärfe auf.

Gibt es eine Klassenunschärfe in einer Variablenkontur *y*, treten also in ihrem konkreter Typ Klassenkonturen unterschiedlicher Klassen auf, so enthält der konkrete Typ von *y* mehrere Klassenkonturen und es existiert damit auch eine Konturunschärfe in *y*. Eine Klassenunschärfe ist also immer auch eine Konturunschärfe, nicht aber umgekehrt.

Methodenaufruf x.g() ***in fc***	***selektierte Konturen des Aufrufs***
Methodenkontur fc: apply(voidVar, x, g) =: cmd value(x) = $\{A^{\bullet}, A^{\bullet\bullet}, B^{\bullet}\}$ selected(fc, cmd) = $\{g^{①}, g^{②}\}$	Methodenkontur $g^{①}$: restrict($g^{①}$) = (r_A) params($g^{①}$) = ($p^{g^{①}}$) value($p^{g^{①}}$) = $\{A^{\bullet}, A^{\bullet\bullet}\}$ Methodenkontur $g^{②}$: restrict($g^{②}$) = (r_B) params($g^{②}$) = ($p^{g^{②}}$) value($p^{g^{②}}$) = $\{B^{\bullet}\}$

Abbildung 18
Kontrollflußunsicherheit

Um einen statischen Kontrollfluß zu erzeugen, muß nicht jede im Programm auftretende Unschärfe auch aufgelöst werden. In Abbildung 18 betrachtet man einen Methodenaufruf *x.g()*, bei dem man annimmt, daß der konkrete Typ von *x* die Werte *value(x)* = $\{A^{\bullet}, A^{\bullet\bullet}, B^{\bullet}\}$ enthält. Für diesen Methodenaufruf sind zwei Konturen $g^{①}$ und $g^{②}$ selektiert. Es ist daher noch nicht klar, welche der beiden Methodenkonturen zur Laufzeit an diese Stelle angesprungen werden. Der Kontrollfluß an dieser Stelle ist nicht statisch, da die Entscheidung über die ausgeführte Methodenkontur noch vom Laufzeitobjekt in *x* abhängig ist.

Die Methodenkontur $g^{①}$ behandelt in diesem Beispiel den Fall, daß das Laufzeitobjekt in *x* einer Klassenkontur aus $\{A^{\bullet}, A^{\bullet\bullet}\}$ angehört, und $g^{②}$ den Fall, daß sich in *x* zur Laufzeit ein Objekt der Klassenkontur $B^{\bullet}$ befindet. Da $g^{①}$ also die Parameterkombination $\{A^{\bullet}, A^{\bullet\bullet}\}$ abdeckt, wird durch Auflösen der Klassenunschärfe in *x* der Kontrollfluß an diesem Methodenaufruf statisch. Es ist nicht nötig, auch die Konturunschärfe in *x* aufzulösen, da für die Werte $A^{\bullet}$ und $A^{\bullet\bullet}$ des konkreten Typs von *x* ohnehin dieselbe Kontur $g^{①}$ selektiert würde.

Die aufzulösende Unschärfe soll deshalb formal spezifiziert werden. Dies geschieht durch die charakteristische Funktion *p* der Unschärfe. Diese Funktion bildet eine Variablenkontur $vc \in VC$ auf eine Menge von Werten ab, die für die Unschärfe verantwortlich sind. Eine Variablenkontur *vc* ist genau dann unscharf bezüglich der charakteristischen Funktion *p*, wenn die Menge *p(vc)* mehrelementig ist.

$\forall$ vc $\in$ VC: |p(vc)| > 1 $\Rightarrow$ vc unscharf bezüglich p
|p(vc)| = 1 $\Rightarrow$ vc scharf bezüglich p

Eine Klassenunschärfe in *vc* ist dadurch gekennzeichnet, daß *vc* die Konturen mehrerer Klassen enthält. Die charakteristische Funktion p_{class} einer Klassenunschärfe bildet *vc* daher auf die Menge der Klassen ab, von denen *vc* Konturen enthält.

p_{class} : VC $\rightarrow$ C mit p(vc) = {class(cc) | cc $\in$ value(vc)}

Eine Konturunschärfe entsteht durch mehrere Klassenkonturen in einer Variablenkontur *vc*. Die charakteristische Funktion $p_{contour}$ einer Konturunschärfe bildet *vc* also gerade auf die Menge *value(vc)*, die Menge der Klassenkonturen in *vc*, ab.

$p_{contour}$: VC $\rightarrow$ CC mit p(vc) = value(vc)

Im Beispiel aus Abbildung 18 hatte die Variablenkontur *x* den konkreten Typ *value(x)* = $\{A^{\textcircled{1}}, A^{\textcircled{2}}, B^{\textcircled{1}}\}$. Damit ist $p_{contour}(x) = value(x) = \{A^{\textcircled{1}}, A^{\textcircled{2}}, B^{\textcircled{1}}\}$ und $p_{class}(x) = \{class(cc) \mid cc \in \{A^{\textcircled{1}}, A^{\textcircled{2}}, B^{\textcircled{1}}\}\} = \{A, B\}$. Beide Mengen sind mehrelementig $|p_{contour}(x)| > 1$ und $|p_{class}(x)| > 1$; es gibt in *x* also sowohl eine Kontur- als auch eine Klassenunschärfe. Eine Variablenkontur *y* mit konkretem Typ *value(y)* = $\{A^{\textcircled{1}}, A^{\textcircled{2}}\}$ ist zwar auch unscharf bezüglich $p_{contour}$, da $|p_{contour}(y)| = |\{A^{\textcircled{1}}, A^{\textcircled{2}}\}| > 1$ ist, aber scharf bezüglich p_{class} da $|p_{class}(y)| = |\{A\}| = 1$.

Zur Spezifikation einer Unschärfe gehört also immer die betroffene Variablenkontur *vc* und die charakteristische Funktion *p* der Unschärfe. Eine Unschärfe wird daher als Paar $(vc, p) \in VC \times (VC \rightarrow (C \cup CC))$ angegeben.

Die Unschärfe einer Variablenkontur *vc* pflanzt sich auf alle Variablenkonturen $\{vc' \mid (vc, vc') \in E\}$ fort, sofern die zugehörigen Kanten keinen Beschränkungen unterliegen. Als Ergebnis der Analyse sind wir an einem Programm mit statischem Kontrollfluß interessiert. Die Analyse versucht deshalb nicht alle Unschärfen in den Datenflußknoten des Programms aufzulösen, sondern nur solche, die zu einem mehrdeutigen Kontrollfluß führen. Solche Unschärfen nennen wir daher *wahrnehmbare* Unschärfen. Wahrnehmbare Unschärfen sind also Argumente von Methodenaufrufen, deren Werte für die Mehrdeutigkeit dieses Aufrufs verantwortlich sind. nehmen

2.9.1 Wahrnehmbare Unschärfen

I_V sei die Menge der wahrnehmbaren Unschärfen. Eine Unschärfe *(vc, p)* ist ein Element von I_V, wenn *vc* das Argument eines mehrdeutigen Methodenaufrufs $apply(v_r, v_0, g, v_1, \ldots, v_n)$ in einer Methodenkontur f*c* ist und die Werte *values(vc)* für die Mehrdeutigkeit dieses Aufrufs verantwortlich sind.

$\exists$ i $\in$ {0,…,n}: vc = v_i^{fc} und value(vc) verantwortlich für die Mehrdeutigkeit im Kontrollfluß

Die charakteristische Funktion *p* der Unschärfe hängt dabei von den Werten des Arguments *value(vc)* und den zugehörigen Beschränkungen der Parameter der selektierten Methodenkonturen ab. Im Beispiel aus Abbildung 18 ist *x* das Argument des unscharfen Methodenaufrufs mit den beiden selektierten Methodenkonturen $g^{①}$ und $g^{②}$. Die Werte *values(x)* = $\{A^{\textcircled{1}}, A^{\textcircled{2}}, B^{\textcircled{1}}\}$ des Arguments werden durch die Beschränkungen r_A und r_B auf die beiden Konturen verteilt. Werte von *x*, die nicht zur Beschränkung eines entsprechenden Methodenkonturparameters passen, sind verantwortlich für die Selektion mehrerer Konturen und damit für die Kontrollflußunschärfe. In *x* handelt es sich daher um eine wahrnehmbare Unschärfe. Die charakteristische Funktion der wahrnehmbaren Unschärfe in *x* bestimmt man wie folgt: $A^{\textcircled{1}}$ und $A^{\textcircled{2}}$ passen nicht zu r_B, $B^{\textcircled{1}}$ nicht zu r_A. Das Auflösen der Klassenunschärfe in *x* würde die Ursache für die Selektion mehrerer Konturen beseitigen. Enthielte *x* nur

die Werte $\{A^{❶}, A^{❷}\}$, würde $g^{①}$, bei $\{B^{❶}\}$ würde $g^{②}$ zur Behandlung ausreichen. Daher ist die wahrnehmbare Unschärfe in *x* eine Klassenunschärfe (x, p_{class}). Zwar gibt es in *x* auch eine Konturunschärfe $(x, p_{contour})$, jedoch ist sie nicht für die Selektion der unterschiedlichen Methodenkonturen $g^{①}$ und $g^{②}$ verantwortlich und daher aus Sicht des Kontrollflusses nicht wahrnehmbar.

Aus einem Wert $cc \in values(vc)$ und einer Beschränkung *r* des entsprechenden Parameters einer selektierten Methodenkontur läßt sich der Typ *p* der wahrnehmbaren Unschärfe ablesen. Erhält man für unterschiedliche Werte als Ergebnis sowohl eine Klassen- als auch eine Konturunschärfe, genügt es die Konturunschärfe aufzulösen, da damit auch die Klassenunschärfe beseitigt wird.

r(cc) = true	⇒	keine Unschärfe
$r = r_A$, $cc = B^m$	⇒	Klassenunschärfe p_{class}
$r = r_{A^n}$, $cc = B^m$	⇒	Klassenunschärfe p_{class}
$r = r_{A^n}$, $cc = A^m$, $m \neq n$	⇒	Konturunschärfe $p_{contour}$

2.9.2 Konfluenzen

Unschärfen lassen sich im allgemeinen nur an Stellen auflösen, an denen sie entstehen. Eine Unschärfe entsteht in einer Variablenkontur, wenn mehrere Datenflußkanten in diese Kontur münden und inkompatible Datenflußwerte in die Zielkontur transportieren. Sind die von jeder Kante abgelieferten Werte scharf bezüglich *p*, so heißt ein solcher Knoten eine *Konfluenz* bezüglich *p*. Zwei Datenflußwerte cc_1 und cc_2 heißen inkompatibel bezüglich *p*, wenn $p(cc_2) \neq p(cc_2)$.

Die unscharfe Variablenkontur *x* aus Abbildung 18 bildet, wie oben ausgeführt, eine wahrnehmbare Unschärfe (x, p_{class}). Betrachtet man eine mögliche Umgebung im Datenflußgraphen, wie in Abbildung 19 dargestellt, erkennt man, daß die Unschärfe in *x* nicht direkt aufgelöst werden kann. Die Variablenkontur *x* erhält alle ihre unscharfen Werte über eine einzelne Datenflußkante von *y*. Erst in *y* entsteht die Unschärfe, da nach *y* Werte aus unterschiedlichen Variablenkonturen y_1 und y_2 fließen, die inkompatibel bezüglich p_{class} sind. Gleichzeitig sind die Werte in y_1 und y_2 scharf bezüglich *p*, so daß es sich bei *y* um eine Konfluenz handelt.

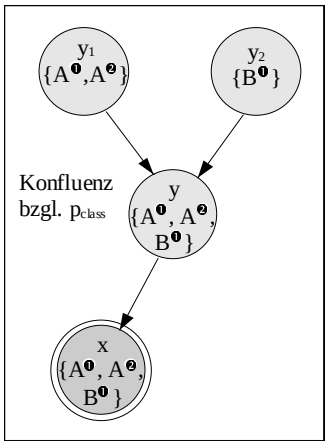

Abbildung 19
Konfluenz

Da die Stellen der Entstehung I_S von Unschärfen und die Menge I_V der wahrnehmbaren Unschärfen wie gesehen meist nicht zusammenfallen, müssen die wahrnehmbaren Unschärfen zu den Stellen ihrer Entstehung im Datenflußgraphen zurückverfolgt werden. Erst dann kann die Analyse versuchen, sie aufzulösen. Man verfolgt eine Unschärfe zu ihren Ursachen zurück, indem man ausgehend von der wahrnehmbaren Unschärfe *(vc, p)* die Kanten im Datenflußgraphen entgegen der Flußrichtung der Datenflußwerte zurückverfolgt bis man auf eine Konfluenz bezüglich *p* stößt.

2.9.3 Traversierung des Datenflußgraphen

Zur Zurückverfolgung von Werten im Datenflußgraphen dienen die Mengen *flow()* und *back()*. Für eine Variablenkontur *vc* enthält *back(vc)* alle Variablenkonturen, von denen aus Kanten auf *vc* zeigen. *flow(vc)* sammelt all diejenigen Variablenkonturen, die Ziel von Kanten ausgehend von *vc* sind.

$\forall$ vc $\in$ VC:
flow(vc) = {vc' | (vc, vc') $\in$ E}
back(vc) = {vc' | (vc', vc) $\in$ E}

Die einzigen bisher vorkommenden Datenflußwerte waren Klassenkonturen bzw. deren Klassen. Sie fließen von den Erzeugungspunkten vorwärts (in Richtung der Kanten) durch den Datenflußgraphen. Im Verlauf der Analyse wird allerdings noch eine weitere Kategorie von Datenflußwerten eingeführt, die in Rückwärtsrichtung durch den Datenflußgraphen fließen. Um die Auflösung von Unschärfen unabhängig von der Art der Datenflußwerte zu halten, die zu der Unschärfe führen, werden zur Traversierung des Graphen weitere Mengen eingeführt, die mit der charakteristischen Funktion der zu verfolgenden Unschärfe parametrisiert sind. Die charakteristische Funktion der weiter unten noch eingeführten rückwärts fließenden Datenflußwerte sei p_{path}.

$$\forall\, vc \in VC:$$
$$flow_p(vc) = \{vc' \mid (vc, vc') \in E_p\}$$
$$back_p(vc) = \{vc' \mid (vc', vc) \in E_p\}$$
$$\text{mit} \quad E_{p_{class}} = E_{p_{contour}} = E,$$
$$E_{p_{path}} = \{(vc, vc') \mid (vc', vc) \in E\}, \text{ die Kanten des inversen Datenflußgraphen}$$

Mit Hilfe dieser Mengen kann jetzt die Menge der Konfluenzen *(vc, p)* unabhängig von der Art der Unschärfe *p* angegeben werden.

$$(vc, p) \in I_S \Leftrightarrow \mid p(vc) \mid > 1 \wedge \forall\, vc' \in back_p(vc): \mid p(vc') \mid \leqslant 1$$

Eine Variablenkontur *vc* gehört also zur Menge der Konfluenzen mit charakteristischer Funktion *p*, wenn *vc* unscharf bezüglich *p* ist und alle Variablenkonturen *vc'* aus denen Werte nach *vc* fließen, scharf bezüglich *p* sind. Ist die letzte Bedingung für ein *vc'* nicht erfüllt, so gibt es eine frühere Ursache für die Unschärfe in *vc*.

Die Mengen *back(.)* für die Variablenkonturen aus Abbildung 19 sind die folgenden:

$back_{class}(x) = \{y\}$ $\qquad$ $p_{class}(x) = p_{class}(y) = \{A, B\}$

$back_{class}(y) = \{y_1, y_2\}$ $\qquad$ $p_{class}(y_1) = \{A\}, p_{class}(y_2) = \{B\}$

Es gilt zwar für *x* und *y*: $|p_{class}(x)| = |p_{class}(y)| = |\{A, B\}| > 1$, da aber $y \in back_{class}(x)$ und $|p_{class}(y)| > 1$, ist (x, p_{class}) keine Konfluenz. Anders verhält es sich bei *y*. Die Menge $p_{class}(y) = \{A, B\}$ ist mehrelementig, nicht aber die Mengen $p_{class}(y_1)$ und $p_{class}(y_2)$ ihrer Vorgänger aus $back_{class}(y)$. Damit ist *y* die Konfluenz, die die wahrnehmbare Klassenunschärfe in *x* verursacht. Die eine wahrnehmbare Unschärfe verursachenden Konfluenzen findet man durch Zurückverfolgen der Unschärfe im Datenflußgraphen. Dies geschieht, wie im nächsten Abschnitt beschrieben, über die Mengen $back_p(.)$.

2.10 Auflösen von Unschärfen

Die Menge I_V der wahrnehmbaren Unschärfen kann während der Berechnung des Datenflußgraphen mitbestimmt werden. Sie enthält all diejenigen Variablenkonturen, deren Werte einen mehrdeutigen interprozeduralen Kontrollfluß verursachen (vgl. 2.9.1).

2.10.1 Quellen einer Unschärfe

Um eine Unschärfe $(vi, p) \in I_V$ aufzulösen, muß sie zum Ort ihrer Entstehung zurückverfolgt werden. Die Quellen *sources(vi, p)* einer wahrnehmbaren Unschärfe *(vi, p)* sind diejenigen Konfluenzen aus $I_S(p)$, deren unscharfe Werte nach *vi* fließen. Die Datenflußwerte einer Konfluenz *vk* gelangen genau dann nach *vi*, wenn *vk* in der rekursiv transitiven Hülle von $back_p(vi)$ liegt. Man versucht nur solche Konfluenzen zu beseitigen, da nur wahrnehmbare Unschärfen aufgelöst werden sollen.

$sources(vi, p) = back_p^*(vi) \cap I_s(p)$ Quellen der Unschärfe (vi, p)

mit $I_s(p) = \{vc \mid (vc, p) \in I_s\}$ der Menge der unscharfen Variablenkonturen bezüglich p

und $vc'' \in back_p^*(vc) \Leftrightarrow (vc'' = vc) \vee \exists\, vc' \in back_p^*(vc)\colon vc'' \in back_p(vc')$

der reflexiv transitiven Hülle von $back_p(.)$

Für jede Konfluenz *(vc, p)* mit *vc ∈ sources(vi, p)* wird daraufhin versucht, die Unschärfe bezüglich *p* aufzulösen. Die nächsten Abschnitte beschreiben alle möglichen Konstellationen von auflösbaren Unschärfen, die Voraussetzungen für ihre Elimination und die dazu notwendige Transformation.

Zum Auflösen einer Unschärfe sind die sie produzierenden Datenflußkanten alleine nicht ausreichend. Zu jeder Kante benötigt man den Programmkontext, aus dem sie stammt. Daher wurden in 2.7 an jede Kante *(vc, vc') ∈ E* die sie erzeugenden Anweisungen *cmds(vc, vc')* annotiert. Zu einer Unschärfe *(vc, p)* kann man damit die Anweisungen $cmds_{vc,\,p}$ angeben, welche für die in *vc* ankommenden unscharfen Datenflußwerte verantwortlich sind.

$cmds_{vc,\,p} := \cup_{vc' \in back_p(vc)}\ cmds_p(vc', vc)$

mit $cmds_{p_{class}}(.,.) = cmds_{p_{class}}(.,.) = cmds(.,.)$

und $cmds_{p_{path}}(a, b) = cmds(b, a)$

2.10.2 Konfluenz in einer Instanzvariablen

Bei der Auflösung einer Konfluenz $(A^c.b, p)$ mit $A^c.b$ der Kontur einer Instanzvariablen *A.b* und $p \in \{p_{class}, p_{contour}\}$ haben alle Anweisungen $(fc, cmd) \in cmds_{A^c.b,\,p}$ die Form $cmd = store(l_1, b, l_2)$, da Zuweisungen an Instanzvariablen ausschließlich über (store)-Anweisungen erfolgen. Eine Anweisung $(fc, store(l_1, b, l_2))$ kann, wie in 2.8.5 besprochen, mehrere Datenflußkanten erzeugen, je nach Anzahl der Klassenkonturen im Parameter l_1^{fc}. Die Menge

$targets(fc, store(l_1, b, l_2)) := \{A^d.b \mid A^d \in value(l_1^{fc})\}$

der durch die Anweisung betroffenen Instanzvariablenkonturen, kann daher mehrelementig sein. Ist dies für eine Anweisung aus $cmds_{A^c.b,\,p}$ der Fall, ist dies eine erste Ursache für die Unschärfe in $A^c.b$. Es werden nämlich Werte an die Variablenkonturen in *tagets(.,.)* verteilt, wo eine genauere Differenzierung möglich wäre. Eine solche Situation ist in Abbildung 20 dargestellt. Der Inhalt von l_2^{fc} wird an die beiden Instanzvariablenkonturen $A^c.b$ und $A^2.b$ weitergeleitet. $A^c.b$ erhält aber auch noch von anderer Seite eine Zuweisung eines Datenflußwertes. Ein erster Ansatz zur Auflösung einer Unschärfe in $A^c.b$ ist die Auflösung der Konturunschärfe in l_1^{fc}. Nach Auflösung der Konturunschärfe $p_{contour}$ in l_1^{fc} erfolgt die betrachtete Zuweisung entweder nur an $A^c.b$ oder $A^2.b$.

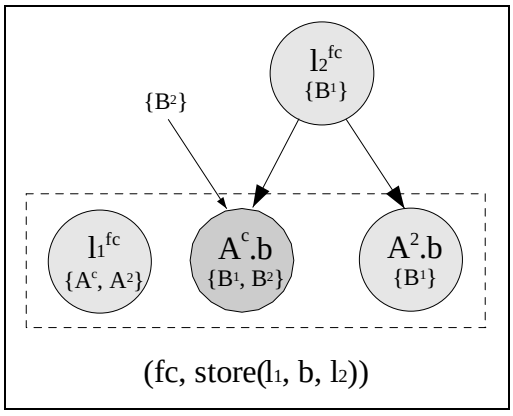

Abbildung 20
Unscharfes Ziel einer (store)-Anweisung

Ist l_1^{fc} scharf, liegt dem Auflösen der Unschärfe $(A^c.b, p)$ folgende Überlegung zugrunde: Die Werte, die über die Kanten nach $A^c.b$ fließen, können nicht verändert werden. Um die Unschärfe in der Instanzvariablenkontur $A^c.b$ aufzulösen, müssen alle Zuweisungen (von überall her im Pro-

gramm), die inkompatible Werte in $A^c.b$ abliefern, so verändert werden, daß sie unterschiedliche Konturen der zu $A^c.b$ gehörenden Instanzvariable $A.b = var(A^c.b)$ ansprechen.

Abbildung 21 ist ein Beispiel für die vorliegende Situation. Drei Anweisungen liefern Werte in der Instanzvariablen $A^c.b$ ab; wir nehmen an, daß wir eine Klassenunschärfe $p = p_{class}$ auflösen wollen. Das geht nicht aus der Abbildung hervor, da es sich hier nicht um eine wahrnehmbare Unschärfe handelt, sondern um eine aufzulösende Konfluenz, auf die man beim Zurückverfolgen einer wahrnehmbaren Unschärfe gestoßen ist. Die beiden Anweisungen *1) und *2) liefern kompatible (in diesem Fall sogar identische) Werte in $A^c.b$ ab, die dritte Anweisung liefert dagegen einen Wert, der zu den beiden ersten inkompatibel ist. Will man die Konfluenz der beiden Werte aus Anweisung *1), *2) und Anweisung *3) verhindern, so muß dafür gesorgt werden, daß in der Anweisung *3) eine andere Kontur der Instanzvariablen $A^c.b$ selektiert wird, als in den beiden anderen Anweisungen *1) und *2).

Dies erreicht man nur, indem man die Klassenkontur, die sowohl in den Variablen l_1, l'_1 der Anweisungen *1) und *2) als auch in der Variablen l''_1 der Anweisung *3) gespeichert ist, in zwei Konturen A^c und A^n teilt. Danach soll z.B. A^c in l_1 und l'_1 stehen und A^n in l''_1.

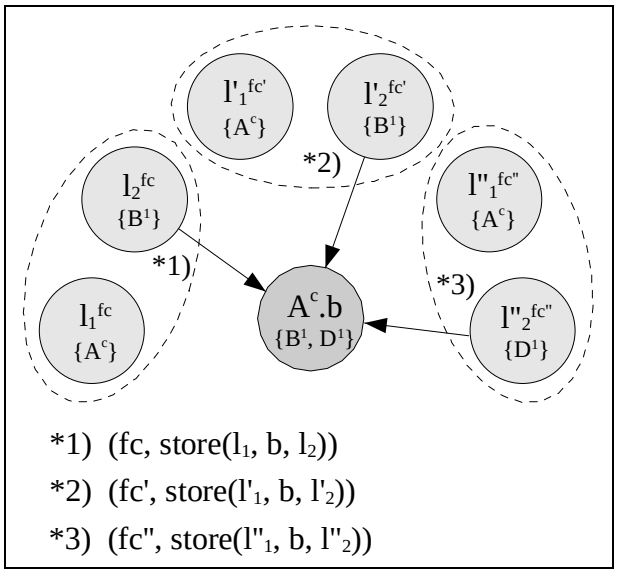

Abbildung 21
Konfluenz in Instanzvariablen

Abbildung 22 beschreibt das angestrebte Resultat einer solchen Teilung. In den Anweisungen **1)* und **2)* wird die Kontur $A^c.b$, in der Anweisung **3)* die Kontur $A^n.b$ der Instanzvariablen selektiert. Die inkompatiblen Datenflußwerte aus den beiden Gruppen von Anweisungen mischen sich nicht mehr in derselben Instanzvariablenkontur.

Die direkte Manipulation der Werte in den Variablenkonturen *targets(fc, cmd)* ist aber gleichfalls nicht möglich. Diese Werte werden ja auch von anderen Anweisungen dort abgeliefert. Abbildung 23 vermittelt einen Eindruck von einer möglichen Umgebung des Datenflußgraphen. Die einzigen Knoten im Datenflußgraphen, deren Werte direkt manipulierbar sind, sind die Variablenkonturen von new-Anweisungen. Sie haben keine Vorgänger im Datenflußgraphen, die andere Werte in sie hinein fließen lassen. Die initiale Klassenkontur, welche von einer new-Anweisung erzeugt wird, kann beliebig verändert werden. Eine new-Anweisung erzeugt immer eine Kontur derjenigen Klasse, die sie zur Laufzeit instanziert (vgl. 2.8.1). Welche Kontur dieser Klasse

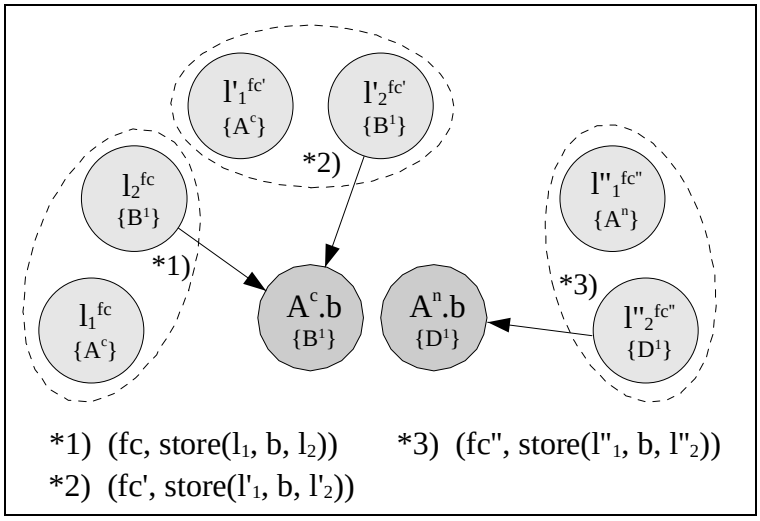

Abbildung 22
Aufgelöste Konfluenz in der Instanzvariablen

die new-Anweisung aber als initiale Kontur in ihre Variablenkontur (Erzeugungspunkt) speichert, spielt keine Rolle, diese Kontur ist der Freiheitsgrad, den die Analyse zur Auflösung von Unschärfen in Instanzvariablen ausnutzen kann.

Da die Klassenkonturen nur an ihren Erzeugungspunkten verändert werden können, sucht man jetzt Pfade im inversen Datenflußgraphen von den Behältern l_1^{fc}, l'_1^{fc}, l''_1^{fc} zurück zu den Erzeugungspunkten der Behälter-Klassenkontur A^c, um diese dort in mehrere Konturen zu teilen. Dafür werden zuerst die Anweisungen $cmds_{A^c.b,\ p}$, welche Werte v in $A^c.b$ abliefern nach gleichen Werten von $p(v)$ partitioniert. Anweisungen, die gemäß p inkompatible Werte abliefern, werden dabei unterschiedlichen Partitionen zugeordnet.

In Abbildung 21 liefern die Anweisungen **1)* und **2)* kompatible Werte ab und gehören daher derselben Partition an. Die andere Partition besteht nur aus der Anweisung **3)*.

Die Menge der partitionierten Anweisungen sei *cmds*.

$$cmds := partition_p(cmds_{A^c.b,\ p}) :=$$
$$\{\{\ (fc', store(l'_1, a, l'_2)) \in cmds_{A^c.b,\ p} \mid p(l'_2{}^{fc'}) = p(l_2{}^{fc})\ \} \mid (fc, store(l_1, a, l_2)) \in cmds_{A^c.b,\ p}\}$$

es gilt:

$\cup_{set \in cmds}\ set = cmds_{VC}$, $\forall\ set, set' \in cmds: set \cap set' = \varnothing$,

$\forall\ set \in cmds: \forall\ (fc, store(l_1, a, l_2)), (fc', store(l_1', a, l_2'))\ \in set: p(l_2{}^{fc}) = p(l_2{}'^{fc'})$

Für jede Partition *set ∈ cmds* bildet man jetzt die Menge $dest_{set}$ der Variablenkonturen, von welchen die Instanzvariable selektiert wird. Dies sind die Variablenkonturen, deren Werte verändert werden sollen, um die gewünschte Selektion der Konturen der Instanzvariablen zu erreichen. Durch die Partitionierung der Anweisungen sind auch sie in Partitionen unterteilt. Jeder dieser Partitionen soll

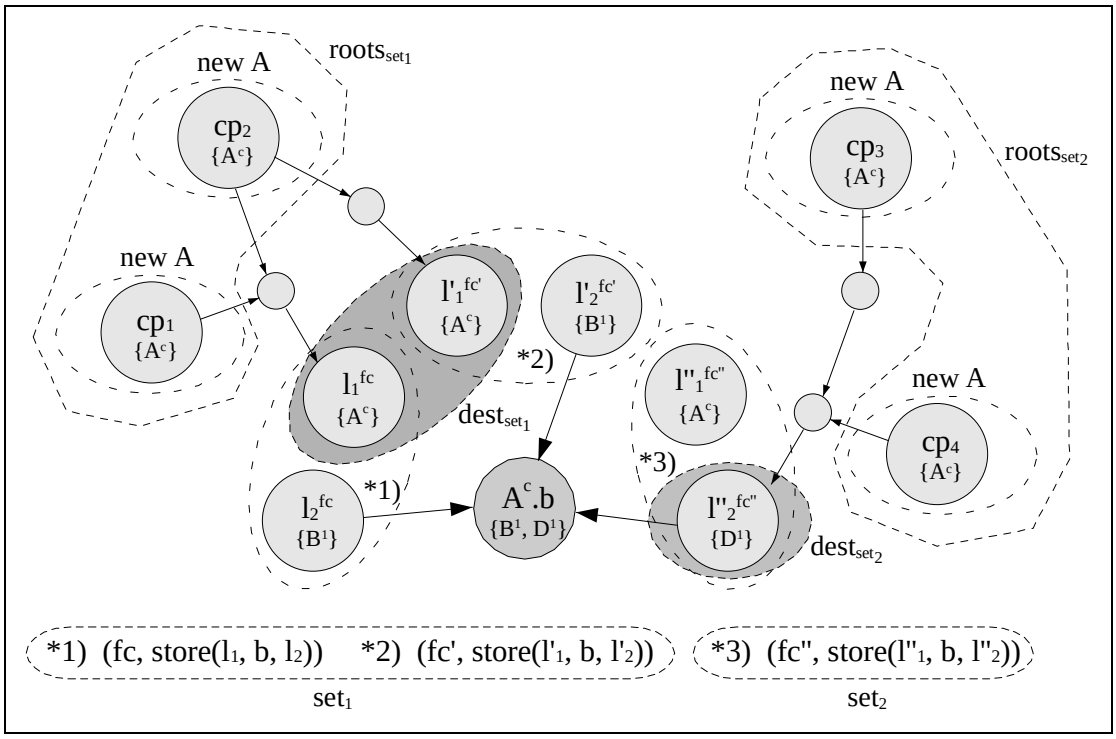

Abbildung 23
Konfluenz in Instanzvariable mit angedeuteter Umgebung

eine neue Kontur der Behälter-Kontur A^C zugewiesen werden.

Im Beispiel aus Abbildung 21 wäre *cmds = {set₁, set₂}* mit *set₁ = {*1), *2)}* und *set₂ = {*3)}*, $dest_{set_1} = \{l_1, l'_1\}$ und $dest_{set_2} = \{l''_1\}$. Alle Anweisungen in einer Partition *set ∈ cmds* liefern kompatible Datenflußwerte in der Instanzvariable $A^C.b$ ab.

$\forall$ set $\in$ cmds:
$\quad dest_{set} := \{l_1^{fc} \mid (fc, store(l_1, a, l_2)) \in set\}$

Die Mengen set_1 und set_2 der inkompatiblen Anweisungen und die Mengen $dest_{set_1}$, $dest_{set_2}$ der Variablenkonturen, deren Werte verändert werden sollen, sind in Abbildung 23 noch einmal graphisch dargestellt. Zusätzlich ist die Umgebung der inkompatiblen Zuweisungen im Datenflußgraphen skizziert. In den Erzeugungspunkten cp_1, cp_2, cp_3 und cp_4 entsteht die Klassenkontur A^C. Sie fließt von dort in die Variablenkonturen aus $dest_{set_i}$, wo von ihr in den Zuweisungen *1)…*3) die Instanzvariable $A^C.b$ selektiert wird.

Die Pfade, auf denen die Klassenkontur A^C von ihren Erzeugungspunkten in die Variablenkonturen aus $dest_{set_i}$ fließt, sind ebenfalls angedeutet. Verfolgt man nun die Variablenkonturen aus $dest_{set_i}$ zurück zu ihren Erzeugungspunkten $root_{set_i}$, indem man entgegen der Kantenrichtung des Datenflußgraphen wandert, erhält man für jede Menge $dest_{set_i}$ einen Mehrfachpfad $path_{set_i}$.

Kreuzen sich diese Pfade, wie in Abbildung 23 dargestellt, nicht, so ergibt sich aus der Partition der Variablenkonturen *targets(fc, cmd)* in die Mengen $dest_{set_i}$ auf natürliche Weise eine Partition der Erzeugungspunkte in die Mengen $roots_{set_i}$. Die Klassenkontur A^C kann dann in zwei Konturen geteilt werden, die eine A^C und die andere eine neue Kontur A^n. Dies geschieht, indem der initiale Datenflußwert A^C in einer der Mengen $roots_{set_i}$ gegen die neue Kontur A^n ausgetauscht wird. Dies bewirkt dann nach Neuberechnung der Datenflußwerte das gewünschte Ergebnis, daß die inkompatiblen Zuweisungsoperationen aus set_1 und set_2 auf unterschiedliche Konturen der Instanzvariable A.b zugreifen. Dieses Resultat wurde schon in Abbildung 22 dargestellt.

Die Pfade, die von jeder Menge $dest_{set}$ zu den Erzeugungspunkten der Klassenkontur A^C führen erhält man rekursiv aus:

$\forall$ set $\in$ cmds: $dest_{set} \subseteq path_{set}$,
$\forall$ vc' $\in path_{set}$: $back_p(vc') \subseteq path_{set}$

Die Behälter-Variablenkonturen aus $dest_{set_i}$ bilden die Anfangspunkte eines zu dieser Partition gehörenden Pfades $path_{set_i}$. Mit jeder Variablenkontur *vc'* des Pfades sind auch alle Variablenkonturen $back_p(vc')$ Teilmenge dieses Pfades.

Variablenkonturen $cp \in path_{set_i}$, die Ziele einer (new)-Anweisung sind, bilden die Menge $roots_{set_i}$ der Erzeugungspunkte auf diesem Pfad. Ihr initialer Datenflußwert soll gegen die geteilte Klassenkontur des Behälters ausgetauscht werden. Das ist nur möglich, wenn sich die Pfade nicht schneiden und damit diese Mengen $roots_{set_i}$ eine Partition der Erzeugungspunkte der Klassenkontur A^C bilden.

$\forall$ set $\in$ cmds:
$\quad roots_{set} = \{vc' \in path_{set} \mid isCreationPoint(vc')\}$
$\qquad$ mit $\quad isCreationPoint(vc') = \exists\, g \in F: \exists\, gc \in FC, fun(gc) = g:$
$\qquad\qquad \exists\, new(cp, A) \in code(f): vc' = cp^{gc}$

Die oben getroffene Annahme, daß sich die Pfade $path_{set_i}$ nicht kreuzen und so eine eindeutige Zuordnung der Erzeugungspunkte zu einer der Mengen von Anweisungen $cmds_{set_j}$ existiert, muß nicht unbedingt erfüllt sein. Abbildung 24 zeigt eine realistischere Umgebung im Datenflußgraphen.

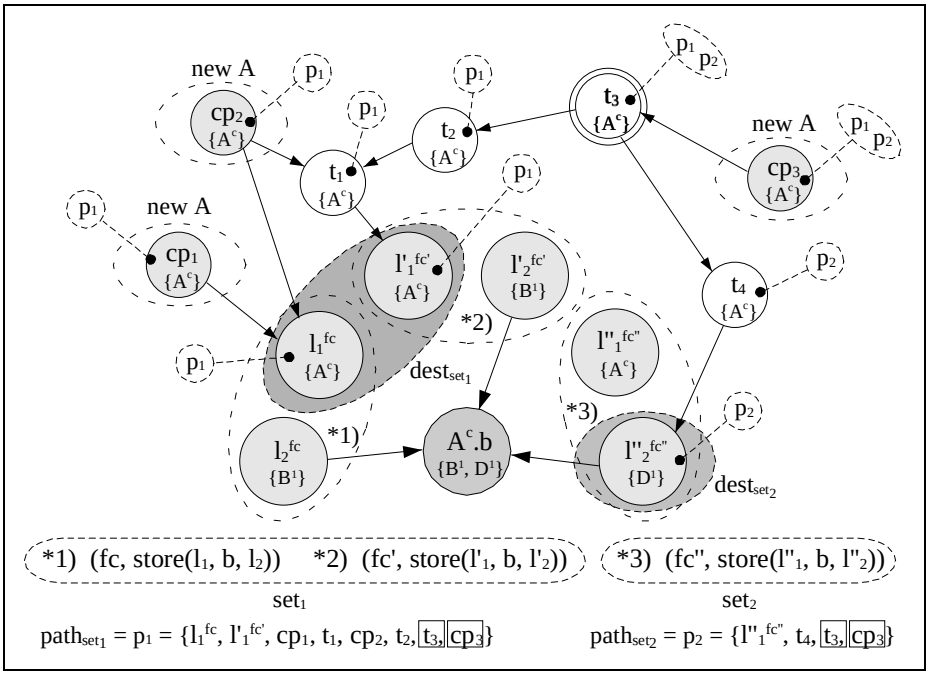

Abbildung 24
Konfluenz in Instanzvariable mit Umgebung

Die beiden Pfade $path_{set_1}$ und $path_{set_2}$ kreuzen sich in der Variablenkontur t_3. In den Variablenkonturen aus det_{set_1} und $dest_{set_2}$ sollen nach der Teilung der Klassenkontur A^c in zwei Konturen A^c und A^n unterschiedliche Konturen eintreffen. Für den Erzeugungspunkt cp_3 ist aber nicht klar, welche der geteilten Konturen A^c oder A^n in ihm entstehen sollen, da die dort entstehende Kontur sowohl nach $dest_{set_1}$ als auch nach des_{set_2} fließt. Die Bedingung, die vor dem Teilen der Klassenkontur A^C sichergestellt werden muß ist nicht erfüllt:

$$\forall \text{ set, set'} \in \text{cmds: path}_{\text{set}} \cap \text{path}_{\text{set'}} = \varnothing$$

Es muß eine sekundäre Datenflußanalyse mit anschließendem Teilungsschritt durchgeführt werden, um sie zu etablieren. In Abbildung 24 ist an jede beteiligte Variablenkontur die Menge der Pfade in kleinen gestrichelten Kreisen annotiert, die durch sie verlaufen. Der Pfad *path* ist an die Kontur *vc'* annotiert, wenn *path* durch *vc'* verläuft ($vc' \in path$). Mit *vc'* aus *path* sind aber auch alle $vc'' \in back_p(vc')$ aus *path*. Stellt man sich die Pfade selbst als Datenflußwerte vor, so sieht man, daß diese rückwärts durch den Datenflußgraphen fließen (von *vc'* nach $back_p(vc')$).

Unschärfen dieser sekundären Datenflußanalyse sind dann solche Variablenkonturen in denen sich Pfade kreuzen. An diese sind also mehrere Pfade annotiert. Die an eine Variablenkontur *vc'* annotierten Pfade seinen *paths(vc')*.

$$\text{paths(vc')} = \{\text{path}_{\text{set}} \mid \text{set} \in \text{cmds}, \text{vc'} \in \text{path}_{\text{set}}\}$$

Pfadunschärfen in Variablenkonturen verhindern die notwendige Teilung einer Klassenkontur. Sie müssen daher immer aufgelöst werden. Die charakteristische Funktion von Pfadunschärfen ist die schon zu Beginn des Kapitels erwähnte Funktion p_{path}. Kreuzen sich zwei Pfade in einer Instanzva-

riablen *vc'*, so ist die Menge *paths(vc')* mehrelementig und *vc'* ist unscharf bezüglich p_{path}. p_{path} bildet daher eine Variablenkontur *vc'* auf die Menge der durch sie verlaufenden Pfade *paths(vc')* ab.

$$p_{path}: VC \rightarrow PATH \quad \text{mit } PATH = \wp(\wp(VC))$$
$$p_{path}(vc') = path(vc')$$

Eine solche Pfadunschärfe tritt in Abbildung 24 in den Variablenkonturen t_3 und cp_3 auf: (t_3, p_{path}) und (cp_3, p_{path}). Bevor die Behälter-Klassenkontur A^c geteilt werden kann, müssen zuerst alle Pfadunschärfen aufgelöst werden. Die Auflösung von Pfadunschärfen läuft genauso ab wie die Auflösung aller anderen Unschärfen auch. Unter Verwendung der charakteristischen Funktion p_{path} für Pfadunschärfen sind rekursiv die Regeln zur Auflösung von Unschärfen dieses Kapitels anwendbar. Man beginnt mit der Suche nach konfluenten Variablenkonturen, indem man bei einer beliebigen Pfadunschärfe beginnend diese entgegen ihrer Flußrichtung im Datenflußgraphen zurückverfolgt.

Die Pfade fließen entgegen der Kantenrichtung, d.h. bei der Zurückverfolgung von Pfadunschärfen bewegt man sich in entgegengesetzter Flußrichtung, also entlang der Kanten des Datenflußgraphen. Beginnt man mit der Pfadunschärfe (cp_3, p_{path}), so findet man bereits nach einem Schritt die Konfluenz (t_3, p_{path}). Nach t_3 fließen (entgegen der Kantenrichtung) aus t_4 der Wert $\{p_2\}$ und aus t_2 der Wert $\{p_1\}$. Da diese beiden Variablenkonturen beide scharf bezüglich p_{path} sind, handelt es sich bei (t_3, p_{path}) um eine Konfluenz.

Die Auflösung dieser Konfluenz geschieht (rekursiv) mit einer Regel dieses Abschnitts 2.10 je nachdem, um welchen Typ von Variablenkontur es sich bei t_3 handelt. Nachdem diese Unschärfe aufgelöst ist, kann auch die Unschärfe in der Instanzvariablen $A^c.b$ durch Teilen ihres Behälters A^c, wie oben beschrieben, aufgelöst werden.

2.10.3 Konfluenz in einer Instanzvariablen bei Pfadunschärfen

Die Auflösung von Pfadunschärfen wird nötig als Lösung einer sekundären Unschärfe, die bei der Elimination von Konfluenzen in Instanzvariablen auftreten. Eine Pfadkonfluenz kann dabei ihrerseits auch in einer anderen Instanzvariablenkontur auftreten. Die Elimination von Pfadunschärfen in einer Instanzvariablenkontur läuft genauso ab wie die Elimination einer Unschärfe des Typs p_{class} oder $p_{contour}$, wenn man berücksichtigt, daß die Pfade in entgegengesetzter Richtung im Datenflußgraphen fließen.

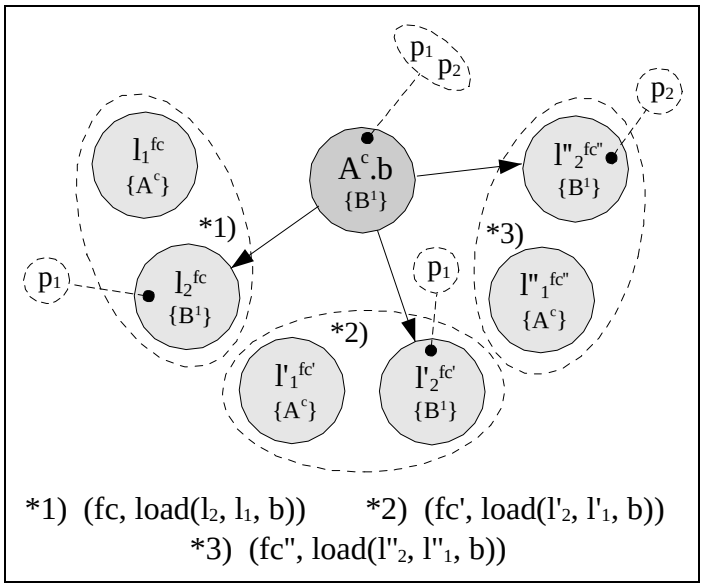

Abbildung 25
Pfadunschärfe in einer Instanzvariable

Abbildung 25 zeigt eine Pfadkonfluenz in einer Instanzvariablenkontur $A^c.b$, die Pfadmengen sind wieder in kleinen gestrichelten Kreisen an die Variablenkonturen annotiert. Der Fluß der Pfade entgegen der Kantenrichtung hat zur Folge, daß die Anweisungen, die zu den konfluenten Kanten $(A^c.b, l_2^{fc})$, $(A^c.b, l'_2^{fc'})$, $(A^c.b, l''_2^{fc''})$ gehören, keine (store)- sondern (load)-Anweisungen sind. Die Vorgehensweise zur Auflösung der Unschärfe ist jedoch identisch mit der Elimination von

Unschärfen p_{class} und $p_{contour}$ in Instanzvariablen, wie in 2.10.2 beschrieben. Die Variablen der (load)-Anweisung *'$l_1 = l_2.b$'* sind nur gegen die entsprechenden Variablen der (store)-Anweisung *'$l_1.b = l_2$'* auszutauschen. Bei (load) enthält nicht l_1 sondern l_2 den Behälter, dessen Instanzvariablenkontur selektiert wird und der daher geteilt werden muß. Wenn man in den Regeln aus 2.10.2 die Anweisung *store(l_1, b, l_2)* gegen die Anweisung *load(l_2, l_1, b)* vertauscht, kann man sie auf den hier vorliegenden Fall einer Pfadunschärfe in einer Instanzvariablen anwenden.

2.10.4 Konfluenz in einer lokalen Variablen

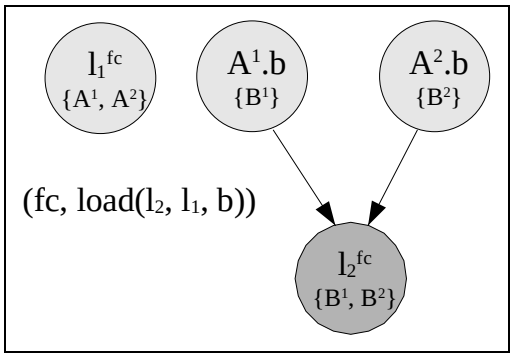

Abbildung 26
Konfluenz in lokaler Variable

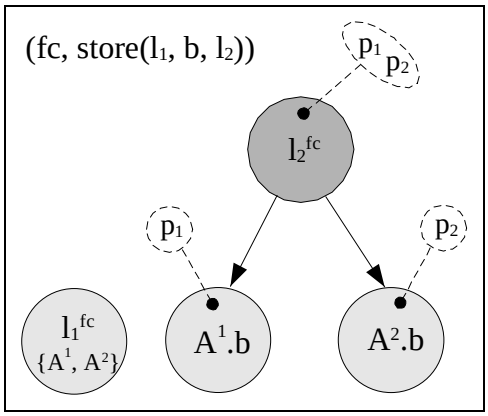

Abbildung 27
Pfadkonfluenz in lokaler Variable

So, wie es bei einer (store)-Anweisung durch einen unscharfen Behälter zu einer Unschärfe p_{class} oder $p_{contour}$ in einer Instanzvariablen kommen kann, ist dies auch bei einer (load)-Anweisung in einer lokalen Variablen einer Methode möglich. Abbildung 26 zeigt diesen Fall. Abbildung 27 gibt den symmetrischen Fall einer Pfadkonfluenz wieder. In beiden Fällen tritt die Konfluenz (l_2^{fc}, p) in einer lokalen Variablen auf. Um sie zu beseitigen, muß man die Unschärfe $(l_1^{fc}, p_{contour})$ in der Behältervariablen auflösen. Dies geschieht wieder rekursiv durch Suchen der Ursachen für die Unschärfe und Auflösen dieser mit einer der Regeln aus 2.10.

Sei (vc, p) Konfluenz in der Kontur einer lokalen Variablen
$\forall$ cmd $\in$ $cmds_{vc, p}$:
(cmd = load(vc, l_1, b)) $\Rightarrow$ löse Unschärfe (l_1, $p_{contour}$)
(cmd = store(l_1, b, vc)) $\Rightarrow$ löse Unschärfe (l_1, $p_{contour}$)

2.10.5 Konfluenz in einer Parametervariablen

In diesem Abschnitt wird die Auflösung einer Konfluenz (pv^{fc}, p) in einer Parametervariablen *pv*, $p \in \{p_{contour}\}$ behandelt. Parametervariablen einer Methodenkontur *fc* sind alle p_i aus $(p_0,\ldots, p_n) = params(fc)$. Eine Konfluenz p_{class} kann in einer solchen Variablen nicht auftreten, da beim Methodenaufruf Konturen immer so selektiert werden, daß ihre Parameter monomorph bleiben. Konfluenzen p_{path} können in Parameterkonturen nicht aufgelöst werden. Der symmetrische Fall für Pfadkonfluenzen wird in 2.10.6 beschrieben, hierbei handelt se sich dann um Ergebnisvariablen einer Methodenkontur.

Bei einer Konfluenz in der Kontur einer Parametervariablen mischen sich verschiedene Klassenkonturen, die in die Parameterkontur aus verschiedenen Aufrufen gelangen. Eine solche Konfluenz läßt sich direkt durch Verschärfen der Parametereinschränkungen der Methodenkontur auflösen. Für

jeden unscharfen Wert *cc* in der Parametervariablen wird eine neue Methodenkontur mit der Beschränkung r_{cc} für diesen Parameter angelegt. Abbildung 28 demonstriert diese Vorgehensweise.

$(p_0,\ldots, p_n)$ = params(fc)
$(r_0,\ldots, r_n)$ = restrict(fc)
Sei $i \in \{0\ldots n\}$ mit $p_i = pv^{fc}$

$\forall$ cc $\in p(p_i)$:
  Sei fc_{cc} eine neue Methodenkontur mit
    $restrict(fc_{cc}) = (r_0,\ldots,r_{i-1}, r_{cc}, r_{i+1},\ldots, r_n)$
    $fun(fc_{cc}) = fun(fc)$

$\forall$ cmd $\in cmds_{pv^{fc}}$: cmd = (gc, apply(l_r, l_0, f, $l_1,\ldots, l_n$))
  lösche fc aus selected(cmd)

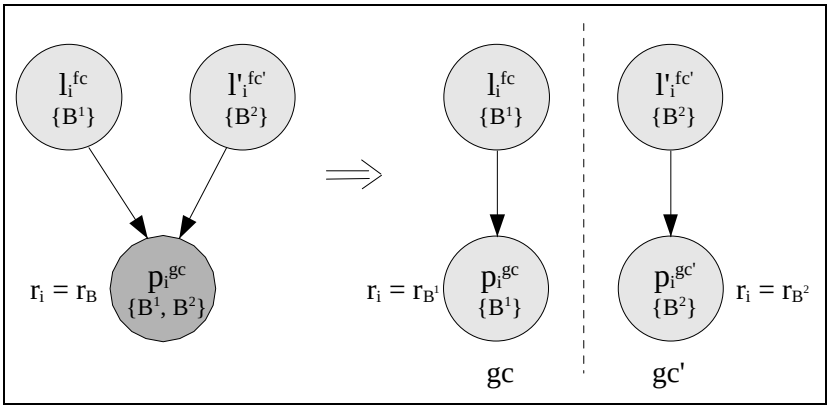

Abbildung 28
Konfluenz in einer Parametervariablen

Die alte Methodenkontur *fc* kann danach gelöscht werden. Beim nächsten Durchgang werden an den entsprechenden Methodenaufrufen automatisch die neuen Konturen *fc* selektiert, bei denen diese Unschärfe nicht mehr auftreten kann.

Die Situation in Abbildung 28 tritt beispielsweise ein, wenn eine Methode *g* in der Methodenkontur *fc* mit *g(…l_i,…)* und in der Kontur *fc'* mit *g(…l'_i,…)* aufgerufen wird. Dabei sollen in beiden Fällen dieselbe Kontur $g^{①}$ selektiert werden. Im Aufruf aus *fc* transportiert l_1^{fc} den Wert B^1 nach p_i, in *fc'* gelangt aus $l'_1{}^{fc'}$ der Wert B^2 in die Variablenkontur p_i des Parameters der Methodenkontur $g^{①}$. Nach dem Teilen von $g^{①}$ wird die Beschränkung r_B dieses Parametern zu r_{B^1} bzw. r_{B^2} verschärft, was für die beiden Aufrufe zur Selektion der neuen Konturen und zur Auflösung der Unschärfe $(p_i, p_{contour})$ führt.

2.10.6 Konfluenz in einer Ergebnisvariablen

Eine Unschärfe p_{path} kann in einer Parametervariablen nicht aufgelöst werden, da wegen der umgekehrten Flußrichtung der Pfade die Werte, welche die Unschärfe erzeugen, aus der Methodenkontur selbst stammen. Auch gibt es für Pfade keine Beschränkungen, die verschärft werden könnten. Für eine Pfadunschärfe ist die Ergebnisvariable einer Methodenkontur die Korrespondenz zur Parametervariablen.

Da keine Beschränkungen verschärft werden können, muß die Auflösung der Unschärfe etwas modifiziert werden: Man sucht ebenfalls alle zu der Unschärfe (vr, p_{path}) gehörenden Anweisungen

(fc, apply(l_r, l_0, g, l_1,… , l_n)) ∈ $cmds_{vr, p_{path}}$. Die zur Unschärfe führenden Kanten sind dann diejenigen, die die Werte aus der Ergebnisvariablen *vr* der Methodenkontur zurück in die Ergebnisvariable l_r des Aufrufs transportieren.

Die Unschärfe wird aufgelöst, indem man für jeden Wert *path ∈ paths(vr)* eine Kopie gc_{path} der ursprünglichen Methodenkontur *gc* anlegt. Da es sich dabei um identische Kopien handelt, kann hier nicht auf die normale Methodenkonturselektion bei der Berechnung des Datenflußgraphen zurückgegriffen werden. In den Mengen *selected(.)* der Aufrufanweisungen müssen die Methodenkonturen sofort ausgetauscht werden, damit bei der Neuberechnung der Datenflußwerte die unterschiedlichen Kopien von *gc* verwendet werden.

$$\forall\ cmd = (fc, apply(lr, l_0, g, l_1,\ldots, l_n)) \in cmds_{vr,\ p_{path}}: \{path\} = paths(l_r)$$
$$\text{tausche gc gegen } gc_{path} \text{ in selected(cmd)}$$

Ein Methodenaufruf trägt eine Pfadunschärfe nicht über die Parameter, sondern über die Ergebnisvariable in eine Methodenkontur. Das geschieht dann, wenn innerhalb der Methode ein Objekt erzeugt wird, das als Resultat zurückgeliefert wird und dessen Klasse in mehrere Konturen geteilt werden soll. Wird die Methode in verschiedenen Aufrufen verwendet, werden zur Laufzeit dort unterschiedliche Objekte erzeugt. Da für alle diese Aufrufe dieselbe Kontur selektiert wurde, sieht die Analyse vor der Teilung nur einen einzigen Erzeugungspunkt. Die verschiedenen Pfade zu diesem Erzeugungspunkt kreuzen sich in der Ergebnisvariable der Methodenkontur. Abbildung 29 zeigt die Situation für zwei Aufrufe der Methode *g*, für die dieselbe Kontur $g^{①}$ selektiert wurden. Ein vorangegangener Analyseschritt hat versucht, die Klassenkontur des von $g^{①}$ zurückgelieferten Resultats so zu teilen, daß Aufruf 1 und Aufruf 2 zwei verschiedene Konturen zurückliefern. Dies ist nicht möglich, da für *g* in beiden Aufrufen dieselbe Kontur $g^{①}$ selektiert wurde. Die beiden Pfade p_1 und p_2, die zur Teilung der Klassenkontur des Resultats von der Verwendung zurück zu der Stelle seiner Erzeugung gebildet wurden, kreuzen sich in *vr*, der Kontur der Ergebnisvariablen von $g^{①}$. Die Teilung der Klassenkontur des Resultats wird erst möglich, wenn für die beiden Aufrufe von *g* unterschiedliche Konturen $g^{①}$ und $g^{②}$ selektiert sind. Da sich die Pfade dann, wie in Abbildung 30 gezeigt, nicht mehr kreuzen; jede Methodenkontur hat ihre eigene Kontur der Ergebnisvariablen. Ansonsten sind die Konturen $g^{①}$ und $g^{②}$ identisch. Mit der Teilung der Methodenkonturen wurden auch zwei Versionen der new-Anweisung erzeugt, welche die zu teilende Klassenkontur der Resultats erzeugt. Der vorausgegangene Analyseschritt kann jetzt die Kontur des Resultats teilen, indem er den beiden new-Anweisungen unterschiedliche Klassenkonturen, wie in 2.10.2 beschrieben, als initiale Datenflußwerte zuweist.

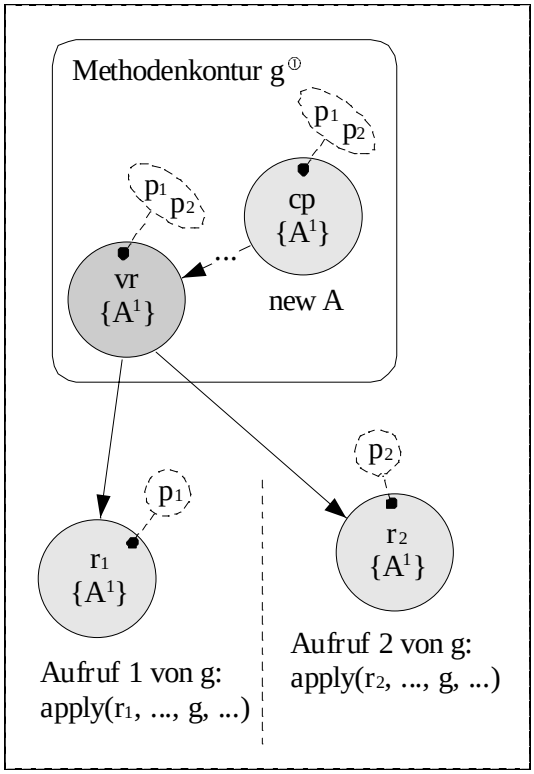

Abbildung 29
Pfade vor Teilung der Methodenkontur

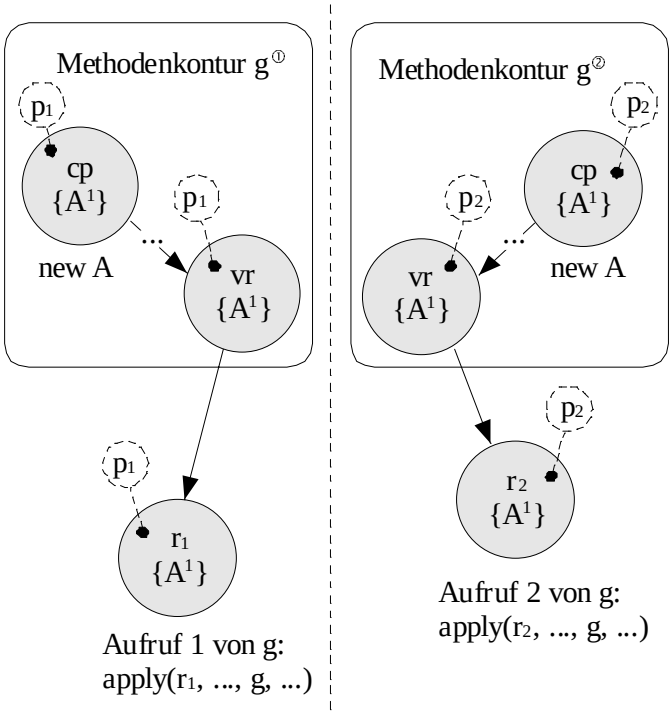

Abbildung 30
Nach der Teilung der Methodenkontur

2.11 Nicht auflösbare Unschärfen

Zwar ist ähnlich den realen Methodenaufrufen zur Laufzeit auch jedes zur Laufzeit instanzierte Objekt monomorph, d.h. die Objekte, welche in Instanzvariablen gespeichert sind, haben einen eindeutigen konkreten Typ. Der Analyse sind aber Grenzen gesetzt was das Auflösungsvermögen in bestimmten Strukturen betrifft. Speichert das Programm beispielsweise Objekte unterschiedlichen Typs in ein Feld, sieht das für die Analyse aus wie eine Speicheroperation in eine einzelne Komponente des Feldes. Alle Typen vermischen sich in ihr. Leseoperationen von dem Feld erhalten dann ebenfalls diesen polymorphen Typ. Das hat im weiteren Verlauf Kontrollflußunschärfen zur Folge, wenn Methoden aus dem Feld geladener Objekte aufgerufen werden. Ähnliches passiert wenn anstatt eines Feldes eine rekursive Struktur auftritt, die echt polymorph verwendet wird. Für eine Untersuchung der Beschränkungen der Typinferenz bei rekursiven Strukturen wird auf [Ple96] verwiesen.

Da alle Sorten von Fallunterscheidungen immer betrachtet werden, als ob alle Äste der Verzweigung ausgeführt würden, ergibt sich daraus eine weitere Quelle von unauflösbaren Unschärfen. Erhält eine einfache lokale Variable auf dem einen Ast einer Verzweigung den Typ *A*, auf dem anderen den Typ *B*, so ergibt sich daraus der resultierende polymorphe Typ *{A , B}*. Eine solche lokale Variable ist dann auch polymorph verwendet. Aus ihrem polymorphen Typ resultieren interprozeduralen Kontrollflußunschärfen, die auf statisch nicht entscheidbare Unschärfen im lokalen Kontrollfluß zurückzuführen sind. An solchen Stellen nützt der Programmierer Polymorphie, um explizite Fallunterscheidungen durch die dynamische Methodenauflösung der objektorientierten Sprache zu ersetzen. Solche Unschärfen kann und will die Analyse nicht auflösen.

Anders ist das bei monomorpher Verwendung von Strukturen in verschiedenen Kontexten mit unterschiedlichen Typen; hier wird Polymorphie nur zur Wiederverwendung von Code ausgenutzt. In solchen Fällen kann die Analyse die verschiedenen monomorphen Verwendungen polymorpher Strukturen trennen und damit eine genaue Typisierung und einen statischen Kontrollflußgraphen berechnen.

3 Kommunikation und Komplexität / Das JavaParty Modell

3.1 Aktivitäten

Genau wie in Java ist auch in JavaParty ein Thread (Aktivität) die kleinste Einheit von Parallelität. Ein Thread beginnt mit der Ausführung seiner run-Methode, der Wurzel der Aktivität. Alle von dieser Methode ausgehenden Aufrufe und Unteraufrufe anderer Methoden werden wie bei der Ausführung des Hauptprogramms synchron durchgeführt. D.h. die Abarbeitung der aufrufenden Methode wird erst dann fortgesetzt, wenn die aufgerufene Methode vollständig abgearbeitet ist. Ein Thread endet dann, wenn die Methode vollständig abgearbeitet ist, mit der er seine Ausführung begonnen hat. Mehrere Threads eines Java-Programms laufen (bis auf Synchronisation) weitgehend unabhängig voneinander wie mehrere einzelne Java-Programme ab mit dem Unterschied, daß alle diese Threads im selben Adreßraum ausgeführt werden und daher auf gemeinsame Variablen zugreifen können.

JavaParty ersetzt den Adreßraum einer virtuellen Maschine, der zur Ausführung eines Programms zur Verfügung steht, durch einen virtuellen Adreßraum, der die Adreßräume mehrerer virtueller Maschinen umfaßt. Objekte sind von überall in diesem verteilten Adreßraum her referenzierbar, wenn sie Instanzen einer Fernklasse sind. Fernklassen werden mit dem zusätzlichen Schlüsselwort *remote* deklariert und leiten damit automatisch die Klasse *RemoteObject* oder eine explizit angegebene andere Fernklasse ab. Indem Zugriffe und Methodenaufrufe von entfernten Objekten intern in Netzwerkkommunikation übersetzt werden, ergibt sich für den Programmierer die Illusion eines gemeinsamen Adreßraums.

Eine Aktivität in JavaParty ist repräsentiert durch die Klasse *jp.lang.RemoteThread*. Sie ist eine Fernklasse. Ihre Instanzen können im Gegensatz zur Klasse *java.lang.Thread* auf entfernten Prozessorknoten angelegt und von überall im verteilten Adreßraum her angesprochen werden. Soll ein Thread auf einer entfernten Maschine angelegt werden, ist das nicht direkt möglich, da die Klasse *java.lang.Thread* eine lokale Klasse ist. Diesen Mangel behebt die Klasse *RemoteThread*. Ihre Instanzen dienen als Steuerobjekt für einen lokalen Thread auf einer entfernten Maschine. Beim Anlegen einer Instanz von *RemoteThread* wird diese zuerst auf einem beliebigen Prozessorknoten erzeugt und legt ihrerseits auf diesem Prozessorknoten einen lokalen *Thread* der Klasse *java.lang.Thread* an. Von nun an leitet der *RemoteThread* nur noch die Aufrufe seiner Methoden an das lokale Threadobjekt weiter, wodurch sich der lokale Thread über entfernte Methodenaufrufe steuern läßt. So erklärt sich auch die Eigenschaft von Instanzen von *RemoteThread*, daß sie nach ihrer Erzeugung nicht mehr migrierbar sind. Sie enthalten eine Referenz auf ein lokales Threadobjekt, welches eine Aktivität auf einer lokalen Maschine repräsentiert. Diese ist nicht auf einen anderen Prozessorknoten verschiebbar.

Da nach der Erzeugung alle Operationen von *RemoteThread* an das lokale Threadobjekt delegiert werden, erklärt sich die übrige Funktionalität allein durch die Klasse *java.lang.Thread*. Der einzige Unterschied besteht darin, daß der kontrollierte Thread möglicherweise auf einer entfernten Maschine abläuft.

Wie oben erwähnt, kann eine Aktivität den Adreßraum, im dem sie einmal angelegt wurde, nicht mehr verlassen. Aktivität meint hier den Kontrollfluß des zu Grunde liegenden Java-Threads. Dieser ist an den Adreßraum der virtuellen Maschine gebunden, in dem der er angelegt wurde. Anders verhält es sich mit dem logischen Kontrollfluß eines JavaParty-Programms. Da JavaParty die bei einem entfernten Methodenaufruf entstehende Netzwerkkommunikation vor dem Programmierer verbirgt, sieht ein solcher entfernter Methodenaufruf von außen so aus, als ob die ausführende Aktivität wie bei einem normalen Methodenaufruf nicht wechselt. Der Schein trügt hier allerdings, JavaParty führt kein Konzept von adreßraumübergreifenden Threads ein. Entfernte Methodenaufrufe werden in einem eigenen Thread auf der entfernten Maschine ausgeführt. Aktivitäten werden immer auf Java-Threads abgebildet, auch wenn unter Verwendung eines *RemoteThread* die Wurzel der

Aktivität in einem entfernten Adreßraum liegt. Aus diesem Grund können beim Einsatz von Synchronisierungsmechanismen Probleme mit Verklemmungen auftreten, wenn bei der Synchronisation keine Rücksicht auf die Eigenschaften des entfernten Methodenaufrufs genommen wird. Diese Problematik soll hier allerdings nicht weiter vertieft werden, Details sind [Zen97] und [Phi97a] zu entnehmen.

3.2 Entfernter Methodenaufruf

Im weiteren soll näher untersucht werden, unter welchen Voraussetzungen der Einsatz von mehreren Aktivitäten einen Geschwindigkeitsvorteil gegenüber der seriellen Lösung erwarten läßt. Diese Überlegungen sollen dann in die Berechnung der Verteilungsstrategie für Objekte und Aktivitäten eingehen.

Zuerst wird ein einzelner entfernter Methodenaufruf betrachtet. Abbildung 31 vermittelt einen Eindruck davon, welche Schritte ablaufen, wenn eine Methode eines entfernten Objekts aufgerufen wird. Das aufrufende Objekt (symbolisiert durch den Kreis *S1* in Abbildung 31) besitzt keine direkte Java-Referenz des gerufenen Objekts, da sich dieses in einem anderen Adreßraum befindet. Java-Referenzen sind immer an ihre virtuelle Maschine gebunden. Statt einer direkten Referenz hält das aufrufende Objekt ein sog. Handleobjekt, welches das entfernte Objekt im Adreßraum des Aufrufers repräsentiert. Da das Handleobjekt alle Methoden des entfernten Objekts implementiert, kann der Aufrufer dieses behandeln wie das entfernte Objekt selbst. Die entsprechende Methode des Handleobjekts (*S2*) lokalisiert jetzt das entfernte Objekt im verteilten Adreßraum über Anfragen an die

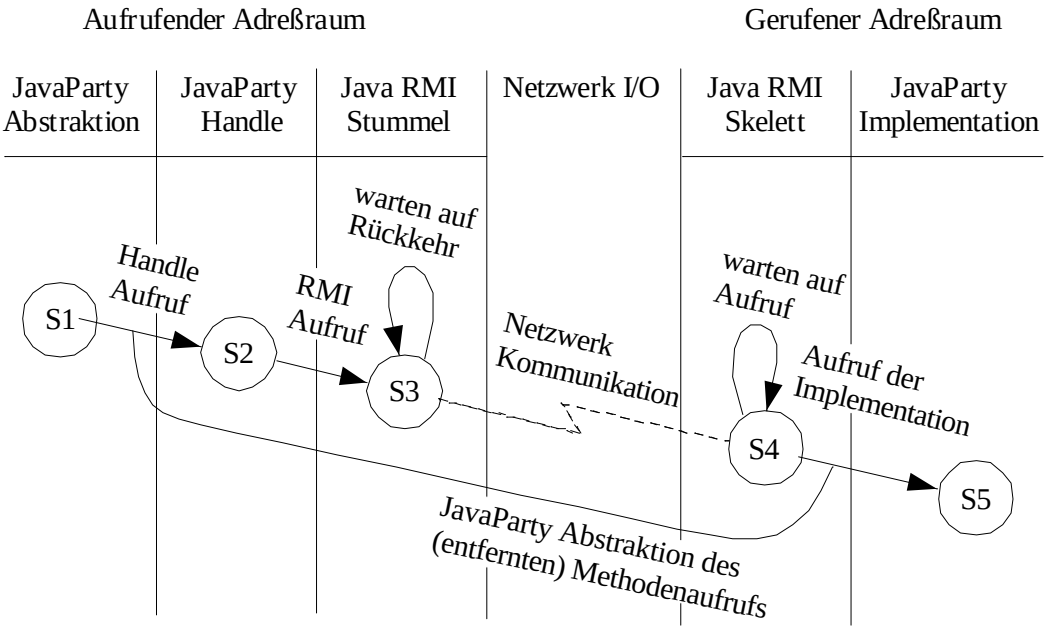

Abbildung 31
Entfernter Methodenaufruf

verteilte Laufzeitumgebung. Anschließend wird der Aufruf an die für die gerufene Klasse erzeugte sog. RMI-Stummelklasse (*S3*) weitergeleitet. Sie ist für das Serialisieren der Methodenparameter und die Netzwerkkommunikation mit dem RMI-Server zuständig. In der Stummelklasse wird daraufhin der aufrufende Thread solange blockiert, bis der entfernte Methodenaufruf zurückkehrt und das Resultat zurückgeschickt wird. Dies geschieht auch beim Aufruf einer Methode, die kein Resultat liefert. Entfernte Methodenaufrufe sind immer synchron. Im entfernten Adreßraum nimmt die sog. RMI-Skelettklasse (*S4*) den über das Netzwerk übermittelten Aufruf nebst dessen Parame-

tern entgegen. Die serialisierten Parameter werden wieder in Java-Objekte verwandelt und ein neuer Thread instanziert, der die Methode des entfernten Objekts (*S5*) ausführt. Mit dem Versenden des Resultats der Methode ist die Aktivität in dem entfernten Adreßraum abgeschlossen. Details zum RMI-Paket sind [Rmi97] zu entnehmen.

Abbildung 32 schematisiert den Ablauf eines einzelnen entfernten Methodenaufrufs. Es sind zwei virtuelle Maschinen *A* und *B* beteiligt. Jede Maschine sei mit der Ausführung einer Aktivität t_A bzw. t_B beschäftigt. Die Aktivität t_A auf der virtuellen Maschine *A* will eine Methode eines entfernten Objekts ausführen, das im Adreßraum der virtuellen Maschine *B* existiert. Die Aktivität auf Maschine *A* iniziiert den Aufruf entsprechend obigem Schema und geht in einen Wartezustand über. War diese Aktivität die einzige auf Maschine A, so gibt es zu diesem Zeitpunkt auf Maschine *A* keine ausführbare Aktivität mehr, Rechenzeit geht verloren. Auf Maschine *B* wird der Aufruf vom RMI-System entgegengenommen und zur Durchführung eine neue Aktivität t_B' instanziert. Die virtuelle Maschine *B* führt ab diesem Zeitpunkt zwei Aktivitäten t_B und t_B' gleichzeitig aus. Wenn man annimmt, daß die virtuelle Maschine *B* nur über einen realen Prozessor verfügt, muß die Rechenzeit dieses Prozessors auf diese beiden Aktivitäten aufgeteilt werden. Die Ausführung des Rumpfes der entfernt ausgeführten Methode dauert damit ungefähr doppelt so lange, wie wenn sie lokal auf Maschine *A* ausgeführt worden wäre. In Abbildung 32 ist die Aufteilung des realen Prozessors von Maschine *B* auf die beiden Aktivitäten t_B und t_B' durch unterschiedliche Dicken der die Aktivitäten symbolisierenden Striche angedeutet. Eine Aktivität verfügt nur dann über den realen Prozessor, wenn ihr Strich dick gezeichnet ist. Nur dann macht sie auch Fortschritte in ihrer Berechnung. Die Aneinanderreihung aller dick gezeichneten Teile einer Aktivität symbolisiert dann die zur Ausführung dieser Aktivität benötigte Prozessorzeit.

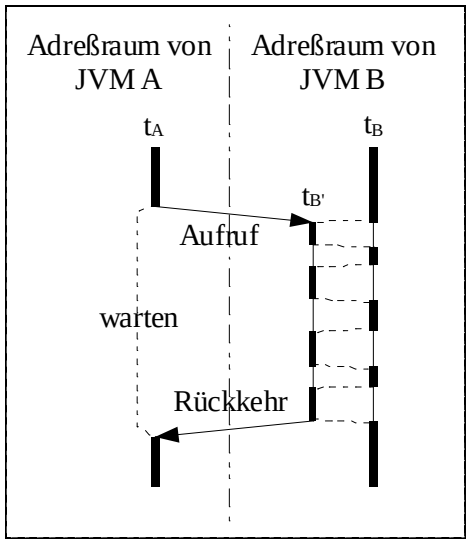

Abbildung 32
Entfernter Methodenaufruf

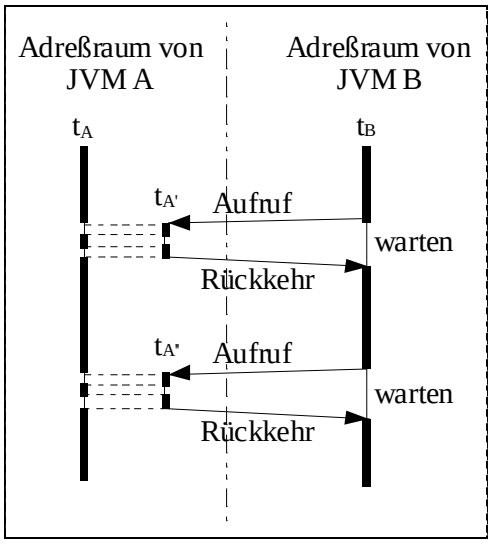

Abbildung 33
Entfernte Methodenaufrufe

Unter gewissen Voraussetzungen wäre es in diesem Beispiel wünschenswert gewesen, das Objekt, dessen Methode in einem entfernten Methodenaufruf ausgeführt wurde, auf Maschine *A* zu plazieren. Im besten Fall benutzt die Aktivität t_B dieses Objekt überhaupt nicht, dann träte durch seine Plazierung auf *A* der maximale Gewinn ein. Es entstünde dann weder Verlust durch Netzwerkkommunikation, noch ginge Rechenzeit auf einer der beiden virtuellen Maschinen durch ungleiche Lastverteilung verloren. Wenn die Aktivität t_B das Objekt benutzt, dessen Methode von t_A entfernt aufgerufen und durch t_B' ausgeführt wurde, kann sich dennoch die Überlegung einer Plazierung dieses Objekts auf *A* lohnen. Abbildung 33 stellt die möglicherweise auftretende Situation bei einer Plazierung dieses Objekt auf *A* vor.

Wir nehmen an, daß bei dieser Plazierung zwei entfernte Methodenaufrufe vom Adreßraum der virtuellen Maschine *B* zum Adreßraum von *A* nötig werden. Die Ausführung dieser beiden Methodenaufrufe wird auf *A* durch die beiden Aktivitäten t_A' und t_A'' übernommen. Dabei erhöht sich einerseits der Kommunikationsaufwand, der allerdings auch von Anzahl und

Größe der übergebenen Parameter abhängt. Da die entfernt aufgerufenen Methoden aber wesentlich kürzer sind, vermindert sich die Zeit, in der die virtuelle Maschine *A* doppelt belastet ist, während *B* wartet. Man würde sich also für diese Version entscheiden, wenn die größere Parallelität den erhöhten Kommunikationsaufwand überwiegt. Wenn man vom Idealfall ausgeht, daß für jede Aktivität eine virtuelle Maschine zur Verfügung steht, und die Aktivitäten normalerweise nur Objekte verwenden, die in Ihrem eigenen Adreßraum existieren, so bedeutet ein einzelner entfernter Methodenaufruf immer eine Einbuße an Parallelität. Die entfernt ausgeführte Methode muß sich den Prozessor mit der Aktivität des Zieladreßraums teilen. Das bedeutet, daß der Code der gerufenen Methode zwar in dem entfernten Adreßraum in einer eigenen Aktivität ausgeführt wird, diese aber durch das Laufzeitsystem mit der anderen Aktivität in diesem Adreßraum serialisiert werden muß, da nur ein realer Prozessor zur Verfügung steht. In diesem einfachen Modell ergibt sich dann der Verlust bei einem entfernten Methodenaufruf als Summe aus geschätzter Ausführungszeit der Methode und Zeit für die Kommunikation mit der entfernten virtuellen Maschine. Aus den Verteilungsalternativen aus Abbildung 32 und Abbildung 33 würde man dann diejenige mit geringerem Verlust auswählen.

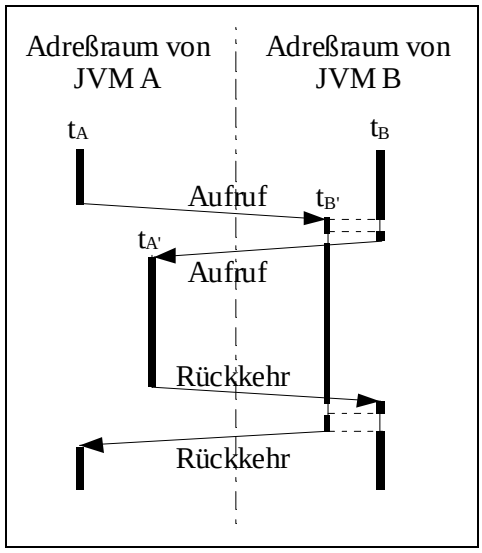

Abbildung 34
Mehrere entfernte Methodenaufrufe

Berücksichtigt man mehrere gleichzeitige entfernte Methodenaufrufe, ist auch die Situation aus Abbildung 34 denkbar. Während die Aktivität t_A eine Methode eines Objekts im Adreßraum *B* entfernt aufruft, führt die Aktivität t_B einen entfernten Methodenaufruf zum Adreßraum *A* durch. Man erkennt, daß hier nur eine geringe Überlappung von zwei Aktivitäten in einem Adreßraum stattfindet. Wenn sich zwei Aktivitäten einen realen Prozessor teilen müssen, ist dies in der Abbildung so angedeutet, daß dann nur eine der beiden durch einen dicken Strich markiert ist. Das tritt hier nur kurz vor und nach dem entfernten Aufruf auf, der von t_B durchgeführt wird. Es gibt also praktisch keinen Verlust durch Einbuße an Parallelität. Nur die Kommunikationszeit für die zwei entfernten Methodenaufrufe fällt ins Gewicht.

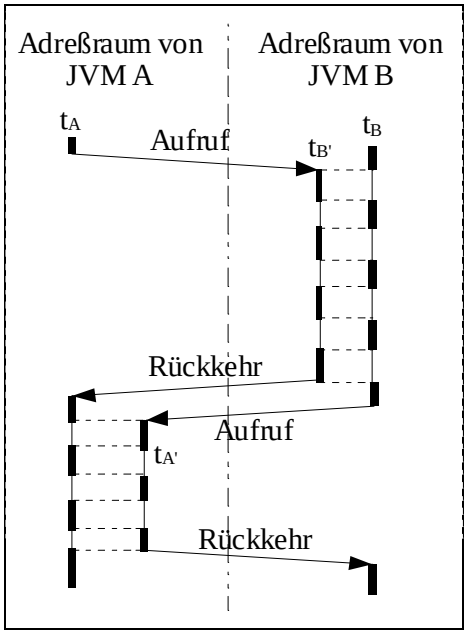

Abbildung 35
Andere Laufzeitbedingungen

Entfernte Methodenaufrufe sind also nicht prinzipiell gleichzusetzen mit einem Verlust an Parallelität. Die Situation in Abbildung 34 tritt allerdings nur unter günstigen Laufzeitbedingungen ein. Wenn sich die Ausführungszeiten der beiden Aktivitäten t_A und t_B ein wenig gegeneinander verschieben, könnte der zeitliche Ablauf wie in Abbildung 35 aussehen. Die beiden gegenseitigen entfernten Methodenaufrufe überlappen nicht mehr. Das hat hier zur Folge, daß fast gar keine Parallelität mehr ausgenutzt werden kann. Es findet Aktivität immer nur entweder im Adreßraum *A* oder Adreßraum *B* statt. Das Verhalten ähnelt dem von zwei Aktivitäten in einem einzelnen Adreßraum, nur daß zusätzlich zu der nicht vorhandenen Parallelität auch noch hohe Kommunikationszeiten in Kauf genommen werden müssen. Unter diesen Laufzeitbedingungen ist eine Verteilung der beiden Aktivitätsstränge auf zwei unterschiedliche virtuelle Maschinen nicht wünschenswert, da sich durch die Kommunikation die

Ausführungszeit gegenüber der Lösung mit nur einer virtuellen Maschine erhöht. Da eine statische Analyse des Programmtextes solche Laufzeitsituationen nicht vorhersagen kann, muß von vereinfachten Voraussetzungen bzw. hier vom schlimmsten Fall ausgegangen werden.

3.3 Leitlienien für die Verteilungsanalyse

3.3.1 Einheiten der Parallelität

Ein JavaParty-Programm ist eine parallele Problemlösung mit Threads als kleinsten parallelen Einheiten. Aufgrund obigen Überlegungen kann ein JavaParty-Programm mit nur einem Thread auch durch die geschickteste Objektverteilung keinen Geschwindigkeitsvorteil erreichen. Alle Methodenaufrufe innerhalb eines Threads finden synchron statt. Durch Verteilung der beteiligten Objekte auf unterschiedliche virtuelle Maschinen wird für die Methoden dieser Objekte die ausführende Maschine gewählt, aber keine zusätzliche Parallelität in das Programm eingeführt, die der Programmierer nicht explizit in Form von Threads ausgedrückt hat. Wenn man von Synchronisationsmechanismen absieht, sind zwei Threads zwei eigenständige Kontrollflüsse, die unabhängig voneinander ausgeführt werden können. Bei Einsatz eines symmetrischen Multiprozessorrechners mit gemeinsamem Adreßraum bestünde daher auch keine Notwendigkeit, sich um die weitere Analyse der einzelnen Aktivitäten zu kümmern. Sofern genügend Prozessoren zur Verfügung stehen, kann jeder Prozessor einen der Threads abarbeiten und man erreicht die durch die Zerlegung des Programms in Threads vorgegebene maximale Geschwindigkeitssteigerung. In dem virtuellen Adreßraum von JavaParty ist die Ausführung einer Methode aber immer an die virtuelle Maschine gebunden, in deren Adreßraum das zugehörige Objekt angelegt ist. Eine Zuordnung von realem Prozessor zu dem abzuarbeitenden Thread ist damit nicht möglich.

3.3.2 Verteilung von Aktivitäten an virtuelle Maschinen in JavaParty

Um eine virtuelle Maschine mit der Bearbeitung einer Aktivität zu beauftragen, muß man dafür sorgen, daß sich alle Objekte, deren Methoden von der Aktivität betreten werden, im Adreßraum dieser Maschine befinden. Wäre dies für alle Aktivitäten und Objekte eines Programms möglich, würde man den maximalen Geschwindigkeitsgewinn gegenüber der Java-Lösung erzielen. Das setzte aber voraus, daß die unterschiedlichen Aktivitäten des Programms keine gemeinsamen Objekte verwenden. Das ist gleichbedeutend damit, daß die Aktivitäten des Programms nicht miteinander kommunizieren. Dieser Fall kommt in der Realität nicht vor, da sich das Programm dann so verhalten würde wie viele einzelne Programme, die nichts miteinander zu tun haben. Die zentrale Eigenschaft von Threads ist aber, daß sie einfach über gemeinsame Objekte untereinander kommunizieren und so Ergebnisse von Berechnungen austauschen können. Ohne Kommunikation der Aktivitäten untereinander ist das Problem trivial. Jedes der einzelnen Programme könnte separat auf einem eigenen Rechner gestartet werden. Damit würde dann die maximale Geschwindigkeitssteigerung erzielt. Dieser Wert ist gleichzeitig eine obere Schranke für die in JavaParty erreichbare Geschwindigkeitssteigerung.

Der Idealfall, daß eine Aktivität ganz von einer virtuellen Maschine und damit echt parallel zu allen anderen Aktivitäten ausgeführt wird, ist in JavaParty wegen des verteilten Adreßraums nicht zu erreichen. Wie oben beschrieben, müßten sich alle Objekte, deren Methoden von der Aktivität ausgeführt werden, in deren Adreßraum befinden. Dies ist aber bei Objekten, die von mehreren Aktivitäten benutzt werden, nicht möglich. Man kann nur versuchen, sich an den Idealfall anzunähern. Um Aktivitäten an die zur Verfügung stehenden virtuellen Maschinen zuordnen zu können, wird als erstes eine Zuordnung von Objekten zu Aktivitäten benötigt. Diese Zuordnung gibt für jedes Objekt an, im Adreßraum welcher Aktivität es angelegt werden soll. Folgende Bedingung muß die Zuordnung für eine gute Verteilung der Aktivitäten erfüllen: Ein Objekt ist einer bestimmten Aktivität zugeordnet, wenn diese Aktivität die meisten und komplexesten Berechnungen in Methoden dieses Objekts durchführt. Plaziert man ein solches Objekt auf einer bestimmten virtuellen Maschine, so wandert mit ihm ein entsprechend großer Teil der Berechnung der zugehörigen Aktivität auf diese

Maschine. Ruft die Aktivität Methoden von Objekten auf, die ihr nach dieser Regel nicht zugeordnet sind, wird für diesen Aufruf ein entfernter Methodenaufruf nötig. Diese entfernt aufgerufenen Methoden führen dann aber keine langen Berechnungen durch, sondern dienen eher der Kommunikation zwischen den Aktivitäten, da Objekte im Adreßraum derjenigen Aktivität angelegt werden, die den Großteil der Berechnung in Methoden dieses Objekt durchführt.

Läßt man Synchronisationsmechanismen außer Acht, so sind die Teile der Berechnung, die in Methoden von Objekten stattfinden, welche der jeweiligen Aktivität zugeordnet sind, parallel zu anderen Aktivitäten ausführbar, da sie auf einer eigenen virtuellen Maschine ablaufen. Methoden, die entfernt aufgerufen werden müssen, sollten möglichst kurz sein, da sie sich die virtuelle Maschine mit einer anderen Aktivität teilen müssen. Je weniger Parameter übergeben werden, desto kürzer ist die Zeit, welche zur Netzwerkkommunikation notwendig ist.

4 Verteilungsanalyse

Die allgemein gehaltenen Aussagen des letzten Kapitels über Aktivitäten und die zu berücksichtigenden Gesichtspunkte bei Ihrer Verteilung im virtuellen JavaParty-Adreßraum sollen nun konkretisiert und in eine Verteilungsstrategie umgesetzt werden, die sich im Übersetzer implementieren läßt. Dafür müssen für die verwendeten Begriffe wie Aktivitäten, Objekte, Berechnungszeit von Methoden und Strukturgrößen von Parametern, die sich auf die Laufzeit des Programms beziehen, geeignete Abschätzungen gefunden werden.

4.1 Identifikation von Aktivitäten und Objekten

Das zu übersetzende Programm kann zur Laufzeit potentiell unbegrenzt viele Aktivitäten erzeugen. Die statische Analyse kann aber nur mit endlich vielen Aktivitäten umgehen. Aus diesem Grund müssen die zur Laufzeit erzeugbaren Aktivitäten für die statische Analyse in endlich viele Äquivalenzklassen eingeteilt werden. Man beschränkt sich dann darauf, jede dieser Klassen als Ganzes zu untersuchen und die Ergebnisse dann für jedes Element dieser Klasse zu verwenden.

Ein Thread beginnt seine Abarbeitung zur Laufzeit mit der Ausführung seiner *run*-Methode. In der durch ihn repräsentierten Aktivität wird also diese *run*-Methode mit allen ihren Unteraufrufen durchgeführt. Die Aktivität endet, wenn der Aufruf der r*un*-Methode des Threads zurückkehrt. Während der Analysephase möchte man gerne den in dieser Aktivität auftretenden Kontrollfluß möglichst gut annähern, um Aussagen darüber zu treffen, welche Methoden welcher Klassen aufgerufen werden. Eine solche Annäherung ist das Ergebnis der Typinferenz aus Kapitel 2. Wo immer möglich wird durch Teilen von Methoden- und Klassenkonturen erreicht, daß die konkreten Typen von Variablen, mit denen Methoden aufgerufen werden, eindeutig sind. Dadurch wird der interprozedurale Kontrollfluß an diesen Stellen statisch, d.h. zur Übersetzungszeit ist die Methode bekannt, die dort zur Laufzeit aufgerufen wird. Für die Analyse der Aktivitäten bedeutet das, daß nach der Typinferenz der von der *run*-Methode einer Threadklasse erreichbare Kontrollflußgraph statisch ist und als Übersetzungszeitapproximation für die Aktivität selbst verwendet werden kann. Die Kontur einer Threadklasse repräsentiert eine Menge von Aktivitäten zur Laufzeit, die als Übersetzungszeitapproximation denselben statischen Kontrollfluß haben.

Wie in 3.3.2 ausgeführt, sind Aktivitäten immer an den Adreßraum gebunden, in dem sich das Objekt befindet, dessen Methode von der Aktivität bearbeitet wird. Daher sind die Objekte von besonderem Interesse, deren Methoden im Laufe der Abarbeitung einer Aktivität aufgerufen werden. Genauso wie die Anzahl der Aktivitäten ist auch die Anzahl der während der Programmausführung erzeugbaren Objekte beliebig groß. Zur Einteilung in eine für Analysezwecke handhabbare Anzahl von Äquivalenzklassen bieten sich ebenfalls die Klassenkonturen der Typinferenz an. Wenn der zur Threadkontur *T* gehörende Kontrollfluß eine Methode *f* der Klassenkontur *A* aufruft, so ist das ein Indiz dafür, daß zur Laufzeit eine Aktivität, deren Threadobjekt der Threadkontur *T* angehört, eine Methode *f* eines Objekts aufrufen kann, das selbst der Klassenkontur *A* angehört.

Die konkreten Typen teilen während der Analysephase sowohl Objekte als auch Aktivitäten in Äquivalenzklassen ein. Elemente einer Äquivalenzklasse sind für die Analyse nicht weiter unterscheidbar. Berechnet werden nur Beziehungen, die zwischen Elementen aus unterschiedlichen Äquivalenzklassen gelten. Eine solche Beziehung zwischen Äquivalenzklassen ist zwischen Elementen dieser Äquivalenzklassen immer nur eine kann-Beziehung. Angewendet auf das Beispiel oben heißt das: Wenn die Analyse herausfindet, daß die Threadkontur T die Methode f der Klassenkontur A aufruft, dann bedeutet dies für eine konkrete Aktivität $t \in T$ und ein Objekt $a \in A$, daß die Aktivität t möglicherweise die Methode f des Objekts a ausführt. Es ist aber noch nicht einmal sicher, daß zur Laufzeit überhaupt eine Aktivität $t' \in T$ und ein Objekt $a' \in A$ existieren, so daß t' die Methode f von a' abarbeitet. Das liegt daran, daß der Aufruf von f zwar im Programmtext steht, die Anweisung aber zur Laufzeit möglicherweise nie ausgeführt wird. Die bei der Typinferenz verwendete Datenflußanalyse ist selbst nur eine konservative Approximation an den lokalen Kontroll- und Datenfluß einer jeden Methode. Es wird jeder syntaktisch mögliche Kontroll- und Datenfluß berücksichtigt unabhängig davon, ob er zur Laufzeit auch tatsächlich eintritt.

Im Umkehrschluß heißt das aber auch, wenn die Analyse keinen Hinweis findet, daß T die Methode f von A aufruft, dann steht fest, daß es auch zur Laufzeit keine Aktivität $t \in T$ und kein Objekt $a \in A$ gibt, so daß t die Methode f von a ausführt. Wenn im folgenden von Objekten und Aktivitäten die Rede ist, bedeutet das immer die entsprechende Äquivalenzklasse. Die durch eine Klassenkontur repräsentierte Äquivalenzklasse ist die Menge der Laufzeitobjekte, die einer bestimmten Klassenkontur angehören.

4.2 Ziel der Aktivitätsanalyse

Der Idealfall, daß Methoden eines Objekts ausschließlich von einer Aktivität ausgeführt werden, daß dieses Objekt also ausschließlich von einer Aktivität verwendet wird, ist in realen Programmen eher die Ausnahme. Um dennoch für alle Objekte eine Verteilungsregel berechnen zu können, die trotzdem eine gute Separierung der Aktivitäten in unterschiedliche Adreßräume erreicht, muß man Maße definieren, anhand derer man widersprüchliche Anforderungen gegeneinander abwägen kann. Wie in 3.3 erläutert, ist eine Verteilung dann gut, wenn

- eine Aktivität möglichst viel Rechenzeit in dem ihr zugeordneten Adreßraum verbringt und
- möglichst wenig Zeit zur Kommunikation mit anderen Adreßräumen aufwendet.

Aus der Sicht eines Objekts, das von mehreren Aktivitäten benutzt wird, heißt das, es muß einerseits in dem Adreßraum plaziert werden, in dem es am meisten verwendet wird, damit die nutzbare Parallelität möglichst groß wird. Andererseits sollte es aber so plaziert werden, daß die dadurch entstehende Kommunikation minimal wird.

Diese Forderungen sollen im weiteren konkretisiert werden. Die abzuschätzenden relevanten Größen sind dabei die Rechenzeit, die im Mittel von einer Aktivität beim Aufruf einer Methode in dieser verbraucht wird und die mittlere Dauer, um die sich ein bestimmter Methodenaufruf verlängert, wenn dieser entfernt durchgeführt wird. Der vom jeweiligen Methodenaufruf abhängige Parameter ist dabei die Größe der in den Parametern übergebenen Strukturen. Hier ist die Eigenschaft eines JavaParty-Methodenaufrufs relevant, daß nur entfernte Objekte als Referenz-Parameter übergeben werden können. Lokale Objekte werden beim entfernten Methodenaufruf als Werte übergeben. Die Zeit für die dafür notwendige Serialisierung und das Verschicken über das Netzwerk ist dabei von der Strukturgröße der übergebenen Parameter abhängig.

4.3 Komplexität von Methoden

Die Komplexität einer Methode soll hier die zu ihrer Abarbeitung nötige mittlere Zeit bedeuten. Dabei soll nur der Rumpf der Methode und nicht die etwa in ihr ausgeführten Unteraufrufe anderer Methoden berücksichtigt werden. Die mittlere Laufzeit einer Methode ist für die Verteilungsanalyse eine zentrale Größe, da aus ihr der erwartete Gewinn bei der Parallelausführung der Methode in

einem eigenen Adreßraum hervorgeht. Leider läßt sich diese Größe nur ungenügend schätzen. Schleifen und Fallunterscheidungen im Rumpf einer Methode entziehen sich im allgemeinen der Möglichkeit einer sinnvollen statischen Vorhersage. Wie oft der Körper einer Schleife im Mittel durchlaufen wird bzw. mit welcher Wahrscheinlichkeit die einzelnen Äste einer Verzweigung ausgeführt werden, kann nicht bestimmt werden. An diesen Stellen muß mit sehr groben Näherungen bzw. Ersatzwerten gerechnet werden.

Aufgrund der Struktur des Methodenrumpfes bestimmt man für jede Anweisung ihre mittlere Ausführungshäufigkeit unter der Voraussetzung, daß die Methode genau einmal aufgerufen wird. Der Rumpf einer Methode ist eine Sequenz von Anweisungen. Jede Anweisung ist entweder eine einfache Anweisung, eine Schleife, oder eine Fallunterscheidung. Jede Anweisung direkt im Rumpf einer Methode wird genau einmal ausgeführt. Nimmt man an, daß jede Schleife die gleiche willkürlich festgelegte Anzahl Iterationen (10) durchführt, multipliziert sich die mittlere Häufigkeit der Ausführung einer Anweisung innerhalb einer Schleife mit diesem konstanten Faktor. Bei einer Fallunterscheidung geht man davon aus, daß die Ausführungswahrscheinlichkeit jedes Astes der Verzweigung gleich hoch ist. Daraus folgt, daß die mittlere Ausführungshäufigkeit einer Anweisung in jedem Ast der Verzweigung gleich der Ausführungshäufigkeit der Verzweigung geteilt durch die Anzahl ihrer Äste ist. *If*-Anweisungen haben immer genau zwei Äste (ein fehlendes *else* ist gleichbedeutend mit einem leeren *else*-Zweig), *switch*-Anweisungen können beliebig viele Verzweigungsäste haben.

Die Komplexität einer Methode ergibt sich dann aus der Summe der Ausführungszeiten der elementaren Anweisungen der Methode multipliziert mit den jeweiligen geschätzten mittleren Ausführungshäufigkeiten.

4.3.1 Lokale Komplexität

In JavaParty gibt es neben entfernten Objekten weiterhin Instanzen lokaler Klassen. Diese Instanzen sind normale Java-Objekte, die nur lokal im Adreßraum einer virtuellen Maschine bekannt sind. Aufrufe von Methoden lokaler Objekte werden daher immer in demselben Adreßraum des aufrufenden Objekts abgearbeitet. Mit der Plazierung eines entfernten Objekts in einem bestimmten Adreßraum werden sowohl alle Methoden dieses Objekts als auch alle Unteraufrufe von Methoden lokaler Objekte in diesem Adreßraum durchgeführt. Da die Verteilungsanalyse an der Abarbeitungszeit interessiert ist, die mit der Plazierung eines entfernten Objekts sicher im Adreßraum dieses Objekts anfällt, ist es sinnvoll, bei der Ausführungszeit einer Methode die Ausführungszeiten aller lokal durchgeführten Unteraufrufe mitzuzählen. Die Schätzung für diese Ausführungszeit soll *lokale Komplexität* heißen.

Bei jedem Methodenaufruf läßt sich am konkreten Typ der Variablen, mit der der Aufruf stattfindet, ablesen, ob es sich um einen lokalen oder entfernten Aufruf handelt. Die lokale Komplexität einer Methode berechnet sich dann als Summe aus ihrer Komplexität und allen lokalen Komplexitäten, der von ihr aus lokal aufgerufenen Methoden, jeweils gewichtet mit der geschätzten Ausführungshäufigkeit des Methodenaufrufs. Ist die aufgerufene Methode nicht eindeutig, so wird über die lokalen Kompexitäten aller an dieser Stelle möglichen Methoden gemittelt. Ein Methodenaufruf ist dann nicht eindeutig, wenn die Typinferenz an dieser Stelle nicht alle Kontrollflußunschärfen auflösen konnte.

Bei rekursiven Methodenaufrufen muß man sich ähnlich wie bei Schleifen im Rumpf einer Methode mit einer groben Näherung abfinden. Rekursive Zyklen im lokalen Untergraph einer Methode werden wie bei Schleifen mit einem konstanten willkürlichen Faktor (10) höher gewichtet.

4.4 Strukturgrößen

Die konkreten Typen der Parameter einer Methode geben an, mit welchen Objekten die Methode zur Laufzeit aufgerufen werden kann. Da bei der Übergabe von lokalen Objekten in entfernten Metho-

denaufrufen keine Referenzen sondern die Objekte selbst als Werte übergeben werden müssen, ist eine Abschätzung der Größe der übergebenen Struktur sinnvoll. Die Größe von Werten eines Basistyps ist bekannt. Die Abschätzung der Größe eines Objekts ist die Summe der geschätzten Größen seiner Instanzvariablen. Die Instanzvariablen sind entweder wieder von einem Basistyp oder selbst Referenzen auf Objekte. Im zweiten Fall helfen bei der Abschätzung deren Größen die durch die Typinferenz ermittelten konkreten Typen. Ist der konkrete Typ einer Instanzvariablen die Kontur einer Fernklasse, so wird ihre Größe mit einer Konstante veranschlagt, da beim Methodenaufruf entfernte Objekte via Referenz übergeben werden. Enthält eine Instanzvariable selbst wieder eine Referenz auf ein lokales Objekt, wird dessen geschätzte Größe verwendet.

Konnte die Typinferenz keinen eindeutigen konkreten Typ für eine Instanzvariable ermitteln, so nimmt man an, daß das Auftauchen von Objekten jedes der Typen gleich wahrscheinlich ist. Man mittelt dann über die geschätzten Größen aller Objekte, die aufgrund des konkreten Typs zur Laufzeit möglich sind. Bis zu diesem Punkt kann man erwarten, gute Schätzwerte für die Größen von Datenstrukturen zu ermitteln. Die Güte der Schätzwerte hängt nur davon ab, wie viele Unsicherheiten im konkreten Typ von Variablen nach der Typinferenz noch zurückbleiben. Die Abschätzung der Größe von Strukturen stößt aber an seine Grenzen, wenn Felder oder rekursive Datenstrukturen verwendet werden.

Wie bei Schleifen und rekursiven Methodenaufrufen muß man sich beim Schätzen der Größe von Feldern und rekursiven Datenstrukturen mit sehr vagen Näherungen abfinden. Felder werden so gezählt, als hätten sie immer eine konstante, vorher festgelegte Anzahl von Elementen. Rekursive Zyklen in Datenstrukturen werden wie bei rekursiven Methodenaufrufen mit einem konstanten Faktor höher gewichtet.

4.5 Kontrollflußgraphen der Aktivitäten

Die lokale Komplexität einer Methode gibt die beim Aufruf anfallende Rechenzeit im Adreßraum des betreffenden Objekts an. Über die Größen der Objekte in ihren Parametern läßt sich die Kommunikationszeit zum einmaligen Aufruf dieser Methode schätzen. Diese Abschätzung muß jetzt vom einmaligen Aufruf einer Methode zu allen Aufrufen innerhalb einer Aktivität und vom Aufruf einer einzelnen Methode zu allen Aufrufen von Methoden eines Objekts innerhalb einer Aktivität erweitert werden.

Im folgenden sollen aus den bisher gewonnenen Größen für jedes Objekt die beiden Parameter entwickelt werden, aufgrund derer die Zuordnung der Objekte zu den Aktivitäten und damit die Verteilung der Objekte auf die zur Verfügung stehenden virtuellen Maschinen erfolgt. Das ist zum einen die Rechenzeit, die mit der Plazierung des Objekts *a* in einem Adreßraum sicher von der Aktivität *t* in diesem Adreßraum verbraucht wird. Diese Größe soll mit *work(t, a)* bezeichnet werden. Zum anderen die Kommunikationszeit *cost(t, a)*, die anfällt, wenn das betreffende Objekt nicht im Adreßraum der Aktivität plaziert wird.

4.5.1 Rechenzeit pro Objekt

Möchte man die Rechenzeit abschätzen, die eine Aktivität während ihres Ablaufs sicher im Adreßraum eines Objekts zubringt, benötigt man dazu eine Schätzung der Aufrufhäufigkeiten der Methoden des Objekts in dieser Aktivität. Die erwartete Ausführungshäufigkeit einer Methode innerhalb einer Aktivität ist die geschätzte Anzahl der Aufrufe dieser Methode unter der Voraussetzung, daß die Wurzel der Aktivität genau einmal ausgeführt wird. Die Wurzel einer Aktivität ist die *run*-Methode des die Aktivität repräsentierenden Threadobjekts. Allgemein benötigt man eine Schätzformel für die erwartete Ausführungshäufigkeit einer Methode *h* unter der Voraussetzung, daß eine andere Methode *f* genau einmal aufgerufen wird.

Diese Häufigkeit soll mit *calls(f, h)* bezeichnet werden. *calls(f, h)* ergibt sich aus der Summe der Ausführungshäufigkeiten von Anweisungen in *f*, die die Methode *h* direkt aufrufen, plus der Summe

aller übrigen Ausführungshäufigkeiten von Anweisungen, die eine Methode *g* aufrufen, multipliziert mit *calls(g, h)*. Rekursive Zyklen in den Aufrufen müssen wieder separat betrachtet werden.

Kürzt man mit *calls(t,* f_a*) = calls(*run_t*,* f_a*)* die erwartete Aufrufhäufigkeit der Methode f_a in der Aktivität *t* ab, ergibt sich für eine Aktivität *t* und ein Objekt *a* direkt die Größe *work(t, a)* als Summe von *calls(t, f)* multipliziert mit der lokalen Komplexität von *f* über alle Methoden *f* dieses Objekts. Dabei ist run_t die Wurzel der Aktivität gleichbedeutend mit der *run*-Methode des die Aktivität *t* repräsentierenden Threadobjekts.

4.5.2 Kommunikation pro Objekt

Die bei Plazierung eines Objekts entfernt von einer Aktivität anfallenden Kommunikationszeiten lassen sich nicht lokal schätzen. Die Kommunikationszeiten hängen immer von der Plazierung der anderen Objekte ab, von welchen die Aufrufe ausgehen. Das spricht dafür, daß ein globales Optimierungsverfahren angebracht wäre. Hier soll aber von einem vereinfachten Modell ausgegangen werden.

Man nimmt an, daß sich das aufrufende Objekt im Adreßraum der den Methodenaufruf ausführenden Aktivität befindet. Finden aus der aufgerufenen Methode weitere Unteraufrufe von Methoden entfernter Objekte statt, so geht man davon aus, daß auch diese im Adreßraum der Aktivität plaziert

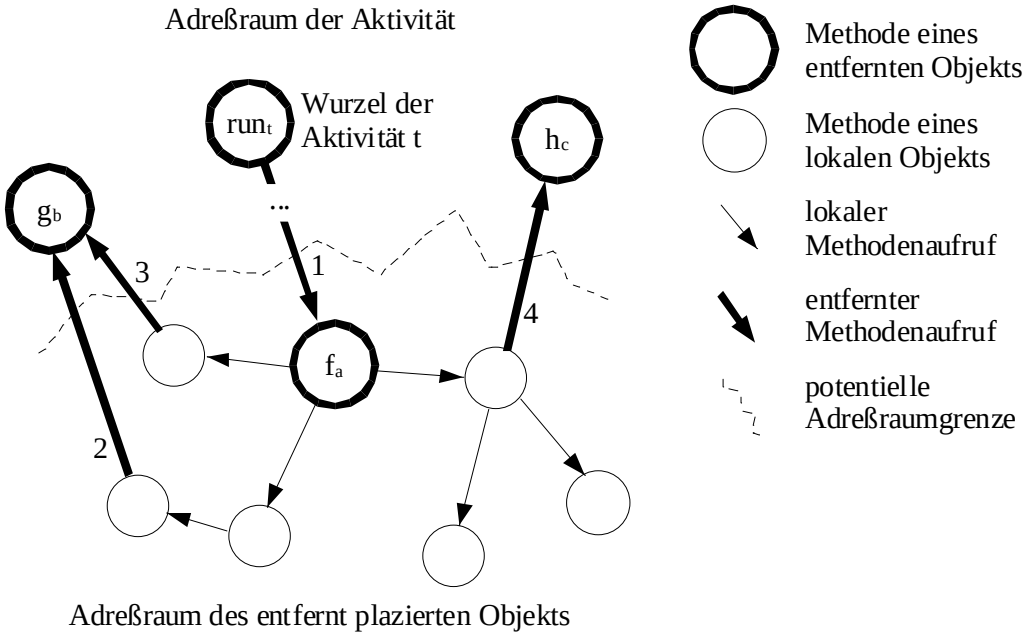

Abbildung 36
Modell einer Aktivität in JavaParty

sind. Diese Annahmen stellen den schlimmsten Fall dar: Wird ein Objekt nicht im Adreßraum der aufrufenden Aktivität plaziert, müssen sowohl der Aufruf als auch eventuelle Unteraufrufe entfernt durchgeführt werden.

Diese Situation ist in Abbildung 36 veranschaulicht. Methoden entfernter Objekte sind durch dicke Kreise markiert, Methoden lokaler Objekte durch kleine dünne Kreise. Entsprechend sind entfernte Methodenaufrufe durch breite Pfeile angedeutet. Die entfernten Methodenaufrufe sind außerdem von 1 bis 4 durchnumeriert. Betrachtet wird hier die Methode f_a eines entfernten Objekts *a*. Der Untergraph des von f_a ausgehenden Kontrollflusses, der bei Plazierung des Objektes *a* sicher in dessen Adreßraum stattfindet, ist durch eine gestrichelte Linie abgetrennt. Alle diese Berechnungen tragen

zur lokalen Komplexität von f_a bei. Zur Schätzung der auftretenden Kosten, wenn a entfernt von der Aktivität *t* plaziert wird, geht man davon aus, daß alle aus dem lokalen Untergraphen von f_a aufgerufenen entfernten Objekte (*b* und *c*) wieder im Adreßraum der Aktivität *t* liegen; alle entfernten Methodenaufrufe 1-4 müssen also auch wirklich entfernt durchgeführt werden. In 4.5.3 wird hierfür eine Begründung geliefert.

Der einmalige entfernte Aufruf einer Methoden *f* benötigt gegenüber dem lokalen Aufruf der selben Methode eine um $delta_f$ längere Zeitspanne. Diese zum entfernten Aufruf zusätzlich nötige Zeit $delta_f$ setzt sich zusammen aus einem konstanten Anteil für jeden entfernten Methodenaufruf und einer variablen Zeitspanne, die proportional zur Größe der Strukturen in den Argumenten ist. Unter Verwendung der konkreten Typen der Methodenparameter und der Größenschätzung für diese (vgl. 4.4) läßt sich für jede Methode *f* die Zeitspanne $delta_f$ berechnen, um die sich der Aufruf von *f* verlängert, wenn er entfernt durchgeführt werden muß.

Zur Schätzung der anfallenden Kommunikationszeiten bei Plazierung des Objekts *a* in einem Adreßraum entfernt von der aufrufenden Aktivität *t* und bei einmaligem Aufruf der Methode f_a benötigen wir noch eine Schätzung der Häufigkeiten von Aufrufen der Methoden g_b und h_c, die in Folge des Aufrufs von f_a direkt aus dem Adreßraum von *a* stattfinden. Diese erwarteten Ausführungshäufigkeiten von g_b und h_c unter der Annahme, daß f_a genau einmal ausgeführt wird, werden mit *directCalls*(f_a, g_b) und *directCalls*(f_a, h_c) bezeichnet. Sie berechnen sich aus den kumulierten Ausführungshäufigkeiten der Anweisungen ausgehend von f_a, die zum Aufruf der entsprechenden Methode g_b bzw. h_c führen (ohne daß zwischendurch ein weiterer entfernter Methodenaufruf durchgeführt wird).

Damit ergeben sich die anfallenden Kommunikationskosten *cost*(f_a) für den einmaligen Aufruf der Methode f_a aus der Summe der Kosten des entfernten Methodenaufrufs 1 ($delta_{f_a}$) und den Kosten ($delta_{g_b}$, $delta_{h_c}$) der entfernten Unteraufrufe von Methoden g_b und h_c gewichtet mit ihren erwarteten Ausführungshäufigkeiten (*directCalls*(f_a, g_b), *directCalls*(f_a, h_c)).

Die entstehenden Kommunikationskosten *cost(t, a)* bei entfernter Plazierung von *a* sind dann die Summe aus allen *cost*(f_a) für alle Methoden des Objekts *a* jeweils multipliziert mit der erwarteten Aufrufhäufigkeit *calls(t,* f_a*)* der Methode f_a in der Aktivität *t*.

4.5.3 Plausibilitätsprüfung der Kostenberechnung

Offensichtlich gewichtet der Ansatz zur Kostenberechnung aus 4.5.2 die Kosten einer Kante im Kontrollflußgraphen, die einen möglicherweise entfernten Methodenaufrufs repräsentiert, doppelt. Wie man sich an Abbildung 36 verdeutlichen kann, gehen die Kosten des Aufrufs der Methode g_b einmal in die Plazierungsentscheidung des Objekts *a* ein, dessen Methode f_a Aufrufe von g_b durchführt. Die Vorgehensweise bei der Berechnung der Kosten eines Aufrufs von f_a wurden oben erläutert. Führt man dieselbe Kostenrechnung auch für g_b durch, so gehen darin die Kosten für den Aufruf von g_b erneut ein.

Daß die Berücksichtigung der Kommunikationskosten einer Methode sowohl bei ihrem Aufruf als auch bei ihren Unteraufrufen sinnvoll ist, kann man sich an einem einfachen Beispiel klarmachen: Man nimmt an, daß die Methode f_a aus Abbildung 36 zusammen mit ihren lokalen Unteraufrufen nur eine geringe lokale Komplexität hat, dafür aber ausgesprochen oft die Methoden g_b und h_c aufruft. Würde diese Kommunikation während der Unteraufrufe bei der Plazierung von *a* außer Acht gelassen, würde sich die Analyse vielleicht aufgrund der geringen Komplexität von f_a für eine

entfernte Plazierung von *a* entscheiden. Die ausgiebige Kommunikation mit g_b und h_c würde dann nur bei den Plazierungsentscheidungen für *b* und *c* als Indiz für eine lokale Plazierung im Adreßraum von *t* bewertet. Dies hätte den unerwünschten Effekt, daß dadurch die Kommunikation erst nötig wird. Berücksichtigt man dagegen die Kommunikationskosten auch beim Aufruf von Methoden aus f_a heraus und entscheidet sich trotzdem für eine entfernte Plazierung von *a*, dann kann man eher erwarten, daß sich die dadurch anfallenden Kommunikationszeiten amortisieren, da sie schon in der Plazierungsentscheidung für *a* berücksichtigt wurden und sich die Analyse trotzdem für eine entfernte Plazierung von *a* entschieden hat.

4.6 Plazierungsentscheidungen

Die die Plazierung eines Objekts *a* betreffenden Faktoren sind jetzt auf *work(t, a)* und *cost(t, a)* reduziert. *work(t, a)* beschreibt die von einer Aktivität *t* in Methoden von *a* aufgewandte Rechenzeit, *cost(t, a)*, die bei entfernter Plazierung anfallenden Kommunikationszeiten bei Aufrufen von Methoden des Objekts *a* innerhalb einer Aktivität *t*. Durch die Plazierung eines Objekts *a* sollte die Rechenzeit maximiert werden, die diejenige Aktivität in Methoden von *a* verbringt, in deren Adreßraum *a* angelegt wird. Gleichzeitig sollte die Summe der Kommunikationszeiten minimiert werden, die dadurch in allen anderen Aktivitäten t_i anfallen, von denen *a* dann entfernt plaziert ist.

Um den Rahmen der Arbeit nicht zu sprengen, wird kein Graphpartitionierungsalgorithmus implementiert, sondern in einem iterativen Verfahren werden zuerst die Objekte an die Aktivitäten verteilt und anschließend die Aktivitäten den zur Verfügung stehenden Adreßräumen zugewiesen. Jeder Schritt versucht den maximalen Gewinn zu erreichen, was in der Regel nicht zu der optimalen Lösung führt.

Nimmt man an, daß bei Plazierung von *a* im Adreßraum von *t* Methodenaufrufe von *a* innerhalb von *t* parallel zu anderen Aktivitäten ausgeführt werden können, gibt *work(t, a)* die Zeit an, welche man durch die Plazierung von *a* im Adreßraum von *t* gewinnt (realisierte Parallelität). Dem gegenüber stehen die Kommunikationszeiten, die in allen anderen Aktivitäten zur Kommunikation mit Methoden von *a* aufgewendet werden müssen. Diese Kommunikationszeiten amortisieren sich dann, wenn *work(t, a)* größer als die Summe aller *cost(t_i , a)* der übrigen Aktivitäten t_i ist.

Wegen der Unsicherheit der verfügbaren Schätzwerte wird aber lediglich die Differenz aus *work(t, a)* und $\Sigma_{t_i \neq t}$ *cost(t_i, a)* maximiert. Für jedes Objekt *a* sucht man diejenige Aktivität *t*, die *work(t, a)* - $\Sigma_{t_i \neq t}$ *cost(t_i, a)* maximiert. Dies ergibt für jedes Objekt *a* eine Zuordnung zu der Aktivität, in dessen Adreßraum das Objekt *a* plaziert werden soll.

$$\text{activity(a)} = \text{t} \Leftrightarrow \text{work(t, a)} - \Sigma_{\text{t}_\text{i} \neq \text{t}}\ \text{cost(t}_\text{i}\text{, a)}\ \text{maximal unter allen Aktivitäten t'}$$

Ordnet man jetzt den Aktivitäten virtuelle Maschinen zu, ergibt sich daraus die Objektverteilung. Da in der Regel nicht so viele virtuelle Maschinen zur Verfügung stehen, wie Aktivitäten im Programm identifiziert werden konnten, müssen sich einige Aktivitäten eine virtuelle Maschine teilen. Es ist im folgenden also noch nötig, die Aktivitäten zu identifizieren, die sich besonders gut eigenen, auf einer gemeinsamen virtuellen Maschine abzulaufen.

Die Gruppierung der im Programm identifizierten Aktivitäten wird in Abschnitt 4.6.1 beschrieben. Jeder der zur Verfügung stehenden virtuellen Maschine wird daraufhin eine der dabei entstehenden Gruppen von Aktivitäten zugeteilt. Alle Objekte, die einer Aktivität aus einer solchen Gruppe zugeordnet wurden, werden dann auf der virtuellen Maschine dieser Gruppe angelegt.

4.6.1 Gruppierung von Aktivitäten

Aus der Zuordnung jedes Objekts *a* zu einer Aktivität *activity(a)* kann, kann für jede Aktivität *t* ihr *Parallelisierungsgewinn win(t)* geschätzt werden. Er errechnet sich aus Summe über *work(t, a)* für

Objekte *a*, die im Adreßraum von *t* plaziert wurden, abzüglich der Summe über *cost(t, b)* für entfernt von *t* plazierte Objekte *b*.

$$\text{win(t)} := \Sigma_{\text{activity(a)} = \text{t}}\ \text{work(t, a)}\ -\ \Sigma_{\text{activity(b)} \neq \text{t}}\ \text{cost(t, b)}$$

Die Summe aus *work(t, a)* steht für die Rechenzeit, die die Aktivität in ihrem Adreßraum unabhängig von anderen Aktivitäten zubringt. Die während dessen verrichtete Arbeit kann daher parallel zu anderen Aktivitäten ausgeführt werden, wenn dem nicht die Verwendung von Synchronisationsmechanismen entgegensteht. Dem gegenüber steht die Zeit, die insgesamt während der Ausführung der Aktivität zur Kommunikation mit anderen Adreßräumen aufgewendet werden muß. Diese Kommunikationszeit ergibt sich aus der Summe über *cost(t, b)* für alle diejenigen Objekte *b*, die nicht der Aktivität *t* zugeordnet wurden.

Stehen zur Ausführung eines Programms weniger virtuelle Maschinen zur Verfügung, als Aktivitäten im Programm identifiziert wurden, werden diejenigen Aktivitäten mit dem kleinsten Schätzwert für den Parallelisierungsgewinn ausgewählt, um einem Adreßraum einer anderen virtuellen Maschine zugeordnet zu werden. Streng genommen könnte man anhand des Schätzwertes für den Parallelisierungsgewinn absolut festlegen, ob sich die Zuordnung einer Aktivität an eine eigene virtuelle Maschine lohnt oder nicht. Bedenkt man aber die Unsicherheit der Faktoren, die in den Schätzwert für den Parallelisierungsgewinn eingehen, erscheint es eher zweifelhaft, ob ein absoluter Schwellwert eine sinnvolle Entscheidung über die Zuordnung einer Aktivität an einen eigenen Adreßraum ergibt. Deswegen sollen in der vorliegenden Implementierung immer alle zur Verfügung stehenden virtuellen Maschinen ausgenutzt werden. Über diesen Parameter kann der Benutzer dann Einfluß auf die Parallelisierungsentscheidungen nehmen.

Stehen *vm* virtuelle Maschinen zur Verfügung, werden die *vm* Aktivitäten $t_1 \ldots t_{vm}$ mit dem höchsten Parallelisierungsgewinn *win(.)* ausgewählt. Sie bilden zu Beginn *vm* einelementige Mengen $G_1 = \{t_1\} \ldots\ G_{vm} = \{t_{vm}\}$. Jede dieser Mengen G_i steht für eine Gruppe von Aktivitäten, die auf der selben virtuellen Maschine angelegt werden. Für die übrigen Aktivitäten mit niedrigerem Parallelisierungsgewinn kann nun erneut ein Parallelisierungsgewinn berechnet werden unter der Voraussetzung, daß sie einer der Mengen G_i zugewiesen werden.

$$\text{win}(G_i, \text{t}) := \Sigma_{\text{activity(a)} \in (G_i \cup \{\text{t}\})}\ \text{work(t, a)}\ -\ \Sigma_{\text{activity(b)} \notin (G_i \cup \{\text{t}\})}\ \text{cost(t, b)}$$

Die Aktivitäten, die bis jetzt noch keiner der Mengen G_i zugewiesen wurden, werden jetzt sukzessive auf diese Mengen verteilt. Eine Aktivität *t* wird zu derjenigen Menge G_i hinzugefügt, für welche sich der maximale Parallelisierungsgewinn $win(G_i, t)$ unter den Mengen $G_1 \ldots G_{vm}$ ergibt. Der Schätzwert für den erwarteten Parallelisierungsgewinn wird dabei für diejenige Menge besonders groß, welche schon Aktivitäten enthält, die gemeinsame Objekte mit der noch zu verteilenden Aktivität benutzen.

Den Aktivitäten aus der Menge G_i ist damit die virtuelle Maschine mit der Nummer *i* zugewiesen. Die resultierende Objektverteilung, die jedem Objekt *a* die Nummer derjenigen virtuellen Maschine *JVM(a)* zuweist, auf der es angelegt werden soll, ergibt sich daraus folgendermaßen:

$$\text{JVM(a)} = \text{i} \Leftrightarrow \text{activity(a)} \in G_i$$

Das Ergebnis der Verteilungsanalyse ist jetzt eine Abbildung von entfernten Objekten des Programms auf die Nummer der virtuellen Maschine, auf der das entsprechende Objekt angelegt werden soll. Um die Verteilungsstrategie zu implementieren, muß an jeder Stelle der Erzeugung eines entfernten Objekts die entsprechende Information eingefügt werden, auf welcher virtuellen Maschine es angelegt werden soll. Aktivitäten müssen nicht mehr gesondert betrachtet werden, da sie durch ihr Threadobjekt repräsentiert sind. Das Threadobjekt einer Aktivität wird dem Adreßraum zugeordnet, der für die jeweilige Aktivität vorgesehen ist. Die zur Durchführung der Objektverteilung während der Laufzeit nötigen Programmtransformationen werden im nächsten Kapitel beschrieben.

5 Programmtransformation

Der Sprachkonvention aus 4.1 folgend bedeutet die Verteilung von Objekten immer die Zuweisung einer ganzen Klasse von Objekten an eine bestimmte virtuelle Maschine im verteilten JavaParty-Adreßraum. Die Klasse von Aktivitäten ist dadurch charakterisiert, daß sie all diejenigen Objekte umfaßt, die derselben Klassenkontur angehören. Die Unterteilung von Klassen des JavaParty-Programms in Klassenkonturen ist das Ergebnis des Typinferenzalgorithmus aus Kapitel 2. Klassenkonturen sind eine Verfeinerung der durch die Java-Klassen vorgegebenen Partitionierung der Laufzeitobjekte. Zu Beginn der Typinferenz existiert für jede Java-Klasse eine Klassenkontur; die Partitionierung durch die Klassenkonturen entspricht also der durch die Java-Klassen. Im Verlauf der Typanalyse wird diese Partitionierung nach und nach verfeinert. Ein Verfeinerungsschritt besteht aus dem Teilen einer Klassenkontur in eine Anzahl neuer Konturen. Dies geschieht, wie in 2.10.2 beschrieben, durch Partitionierung der Erzeugungspunkte einer Klassenkontur. Die new-Anweisungen, an denen ein Objekt der zu teilenden Klassenkontur erzeugt wird, werden selbst partitioniert. Für die zu teilende Klassenkontur werden so viele neue Klassenkonturen erzeugt, wie die Partition der new-Anweisungen Elemente umfaßt. Daraufhin werden die an den new-Anweisungen entstehenden Klassenkonturen so geändert, daß new-Anweisungen aus unterschiedlichen Partitionen unterschiedliche Klassenkonturen erzeugen. Daraus folgt, daß die Unterteilung von Java-Klassen in Klassenkonturen eine Partitionierung nach der Stelle ihrer Erzeugung sind. An einer bestimmten new-Anweisung wird demnach immer genau eine Klassenkontur erzeugt.

Da die Verteilungsanalyse jeder Klassenkontur einer Fernklasse eine virtuelle Maschine zuweist, auf der Instanzen dieser Klassenkontur angelegt werden, kann bei jeder Objekterzeugung die Zielmaschine angegeben werden.

Allerdings teilt die Typanalyse auch Methoden in Methodenkonturen. Diese Methodenkonturen unterscheiden sich von ihrer Struktur her nicht. Der Code im Rumpf zweier Konturen derselben Methode ist bis auf zwei Ausnahmen gleich. Erstens kann, wie oben erwähnt, an zwei korrespondierenden new-Anweisungen in zwei Methodenkonturen eine unterschiedliche Klassenkontur erzeugt werden. Zweitens können sich die Ziele von Methodenaufrufen in zwei Konturen derselben Methode unterscheiden. Die Ziele eines Methodenaufrufs nach der Typanalyse sind keine Methoden mehr, sondern selber wieder Methodenkonturen. Man erkennt, daß sich die Verteilungsstrategie nicht direkt implementieren läßt, da zwar unterschiedliche Klassenkonturen immer an unterschiedlichen new-Anweisungen erzeugt werden, sich diese new-Anweisungen aber nur in der Methodenkontur unterscheiden müssen. Es ist daher nach der Typanalyse nicht möglich, Methodenkonturen wieder mit Methoden zu identifizieren und nur an new-Anweisungen entsprechende Hinweise für das Laufzeitsystem zur richtigen Objektverteilung einzufügen.

Für die Transformation des Programms stehen zwei unterschiedliche Wege offen. Einmal wäre es möglich, die Klassenkonturen einer Java-Klasse *A* durch Unterklassen von *A* zu realisieren und die Methodenkonturen durch unterschiedliche Methoden explizit zu implementieren. Um nicht in Konflikt mit der durch das Programm vorgegebenen Vererbungshierarchie zu geraten, müßte man aus einer Java-Klasse *A* mit Konturen $A^1,\ldots, A^n$ eine abstrakte Oberklasse *A*, welche Signatur und Instanzvariablen der ursprünglichen Klasse deklariert, und Unterklassen *A$1,… A$n* generieren, die die Klassenkonturen darstellen und die Implementation der Methodenkonturen bereitstellen. Dieser Ansatz hat den Vorteil, daß die Unterklassen *A$1,… A$n* mit dem Schlüsselwort *final* deklariert werden können. Der durch die Typinferenz erzeugte statische Kontrollfluß ließe sich dann insofern ausnutzen, als der Aufruf einer Methode einer finalen Klasse erheblich schneller ausgeführt wird, da man den dynamischen Dispatch, also das Ermitteln der auszuführenden Methode zur Laufzeit, einsparen kann. Die Expansion der Klassenkonturen zu eigenständigen Klassen und die explizite Implementation der Methodenkonturen bedeutet unter Umständen eine Vervielfachung der resultierenden Programmgröße. Da das Hauptaugenmerk dieser Arbeit auf der Entwicklung der Verteilungsstrategie durch statische Analyse liegt und nicht die Vermeidung des dynamischen Dispatch im

Vordergrund steht, wird diese Variante nicht weiter vertieft. Die zweite Variante, die weitgehend auf Codevervielfachung verzichtet und die Anzahl der Klassen des Programms beibehält, soll hier vorgestellt werden.

5.1 Randbedingungen der Transformation

Die Analyse der konkreten Typen und die darauf aufbauende Berechnung der Verteilungsstrategie ist eine globale Operation, die sich auf alle am Programm beteiligten Klassen erstreckt. Das heißt, es müssen prinzipiell alle Klassen, also auch solche, die in vorübersetzten Bibliotheken enthalten sind, in die Analyse mit einbezogen werden. Dies liegt daran, daß die konkreten Typen von Methoden und Instanzvariablen einer Klasse nicht allein von der Klasse selbst abhängen, sondern sich je nach Verwendung der Klasse im umgebenden Programm ändern. So ändert sich der konkrete Typ von Zugriffsmethoden einer Behälterklasse je nach dem, welche Objekte in ihren Instanzen gespeichert werden. Ein gutes Beispiel ist die Klasse *java.lang.Thread* der Java-Klassenbibliothek bzw. die Klasse *jp.lang.RemoteThread* der JavaParty-Bibliothek. Objekte beider Klassen enthalten eine Instanzvariable vom definierten Typ *java.lang.Runnable*, in der das Objekt gespeichert wird, dessen run-Methode die neue Aktivität ausführt. Das Objekt, das in dieser Instanzvariablen gespeichert wird, wird aber nicht innerhalb der Behälterklasse bestimmt sondern durch die Verwendung in ihrer Umgebung. Das in der Instanzvariable gespeicherte Objekt bestimmt deren konkreten Typ und damit auch die beim Start der Aktivität ausgeführte Methodenkontur. Erst die konkrete Verwendung einer Klasse bestimmt also den konkreten Typ ihrer Methoden und Instanzvariablen.

Die Typinferenz darf daher nicht an Bibliotheksschranken halt machen; sie muß das gesamte Programm analysieren, um korrekte konkrete Typen bestimmen zu können. Diese Forderung ließe sich am einfachsten erfüllen, indem man verlangt, daß stets alle am Programm beteiligten Klassen mitübersetzt werden. Leider würde diese Einschränkung zu keinem praktikablen Übersetzer führen. Ein Ausweg wird in 5.6 vorgestellt, der es einerseits ermöglicht, alle benutzten Klassen und deren Methoden zu analysieren, aber nicht verlangt, daß alle verwendeten Klassen mitübersetzt werden, also auch nicht im Quellcode vorliegen müssen.

Im Moment ist nur wichtig, daß zwar alle Klassen in die Analyse einbezogen werden, aber nicht auch für alle Klassen Code erzeugt wird. Die Implementierung von Klassen- und Methodenkonturen ist demnach nur für Klassen möglich, die auch mitübersetzt werden. Diese Klassen heißen im folgenden veränderbare Klassen. Solche, die schon vorübersetzt sind, werden als unveränderbare Klassen bezeichnet.

5.2 Methodenkonturen

Zur Implementation von Methodenkonturen soll nicht der Code für jede Kontur einer Methode dupliziert werden. Die Konturen einer Methode *f* werden durch Zahlen unterschieden und durch eine Methode *f$contour* implementiert. Ihr wird die Nummer der aktuellen Kontour als zusätzlicher Parameter *method$contour* übergeben.

ursprüngliche Methode	***nach der Transformation***

`T foo(P) {`
`  <body>`
`}`
	`T foo$contour(int method$contour, P) {`
	`  <transform(body)>`
	`}`
	
	`T foo(P) { <call foo$contour> }`

Abbildung 37
Implementation der Methodenkonturen

Aufrufe von *f* werden in Aufrufe von *f$contour* geändert. Da das nur in Methoden von veränderbaren Klassen möglich ist, muß in der transformierten Klasse eine Methode der ursprünglichen Signatur erhalten bleiben. Diese Methode wählt dann die Methodenkontur aus und leitet den Aufruf an f$contour weiter. Da in diesem Fall der Aufrufer zur Auswahl der Kontur keine Information beisteu-

ern kann, ist die Auswahl der Methodenkontur nur aufgrund der Klassenkontur des Objekts möglich. Die Auswahl der Methodenkontur in diesem Fall wird weiter unten erläutert. Abbildung 37 zeigt die Transformation einer Methode *foo* mit Ergebnistyp *T* und Parameterliste *P*.

5.3 Methodenaufruf

Ist der Kontrollfluß an einem Methodenaufruf eindeutig, kann der ursprüngliche Aufruf durch den direkten Aufruf der die Methodenkontur implementierenden Methode ersetzt werden.

ursprünglicher Methodenaufruf	*transformierter Methodenaufruf*
`x.foo(q) // Aufruf von Kontur 3` `  // der Methode foo`	`x.foo$contour(3, q)`

Abbildung 38
Transformation eines Methodenaufrufs (vereinfacht)

Abbildung 38 vereinfacht die Situation der Transformation eines Methodenaufrufs in zweierlei Hinsicht. Es soll nicht für jede einzelne Kontur der aufrufenden Methode eigener Code erzeugt werden. Die nach der Transformation resultierende Anweisung muß in Abhängigkeit der aktuellen Methodenkontur die richtige Kontur der aufgerufenen Methode auswählen. Da sich die aus unterschiedlichen Konturen heraus aufgerufenen Methodenkonturen aber in der Regel unterscheiden, auch wenn der Aufruf in jeder einzelnen Kontur eindeutig ist, kann die gerufene Kontur nicht direkt über ihre Nummer ausgewählt werden. Statt dessen muß die gerufene Methodenkontur in Abhängigkeit der aktuellen Methodenkontur und der Nummer der Anweisung in einer Tabelle nachgeschlagen werden. Abbildung 39 berücksichtigt diesen Aspekt. Die aufrufende Methode in diesem Beispiel heißt *caller*. Bei ihrer Transformation wird eine Tabelle *caller$apply* angelegt, die für jede Methodenkontur von *caller* und jeden Methodenaufruf innerhalb von *caller* die Nummer der aufgerufenen Methodenkontur angibt. Der transformierte Methodenaufruf schlägt dann anhand der aktuellen Methodenkontur von *caller* und der Nummer des Methodenaufrufs innerhalb von *caller* die Nummer der aufzurufenden Methodenkontur von *foo* nach.

Noch ein weiterer Umstand wurde in Abbildung 38 außer Acht gelassen, der in Abbildung 39

ursprünglicher Methodenaufruf

```
T caller(P) {
  …
  x.foo(q)  // In den Konturen 0, 1, 2 von
            // caller sollen die Konturen
  // 3, 1, 4 von foo aufgerufen werden.
  // x sei deklariert vom Typ A (eine nicht
  // veränderbare Klasse).
  // In den Konturen 0 und 1 von caller habe
  // x den konkreten Typ {B}, in der
  // Kontur 2 den konkreten Typ {C}.
  // B und C seien veränderbare direkte
  // Unterklassen von A
  …
}
```

transformierter Methodenaufruf

```
private static final int[][]
  caller$apply = {{…3…}, {…1…}, {…4…}};

private static final int[]
  caller$code = {0, 0, 1};

T caller$contour(int method$contour, P) {
  switch(caller$code[method$contour]) {
    case 0: {
      …
      ((B) x).foo$contour(
        caller$apply[method$contour]
          [ <Nr. des Methodenaufrufs> ],
        q)
      …
    }
    case 1: {
      …
      ((C) x).foo$contour(
        caller$apply[method$contour]
          [ <Nr. des Methodenaufrufs> ],
        q)
      …
    }
  }
}

T foo(P) { <call foo$contour> }
```

Abbildung 39
Transformation eines Methodenaufrufs

berücksichtigt ist. Um eine Kontur direkt aufzurufen, muß die Signatur des Aufrufs von *x.foo(…)* in *x.foo$contour(…)* geändert werden. Je nach deklariertem Typ der Variablen *x* kann diese Änderung Probleme verursachen. Nimmt man an, der deklarierte Typ von *x* sei *A*. In *A* ist wegen der Typkorrektheit des Programms vor der Transformation eine Methode *foo* deklariert. Ist aber *foo* in *A* eine abstrakte Methode, A eine Schnittstelle, oder *A* eine nicht veränderbare Klasse, so gibt es in *A* keine Deklaration von *foo$contour*. Der Aufruf nach der Transformation wäre also nicht mehr typkorrekt. Diese Situation ist möglich, da die Typanalyse einen konkreten Typ für *x* berechnet hat und damit bekannt ist, welcher Klasse das Laufzeitobjekt in *x* angehören wird. Diese Klasse muß in diesem Fall eine Unterklasse von *A* sein, bzw. die Schnittstelle *A* implementieren. Da der Laufzeittyp von *x* bekannt ist, kann eine entsprechende Typkonversion eingefügt werden. Mit dieser Typkonversion läßt sich dann *foo$contour* direkt aufrufen.

Eine Typkonversion ist aber anders als die Nummer der aufgerufenen Kontur nicht parametrisierbar. Sind in verschiedenen Konturen des Aufrufers unterschiedliche Typkonversionen nötig, so läßt sich für diesen Aufruf kein gemeinsamer Code für beide Methodenkonturen erzeugen. Die Transformation vervielfacht in diesem Fall den Code aus dem Rumpf der Ausgangsmethode. Die Auswahl der zur Methodenkontur passenden Version des Codes geschieht über eine weitere Tabelle *caller$code*. Abbildung 39 berücksichtigt diese Aspekte. Im Beispiel soll die den Aufruf umgebende Methode *caller* drei Konturen haben. Der konkrete Typ von *x* soll sich in den Konturen mit der Nummer 0, 1 und 2 unterscheiden. In allen drei Konturen ist eine Typkonversion notwendig. Der Code aus dem Rumpf von *caller* wird verdoppelt und der Aufruf von *foo* mit den jeweils passenden Typkonversionen versehen.

5.3.1 Unscharfe Aufrufe

Nicht an allen Stellen kann die Typanalyse einen statischen Kontrollfluß erzeugen. In 2.11 werden solche Fälle von nicht auflösbaren Unschärfen behandelt. Eine Kontrollflußunschärfe liegt dann vor, wenn an einem Methodenaufruf in einer Kontur mehrere Methodenkonturen als Ziel in Frage kommen. Dann kann statisch nicht entschieden werden, welche der Konturen zur Laufzeit auszuwählen sind.

Wenn für einen Methodenaufruf mehrere Konturen selektiert werden, kann das entweder daran liegen, daß die Variable, mit der die Methode aufgerufen wird, oder einer der Parameter unscharf ist (vgl. 2.10.5). Zur Laufzeit findet ein Dispatch aber nur nach der Klasse des Objekts statt, das sich in der Variablen befindet, mit der die Methode aufgerufen wurde (this-Argument), nicht aber nach den Methodenparametern. Geht die Mehrdeutigkeit eines Methodenaufrufs auf eine Unschärfe in einer Parametervariablen zurück, kann diese Unterscheidung der gewählten Methodenkonturen zur Laufzeit nicht nachgebildet werden. Solche Unschärfen sind nicht implementierbar. Die betroffenen Methodenkonturen müssen daher wieder zu einer Kontur vereinigt werden.

Hat die Mehrdeutigkeit der Methodenkonturselektion ihre Ursache in einem unscharfen this-Argument, läßt sich die Wahl der richtigen Methodenkontur zur Laufzeit trotzdem nicht allein über den dynamischen Dispatch durchführen. Klassenkonturen werden bei diesem Ansatz nicht als separate Klassen implementiert, sondern ähnlich wie Methodenkonturen nur über eine Nummer unterschieden. Jedes Objekt erhält dazu eine zusätzliche Instanzvariable *this$contour*, die die Nummer der Klassenkontur dieses Objekts speichert. Daher sind für den dynamischen Dispatch zur Laufzeit Objekte zweier Konturen derselben Klasse nicht unterscheidbar. Die Auswahl der richtigen Kontur für einen Methodenaufruf wird also in zwei Schritten durchgeführt.

5.3.2 Namensaufruf

Im ersten Schritt wird über den dynamischen Dispatch (wie bei jedem normalen Methodenaufruf) die richtige Version der Methode ausgewählt, die die Konturen der ursprünglichen Methode implementiert. Im Beispiel aus Abbildung 39 wird der Aufruf der Methode *foo* durch den Aufruf von *foo$contour* ersetzt. So wie beim ursprünglichen Aufruf zur Laufzeit die richtige überschriebene

Version von *foo* ausgewählt wird, geschieht das nach der Transformation auch mit *foo$contour*. Dafür gibt es allerdings zwei Voraussetzungen. Erstens müssen alle selektierten Methodenkonturen zu Methoden gehören, die in mitübersetzten Klassen definiert wurden, da zu einer Methode *foo* aus einer unveränderbaren Klasse keine entsprechende Methode *foo$contour* existiert. Ist eine solche Methode unter den für einen Aufruf selektierten Methodenkonturen, muß an dieser Stelle ein default-Aufruf durchgeführt werden (siehe 5.3.3).

Auch wenn alle selektierten Methodenkonturen einer mitübersetzten Klasse angehören, kann es trotzdem sein, daß der direkte Aufruf von *foo$contour* nicht möglich ist. Nennt man die Klassen, in denen die Methoden deklariert sind, deren Konturen am Aufruf beteiligt sind „Zielklassen", so muß in einer gemeinsamen mitübersetzten Oberklasse aller Zielklassen eine Definition von *foo* (und damit auch von *foo$contour*) existieren. Die Existenz einer solchen Oberklasse ist wichtig, da man sie für den typkorrekten Aufruf von *foo$contour* in der Typkonversion (siehe 5.3) benötigt. Existiert sie nicht, muß ebenfalls ein default-Aufruf durchgeführt werden (siehe 5.3.3).

Bis hierher wurde lediglich die Zulässigkeit der Änderung der Signatur des Methodenaufrufs von *foo(q)* in *foo$contour(n, p)* diskutiert, unter der Voraussetzung, daß die aufgerufene Kontur eindeutig ist. Ist diese Kontur nicht eindeutig, kann auch ihre Nummer dem Aufruf von *foo$contour* nicht mitgegeben werden. Anstatt der Nummer der aufzurufenden Kontur wird ein sog. Name übergeben. Aufgrund des Namens und der Klassenkontur des Objekts, dessen Methode aufgerufen wird, kann die Implementation von *foo$contour* durch Nachschlagen in der Tabelle *foo$vct* die richtige Kontur auswählen. Als Namen werden ebenso wie bei Konturen Nummern verwendet. Um Namen und Konturen zu unterscheiden, werden für Namen negative, für Konturen positive Zahlen reserviert. Der Aufruf einer Kontur direkt oder über einen Namen kann dann dieselbe Signatur *foo$contour(n, p)* verwenden. Es ist dann nicht nötig noch eine weitere Methode für den Namensaufruf zu generieren; nur das Vorzeichen der übergebenen Zahl entscheidet, ob in *foo$contour* ein weiterer Indirektionsschritt zur Auswahl der richtigen Kontur notwendig ist. Für einen mehrdeutigen Aufruf wird also ein neuer Name angelegt und in der Tabelle *foo$vct* für jeden Namen und jede Klassenkontur die entsprechende Nummer der auszuführenden Methodenkontur gespeichert.

Namensaufruf

```
x.foo$dispatch(-1, q)
  // x enthält ein Objekt der
  // Klassenkontur 0, 1, oder 2
  // es werden dann die Konturen
  // 1, 4 bzw. 2 von foo aufgerufen

x.foo$dispatch(-2, q)
  // x enthält ein Objekt der
  // Klassenkontur 0, 1, oder 2
  // es werden dann die Konturen
  // 1, 0 bzw. 3 von foo aufgerufen
```

Namensauflösung

```
private static int[][]
  foo$vct = {
    {1, 4, 2},  // Tabelle für Namen -1
    {1, 0, 3}   // Tabelle für Namen -2
  }

S foo$contour(int method$contour, Q) {
  if (method$contour < 0) {
    // es handelt sich um einen Namen

    method$contour =
      foo$vct[-method$contour-1][this$contour];
  }

  …
}
```

Abbildung 40
Namensauflösung (vereinfacht)

Bei der Namensauflösung, wie sie in Abbildung 40 implementiert ist, wird folgendes noch nicht berücksichtigt: Die konkreten Typen von *x* mögen sich nicht nur in der Kontur ihrer Klasse, sondern auch in der Klasse selbst unterscheiden. *x* sei vom konkreten Typ $\{A^0, A^1, A^2, B^0\}$, dabei sei *B* eine Unterklasse von *A*, die keine eigene Version von *foo* implementiert. Die abgebildete Version von *foo* bzw. *foo$contour* sei in der Klasse *A* definiert. Unabhängig vom Laufzeittyp von *x* wird also die abgebildete Version von *foo$contour* aufgerufen. Die Nummern der Klassenkonturen sind aber nur pro Klasse eindeutig. Innerhalb von *foo$contour* kann in der Tabelle *foo$vct* nur aufgrund der Nummer der Klassenkontur des Objekts in *this* die Nummer der auszuführenden Methodenkontur nachgeschlagen werden. Die Auswahl der richtigen Methodenkontur muß aber zusätzlich auch die

Klasse des Objekts berücksichtigen. Wie oben erwähnt, kann die Methode *foo$contour* auch mit Unterklassen von *A* aufgerufen werden. An der Nummer (z.B. *0*) der Kontur läßt sich dann nicht mehr entscheiden, ob es sich um ein Objekt der Klassenkontur A^0 oder B^0 handelt. Da in beiden Fällen aber unterschiedliche Methodenkonturen ausgeführt werden sollen, muß die Namensauflösung in einer eigenen Methode durchgeführt werden. Abbildung 41 berücksichtigt diesen Umstand. Die Auflösung des Namens geschieht in einer neuen Methode *foo$dispatch*, die in allen Klassen überschrieben wird, die *A* erweitern und von denen Objekte an einem Namensaufruf von *foo$contour* beteiligt sind. In *foo$contour* wird dann anhand des Vorzeichens von *method$contour* entschieden, ob es sich um einen Namensaufruf handelt. Wenn ja wird der Name in einem Aufruf von *foo$dispatch* durch Nachschlagen in der Tabelle *foo$vct* in eine Methodenkontur aufgelöst.

```
class B extends A {
  private static int[][]
    foo$vct = {
      {5},  // Tabelle für Namen -1
      {6}   // Tabelle für Namen -2
    }
  int foo$dispatch(int name) {
    return foo$vct[name][this$contour];
  }
}

class A {
  private static int[][]
    foo$vct = {
      {1, 4, 2},  // Tabelle für Namen -1
      {1, 0, 3}   // Tabelle für Namen -2
    }
  int foo$dispatch(int name) {
    return foo$vct[name][this$contour];
  }

  S foo$contour(int method$contour, Q) {
    if (method$contour < 0) {
      // es handelt sich um einen Namen
      method$contour =
        foo$dispatch(-method$contour-1);
    }
    …
  }

  S foo(Q) {
    // Aufruf mit Default-Namen -1
    foo$contour(-1, Q);
  }
}
```

Abbildung 41
Namensauflösung und Default-Aufruf

5.3.3 Default-Aufruf

Der Default-Aufruf wird an allen Stellen verwendet, an denen die Signatur des Methodenaufrufs nicht verändert werden kann. Das betrifft Aufrufe in nicht veränderbarem Code und solche, wo keine Definition der neuen Signatur in einer gemeinsamen veränderbaren Oberklasse der Laufzeitobjekte existiert, die an dem Aufruf beteiligt sind.

Ein Default-Aufruf von *foo* erreicht nach der Transformation immer noch die Methode mit derselben Signatur *foo*. Diese Methode enthält nach der Transformation aber nicht mehr den ursprünglichen Code, sondern einen Aufruf von *foo$contour*. Damit an Stellen mit Default-Aufruf die Methodenkontur wenigstens in Abhängigkeit der Klassenkontur des jeweiligen Objekts stattfindet, wird für alle Default-Aufrufe einer Methode ein spezieller Name (*-1*) als Default Name reserviert.

Der Default Aufruf wird immer mit diesem eindeutigen Namen ausgeführt, wogegen beim Namensaufruf an jeder Aufrufstelle von *foo* ein neuer Name angelegt wird. Ansonsten gibt es aber keine Unterschiede zwischen Namens- und Default-Aufruf.

5.4 Klassenkonturen

Wie oben bereits erwähnt, werden Klassenkonturen über eine pro Klasse eindeutige Nummer identifiziert. Sie dient bei der Implementation von nicht auflösbaren Unschärfen zur Auswahl unter den selektierten Methodenkonturen. Die Klassenkontur wird pro Objekt in einer zusätzlichen Instanzvariable *this$contour* gespeichert. Das ist selbstverständlich nur bei mitübersetzten Klassen möglich. Bei Instanzen von unveränderbaren Klassen ist die Kontur nur an der Stelle der Objekterzeugung bekannt. Das ist ausreichend, um das Objekt auf der richtigen Zielmaschine zu plazieren. Später wird die Klassenkontur eines Objekts einer nicht mitübersetzten Klasse nicht mehr benötigt, da keine Methodenkonturen aufgrund der Klassenkontur des Objekts ausgewählt werden müssen. Wird eine Klasse nicht mitübersetzt, so existiert im Programm nur genau eine Kontur pro Methode.

5.4.1 Konstruktoraufrufe

Bei mitübersetzten Klassen muß bei der Objekterzeugung die Klassenkontur des erzeugten Objekts mit angegeben werden. Dafür werden Konstruktoraufrufe so transformiert, daß neben der Methodenkontur *method$contour* des Konstruktors auch die Klassenkontur der erzeugten Objekts *object$contour* mitübergeben wird.

Klasse mit Konstruktor

```
class A extends B {
  A(P) {
    super(q);
    …
  }
}
```

Klasse und Konstruktor nach der Transformation

```
class A extends B {
  int this$contour;

  A(int method$contour, int object$contour, P) {
    super(q);
    this$contour = object$contour;
    …
  }
}
```

Abbildung 42
Transformation eines Konstruktors

Die Klasse *A* aus Abbildung 42 erweitert eine nicht mitübersetzte Klasse *B*. Sie erhält die zusätzliche Instanzvariable *this$contour*, die in ihrem Konstruktor nach dem Aufruf des super-Konstruktors initialisiert wird. Die Transformation des restlichen Rumpfes erfolgt wie bei normalen Methodenkonturen. Die Implementation eines Default-Konstruktors analog zur Default-Kontur eines Methodenaufrufs ist nicht möglich, da der Name des Konstruktors immer der Klassenname ist. Ein Default-Konstruktor würde daher zu Namenskonflikten führen, da er sich nur durch die Signatur seiner Parameter unterscheiden würde. Außerdem ist ein Default-Konstruktor gar nicht möglich, da die Klassenkontur bei der Objekterzeugung bekannt sein muß. Er ist auch nicht nötig, wenn man verlangt, daß eine Klasse *C*, die Objekte einer mitübersetzten Klasse *A* erzeugt, selbst auch mitübersetzt wird. Dann können nämlich alle Konstruktoraufrufe von *A* entsprechend transformiert werden. Dies stellt keine echte Einschränkung dar, da eine solche Klasse *C* nicht in einer Bibliothek definiert sein kann, zu der kein Quellcode vorliegt. Beim Übersetzen von *C* muß die Definition von *A* schon vorliegen.

Wird die Klasse *B* aus Abbildung 42 selbst auch mitübersetzt, ist schon in ihr die Instanzvariable *this$contour* definiert. Der Parameter *object$contour* des Konstruktors wird dann nur an den super-Konstruktor weitergereicht.

5.5 Objekterzeugung

Abbildung 43 zeigt die Transformation einer Instanzierung einer mitübersetzten Fernklasse. Der new-Ausdruck wird transformiert in eine Sequenz von zwei Anweisungen. Die erste wählt mit *setTarget()* die Zielmaschine für diese Objekterzeugung aus. Die zweite führt die eigentliche Objekterzeugung durch. Dem Konstruktor wird dabei zusätzlich neben der Methodenkontur auch noch die Klassenkontur des zu erzeugenden Objekts mitgegeben. Die Nummer der virtuellen Maschine und der Klassenkontur werden in den für die Methode *foo* generierten Tabellen *foo$newAt* bzw. *foo$new* nachgeschlagen. Die Auswahl der Methodenkontur des Konstruktors geschieht wie beim Methodenaufruf.

Die Sequenz aus Auswahl der Zielmaschine und anschließender Erzeugung des Objekts wird über einen *seq*-Block, ein Konstrukt der erweiterten Syntax von Espresso Grinder, wieder zu einem Ausdruck zusammengefaßt, der den ursprünglichen new-Ausdruck in jedem Kontext ersetzten kann. Die Bedeutung von *seq* <*block*> ist dabei, daß die Anweisungen des Blocks ausgeführt werden und das mit *return* zurückgelieferte Ergebnis des Blocks als Wert des gesamten Ausdrucks betrachtet wird. Um ein entferntes Objekt auf einer bestimmten Maschine anzulegen, macht die Transformation nur von den durch die verteilte Laufzeitumgebung bereitgestellten Möglichkeiten Gebrauch. Dies geschieht in erster Linie, damit zur Implementation nicht in das Konzept von JavaParty eingegriffen werden muß.

Objekterzeugung	*Transformierte Objekterzeugung*

`new A(p)`	`seq {`
	`  DistributedRuntime.setTarget(`
	`  foo$newAt[method$contour][ <Nummer d. Objekterzeugung> ],`
	`  );`
	`  return new A(`
	`  foo$apply[method$contour][ <Nummer d. Methodenaufrufs> ],`
	`  foo$new[method$contour][ <Nummer d. Objekterzeugung> ],`
	`  p`
	`  );`
	`}`

Abbildung 43
Instanzierung einer mitübersetzten Fernklasse

Die durch das JavaParty Laufzeitsystem zur Verfügung gestellten Möglichkeiten zur Objektverteilung sind aber in mehrerlei Hinsicht unzulänglich. Der Aufruf von *setTarget(<JVM Nummer>)*, der die virtuelle Maschine für nachfolgende Objekterzeugungen festlegt, ist einerseits ineffizient und kann andererseits unvorhersehbare Ereignisse produzieren. Versuchen zwei Aktivitäten im selben Adreßraum gleichzeitig eine Objekterzeugung durchzuführen, so setzt zuerst die erste Aktivität das Ziel der Objekterzeugung fest, worauf das Laufzeitsystem möglicherweise zwischen den Aktivitäten umschaltet. Danach wählt die zweite Aktivität das Ziel für ihre Objekterzeugung. Nach erneutem Umschalten legt die erste Aktivität dann ihr Objekt in dem Adreßraum an, den die zweite Aktivität für ihr Objekt ausgesucht hatte.

Bei näherer Betrachtung erkennt man eine weitere Schwachstelle dieses Vorgehens der Trennung von Bestimmung der Zielmaschine und eigentlicher Objekterzeugung. Transformiert man zwei ineinander geschachtelte new-Anweisungen, bei denen die innere als Parameter der äußeren auftritt wie in Abbildung 44, hat die Transformation nicht den gewünschten Effekt. Der zum Setzten der Zielmaschine in der inneren Sequenz notwendige Seiteneffekt bleibt nach der inneren Objekterzeugung bestehen und überschreibt das Setzten der Zielmaschine vor der Durchführung der äußeren Objekterzeugung. Da es in der vorliegenden Version von JavaParty keine weiteren Alternativen zur Auswahl der Zielmaschine einer Objekterzeugung gibt, muß man in diesem Fall die unbeabsichtigte Plazierung des Objekts *A* in Kauf nehmen oder komplexere „Umbauarbeiten“ an geschachtelten Objekterzeugungen vornehmen.

Ausdruck	*Transformation*	*Ausführungsreihenfolge*

`new A(new B())`	`seq {`
	`  setTarget(2);`
	`  return new A(seq {`
	`  setTarget(3);`
	`  return new B();`
	`  });`
	`}`
		`setTarget(2);`
		`  // Auswerten der Parameter`
		`  setTarget(3);`
		`  tmp = new B();`
		`new A(tmp);`

Abbildung 44
Transformation mit unerwünschtem Effekt

Eine andere Möglichkeit zur Wahl der virtuellen Maschine für eine Objekterzeugung, besteht darin, einen sog. Objekt-Distributor anzugeben, bei dem das Laufzeitsystem vor jeder Instanzierung anfragt, auf welcher Maschine das Objekt angelegt werden soll. Allerdings muß der Objekt-Distributor in der gegenwärtigen Implementation von JavaParty diese Entscheidung aufgrund des Namens der zu instanzierenden Klasse fällen, den er als Zeichenkette übergeben bekommt. Da die Klassenkonturen aber nicht als eigenständige Klassen realisiert werden, scheidet diese Möglichkeit von vornherein aus. Mit dem Namen der Klasse alleine kann der Objekt-Distributor nicht die Plazierungsentscheidungen fällen, die zur Realisation der berechneten Verteilungsstrategie notwendig sind. Bedenkt man, daß auch von nicht mitübersetzten Klassen Klassenkonturen erzeugt werden können, den erzeugten Instanzen aber ihre Klassenkontur nicht mehr anzusehen ist, kommt man zu dem Schluß, daß das Konzept des Objekt-Distributors noch weniger zur Implementation der Verteilungsstrategie brauchbar ist.

Mitübersetzte Klassen könnten in ihrer Handle-Klasse aufgrund der an den Konstruktor ohnehin übergebenen Klassenkontur selbständig über die Zielmaschine entscheiden, auf der die Instanz angelegt werden soll, oder zumindest dem Objektdistributor die Information über die zu erzeugende Klassenkontur zusätzlich zur Verfügung stellen. Dadurch ließen sich die oben angesprochenen Probleme lösen, da zur Objekterzeugung auf einer bestimmten virtuellen Maschine keine globalen Variablen mehr manipuliert werden müßten. Allerdings wäre ein Eingriff in die Objekterzeugung des JavaParty-Systems nötig. Vorübersetzte Klassen lassen sich auf diese Weise nicht behandeln, da an ihren Konstruktor die Nummer der Klassenkontur nicht übergeben wird. (Zum Zeitpunkt ihrer Übersetzung wurden von ihnen noch keine Konturen angelegt.) Um ohne Seiteneffekte in globalen Variablen auch von diesen Klassen Instanzen auf unterschiedlichen virtuellen Maschinen anlegen zu können, müßte die Transformation zur Erzeugung der Handle-Klassen zusätzliche Konstruktoren generieren, denen die Nummer der Zielmaschine als weiterer Parameter übergeben werden kann. Solche zusätzlichen Konstruktoren sind bis jetzt nur in der Klasse *RemoteThread* realisiert, weswegen man in der gegenwärtigen Implementierung die genannten Einschränkungen in Kauf nehmen muß.

5.6 Bytecode Disassembler

Die Typinferenz muß das gesamte Programm analysieren, aber die Forderung, bei der Übersetzung immer alle verwendeten Klassen mitzuübersetzen, ist wenig praktikabel. Zum einen stehen unter Umständen nicht alle verwendeten Bibliotheken im Quellcode zur Verfügung, zum anderen ist bei der Übersetzung eines Programms gar nicht klar, welche Teile der Bibliothek überhaupt mitübersetzt werden müssen, da meist nicht einzelne Bibliotheksklassen importiert werden sondern ganze Pakete. Die Verwendung der Bibliotheksklassen untereinander bleibt dem Benutzer der Bibliothek ohnehin verborgen. Die Einschränkung, daß bei jedem zu übersetzenden Programm z.B. die ganze Java-Klassenbibliothek mitübersetzt werden muß, stellt eine zu große Beschränkung der Praktikabilität des Verfahrens dar. Die Möglichkeit, schon bei der Übersetzung einer Bibliothek die benötigten Datenflußinformation zu extrahieren und zur Wiederverwendung bei der Übersetzung eines die Bibliothek verwendenden Programms separat zu speichern, wurde ebenfalls verworfen, da das nur bei Bibliotheken möglich wäre, die im Quellcode vorliegen, und einen zusätzlichen Verwaltungsaufwand bedeuten würde.

Eine viel elegantere Lösung eröffnet die Eigenschaft der Sprache Java und ihrer virtuellen Maschine, daß der Code, der aus einem Java-Programm zur Ausführung durch die virtuelle Maschine erzeugt wird, sehr viel von der Struktur des Quellprogramms beibehält. Der Java-Bytecode ist nicht mit dem Maschinencode eines herkömmlichen Mikroprozessors vergleichbar. Die virtuelle Maschine arbeitet ebenso wie die Sprache Java selbst mit Konzepten wie Klassen, Instanzvariablen, Objekterzeugung, Methoden und Methodenaufruf anstatt wie herkömmliche Prozessoren mit Speicherbereichen, Codesequenzen, Stapelverwaltung und Sprungbefehlen. Daher läßt sich aus übersetztem Java-Bytecode ohne allzu großen Aufwand wieder Java-Quellcode gewinnen, der von seiner Struktur her mit dem identisch ist, aus dem er generiert wurde. Eine solche Rücktransformation wurde im Übersetzer soweit realisiert, als nötig ist, um die für die Datenflußanalyse wichtigen Informationen aus einer schon übersetzten Bibliothek zu extrahieren. Der Bytecode selbst wird dabei als Repräsentation der für die Analyse wichtigen Datenflußinformationen gewählt; eine separate Speicherung wird damit unnötig.

6 Erweiterung des Typinferenzalgorithmus

In zweierlei Hinsicht liefert die in Kapitel 2 vorgestellte Typinferenz keine zufriedenstellende Grundlage für die Berechnung der Verteilungsstrategie. So wird als Voraussetzung für das Auflösen einer Unschärfe in einer Variablenkontur *vc* verlangt, das alle *vc* erreichenden Datenflußkanten eindeutige Werte abliefern. Ist diese Bedingung nicht erfüllt, kann zwar die Unschärfe in *vc* nicht ganz aufgelöst werden; sehr oft läßt sich die Anzahl der Datenflußwerte in *vc* dennoch reduzieren. Ohne die Erweiterung verhindert schon eine einzige nicht auflösbare Unschärfe in einer Instanzvariablen das Teilen der zugehörigen Klassenkontur, obwohl es neben der einen polymorphen Verwendung mehrere monomorphe Verwendungen dieser Instanzvariablen im Programm gibt. Die andere Erweiterung betrifft direkt die Aktivitäten eines Programms.

6.1 Splitting nach Aktivitäten

Die Typinferenz teilt eine Klassenkontur einer Klasse *A* nur dann, wenn ihre Instanzvariablen einen polymorphen Typ haben und sich dieser auf unterschiedliche monomorphe Verwendungen der Klasse im Programm zurückführen läßt. Benutzen unterschiedliche Aktivitäten Instanzen von *A* mit denselben Typen, wird *A* nicht in unterschiedliche Konturen geteilt. Da aber die Verteilungsstrategie nur Klassenkonturen verteilt, ist dann keine gute Lokalität möglich.

```
class B implements Runnable {
  public void run() {
    jp.util.RemoteVector v =
      new jp.util.RemoteVector();

    < ausgiebige Benutzung von v z.B
      mit Instanzen der Klasse
      java.lang.Integer >
  }
}

class C implements Runnable {
  public void run() {
    jp.util.RemoteVector w =
      new jp.util.RemoteVector();

    < ausgiebige Benutzung von w z.B
      auch mit Instanzen der Klasse
      java.lang.Integer >
  }
}
```

Abbildung 45
Verwendung in unterschiedlichen Aktivitäten

Die beiden Instanzen *v* und *w* der Klasse *jp.util.RemoteVector* aus Abbildung 45 sind nicht durch die konkreten Typen ihrer Instanzvariablen, sondern nur durch die Verwendung in unterschiedlichen Aktivitäten partitioniert. (Das Beispiel geht davon aus, daß die run-Methoden von *B* und *C* in unterschiedlichen Aktivitäten ausgeführt werden.) In beide Instanzen von *jp.util.RemoteVector* werden Objekte der selben Klasse *java.lang.Integer* gespeichert. Die Instanzvariable, die diese Objekte aufnimmt hat damit den scharfen konkreten Typ *java.lang.Integer*. Die Partitionierung wird bis jetzt von der Analyse noch nicht durch eine Teilung der Klassenkontur modelliert. Damit werden alle Instanzen von *jp.util.RemoteVector* auf derselben virtuellen Maschine angelegt. Dies führt zu keiner guten Lokalität.

Um nicht nur bei polymorpher Verwendung, sondern auch bei monomorpher Benutzung in unterschiedlichen Aktivitäten eine Teilung der Klassenkontur einer Klasse zu ermöglichen, muß die Analyse für jede Methodenkontur Information darüber gewinnen, in welcher Aktivität ihre Abarbeitung stattfinden kann. Ist dies für jede Methodenkontur bekannt, kann bei jedem Methodenaufruf eines Objekts festgehalten werden, von welcher Aktivität dieser Aufruf ausging. Eine Klasse wird dann in unterschiedlichen Aktivitäten verwendet, wenn ihre Methoden aus unterschiedlichen Aktivitäten heraus aufgerufen werden. Liegt ein solcher Fall vor, gibt es entweder die Möglichkeit, daß dasselbe Laufzeitobjekt von mehreren Aktivitäten verwendet wird, oder daß unterschiedliche Instanzen derselben Klasse in verschiedenen Aktivitäten verwendet werden. Im ersten Fall ist die Teilung der Klassenkontur weder möglich noch sinnvoll, hier muß gerade eine günstige Plazierung dieses einen Objekts „zwischen“ den es verwendenden Aktivitäten berechnet werden. Im zweiten Fall können die Instanzen der Klasse entsprechend ihrer Verwendung in den unterschiedlichen Aktivitäten partitioniert werden. Mit der Teilung ihrer Klassenkontur berechnet die Verteilungsanalyse für jede Kontur eine Plazierung, welche die verschiedenartige Verwendung der Instanzen dieser Kontur berücksichtigt.

6.1.1 Benutzung

Jede Methodenkontur *f* erhält einen zusätzlichen Datenflußknoten act_f, der die Aktivitäten speichert, in denen *f* ausgeführt wird. Zusätzlich erhält jede Klassenkontur *A* einen Datenflußknoten *A.act*, der sich wie eine zusätzliche Instanzvariable verhält und in dem alle Aktivitäten gespeichert werden, die Methoden dieser Klassenkontur aufrufen. Abbildung 46 veranschaulicht die Vorgehensweise anhand von zwei Methodenaufrufen *x.g(…)* und *y.g(…)* aus zwei Methodenkonturen *f* und *h* heraus. Die Methodenkontur *f* soll dabei innerhalb der Aktivität a_1 und die Methodenkontur *h* innerhalb von a_2 ausgeführt werden. Die Aktivitätsknoten act_f und act_h enthalten folglich die Aktivitäten a_1 bzw. a_2. In *x* und *y* befindet sich dieselbe Klassenkontur *A*. Für den Aufruf *x.g(…)* in *f* und *y.g(…)* in *h* wurde dieselbe Methodenkontur *g* selektiert. Der Aufruf *x.g(…)* verursacht jetzt zusätzlich zur Parameterübergabe und Rückgabe des Resultats folgende Datenflußkanten: Da die Aktivität beim Aufruf von *g* nicht wechselt, wird *g* von allen Aktivitäten aufgeführt, von denen auch die aufrufende

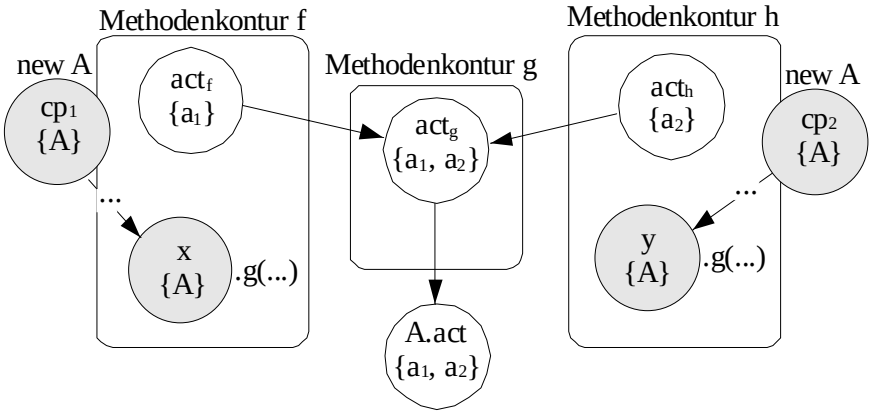

Abbildung 46
Datenfluß in den Aktivitätsknoten

Methodenkontur *f* ausgeführt wird. Dies symbolisiert die Kante $(act_f \rightarrow act_g)$, die alle *f* ausführenden Aktivitäten in den Aktivitätsknoten von *g* transportiert. Da *g* mit der Variable *x* aufgerufen wird, werden alle Klassenkonturen in *x* von den Aktivitäten act_g benutzt. Die zugehörige Datenflußkante ist $(act_g, A.act)$, da sich in *x* die Klassenkontur *A* befindet. Betrachtet man *A.act* wie eine zusätzliche Instanzvariable, so kann man sich diese Kante wie das Resultat einer zusätzlichen Zuweisung $x.act = act_g$ vorstellen.

Der Aufruf aus der Aktivität a_2 führt zu entsprechenden Kanten, die bewirken, daß sowohl der Datenflußknoten act_g als auch *A.act* beide die Aktivitäten a_1 und a_2 enthalten. Sowohl *g* als auch *A* werden also von den Aktivitäten a_1 und a_2 verwendet. Klassenkonturen von Threads und die Aktivitäten des Programms werden dabei miteinander identifiziert. Bei der Erzeugung eines Threads wird eine Datenflußkante von der Variablenkontur der new-Anweisung zu dem Aktivitätsknoten act_{run} der run-Methode des Threads eingeführt. Die Erzeugung eines Threads ist in Abbildung 47 dargestellt. Mit der run-Methode der Threadklasse beginnt dieser seine Ausführung. Sie ist die Wurzel der neuen Aktivität; von ihr ausgehend fließt die Threadkontur durch den Teil des Datenflußgraphen, der aus den Aktivitätsknoten besteht. Die Kanten von den Variablenkonturen der new-Anweisung zu den Aktivitätsknoten der entsprechenden run-Methoden sind die

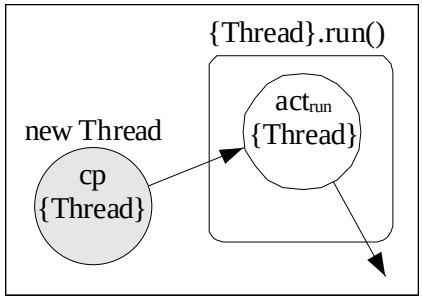

Abbildung 47
Ursprung der Aktivitäten

einzigen, welche den Teil des Datenflußgraphen, dessen Knoten die Variablen des Programms darstellen, mit den zusätzlichen Aktivitätsknoten verbinden, die die Benutzung von Objekten und Methoden verwalten.

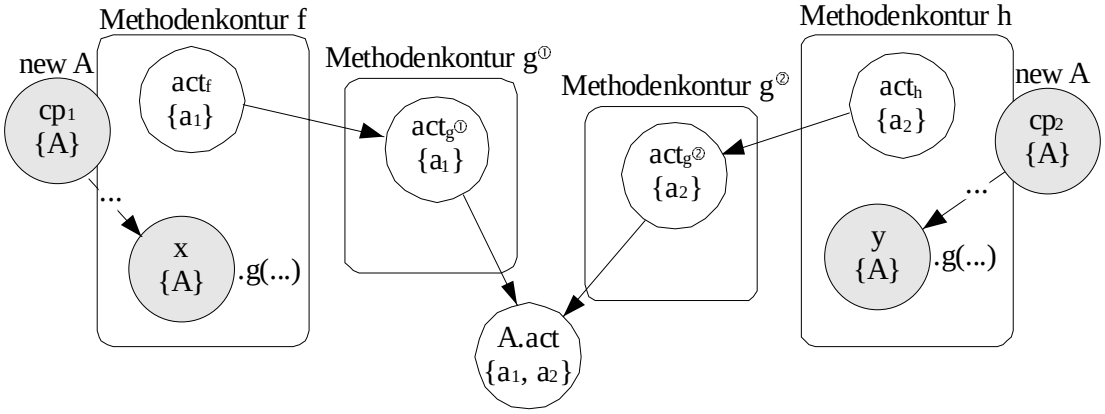

Abbildung 48
Methodenkontur-Splitting nach Aktivitäten

Um die in Abbildung 46 auftretende Unschärfe in act_g und *A.act* aufzulösen, wird mit denselben Vorgehensweisen aus 2.10 gearbeitet. Die Methodenkontur *g* kann, wenn sie von zwei verschiedenen Stellen im Programm aus aufgerufen wird, in zwei Konturen $g^{①}$ und $g^{②}$ geteilt werden. Der Aufruf aus *f* erreicht dann $g^{①}$. Nach $g^{①}$ wird dadurch nur noch die Aktivität a_1 transportiert. Der Aufruf aus *h* erreicht $g^{②}$ und transportiert a_2 in den Aktivitätsknoten $act_{g^{②}}$ von $g^{②}$. Diese Situation ist in Abbildung 48 dargestellt. Die Vermischung der Datenflußwerte geschieht jetzt erst bei der Speicherung der Aktivitäten aus $act_{g^{①}}$ und $act_{g^{②}}$ in die pseudo-Instanzvariable *A.act*. Hier greift die Regel aus 2.10.2; Beim Auflösen einer Unschärfe in einer Instanzvariablen identifiziert man alle Anweisungen, die für inkompatible Datenflüsse in die Instanzvariable verantwortlich sind. In diesem Fall sind das gerade die beiden Methodenaufrufe von $g^{①}$ und $g^{②}$ in *f* bzw. *h*. Die Zuweisung der beiden Aktivitäten geschieht dabei an die Instanzvariable *x.act* bzw. *y.act*. Die Zurückverfolgung im Datenflußgraphen von *x* und *y* zu ihren Erzeugungspunkten liefert die beiden Variablenkonturen cp_1 und cp_2. Kreuzen sich die Pfade dorthin nicht, kann die erzeugte Klassenkontur *A* in A^1 und A^2 wie

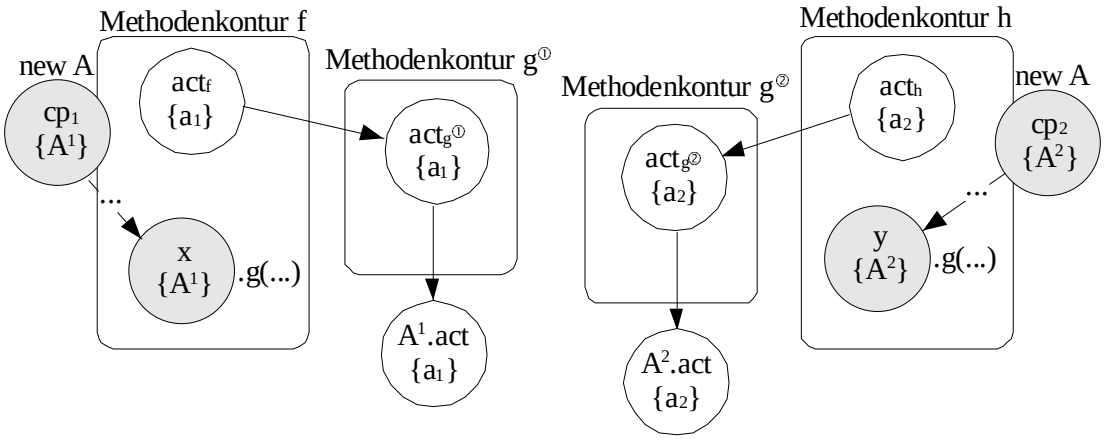

Abbildung 49
Klassenkontur-Splitting nach Aktivitäten

in Abbildung 49 dargestellt geteilt werden.

Durch die Teilung von *A* findet der Methodenaufruf in *f* und *h* auf zwei unterschiedlichen Klassenkonturen A^1 und A^2 statt. Jede der Aktivitäten a_1 und a_2 benutzt nur noch eine der beiden Klassenkonturen A^1 bzw. A^2. Für beide kann getrennt eine Plazierung berechnet werden, die auf die Verwendung in unterschiedlichen Aktivitäten Rücksicht nehmen kann.

6.2 Partielle Auflösung von Unschärfen

Die Regeln zum Auflösen von Unschärfen aus 2.10 gehen immer davon aus, daß eine Konfluenz vorliegt, d.h. daß die in die Unschärfe hineinfließenden Datenflußwerte selbst alle scharf sind. Für eine Unschärfe *(vc, p)* heißt das, daß in den Variablenkonturen aus $back_p(vc)$ selbst keine Unschärfe vorliegt. Diese Bedingung ist sehr oft nicht erfüllt. Werden z.B. die Instanzvariablen einer Klasse an einer Stelle des Programms echt polymorph verwendet, d.h. in einer Weise, welche die Analyse nicht auflösen kann (vgl. 2.11), ist es nicht mehr möglich, monomorphe Verwendungen derselben Klasse an anderer Stelle des Programms von dieser polymorphen Verwendung zu trennen.

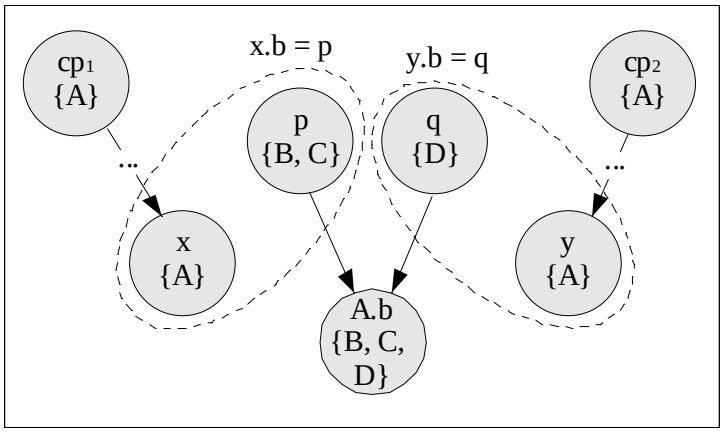

Abbildung 50
Unschärfe in $back_p(A.b)$

Abbildung 50 gibt ein Beispiel für eine solche Unschärfe $(A.b, p_{class})$ in einer Instanzvariablen. In sie fließen Werte aus *p* und *q*. Die Werte *value(p) = {B, C}* sind dabei selbst unscharf und wir nehmen an, daß sich diese Unschärfe nicht auflösen läßt. Die Unschärfe in *A.b* läßt sich zwar nicht ganz auflösen, aber dennoch reduzieren. Die Klassenkontur *A*, zu der die unscharfe Instanzvariable *A.b* gehört, läßt sich genau dann in ihren Erzeugungspunkten cp_1 und cp_2 in zwei Konturen A^1 und A^2 teilen, wenn sich die Pfade von *x* nach cp_1 und von *y* nach cp_2 nicht kreuzen (vgl. 2.10.2). Die

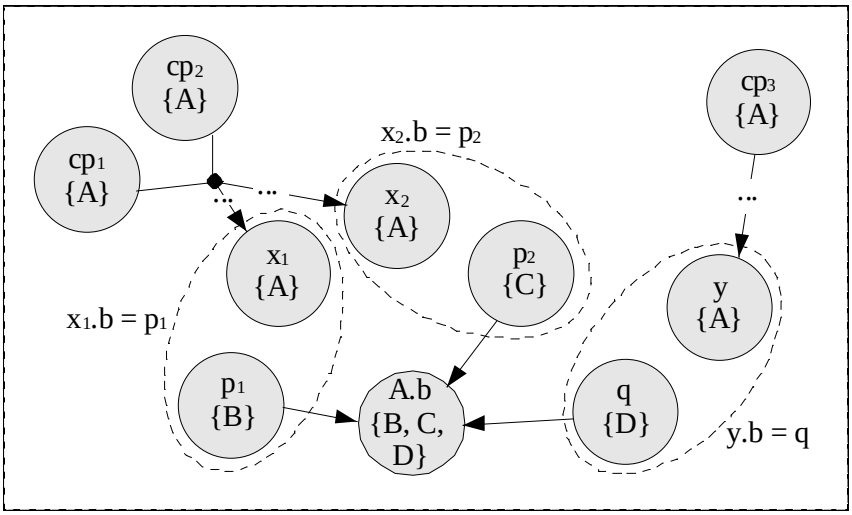

Abbildung 51
nicht auflösbare Pfadunschärfe

unscharfe Instanzvariable *A.b* zerfällt dann in zwei Konturen, $A^1.b$ und $A^2.b$, von der die eine $A^2.b$ scharf (sie enthält nur den Wert *D*), die andere A^1b immer noch unscharf ist. Die unscharfen Werte in $A^1.b$ werden nach der Teilung von *A* allerdings weniger. $A^1.b$ enthält dann nur noch die Werte $value(A^1.b) = \{B, C\}$.

Eine ganz ähnliche Situation ist in Abbildung 51 dargestellt. Die betrachtete Unschärfe ist wieder $(A.b, p_{class})$. Hier tritt die nicht auflösbare Unschärfe zwar nicht direkt in einer Variablenkontur aus $back_p(A.b) = \{p_1, p_2, q\}$ auf, es fließen jedoch aus p_1 und p_2 inkompatible Werte *B* und *C* nach *A.b* und die Pfade von x_1 und x_2 zu den Erzeugungspunkten cp_1 und cp_2 des Behälters kreuzen sich. Diese Pfadunschärfe möge hier nicht auflösbar sein. Wenn sich jedoch diese beiden Pfade nicht mit dem Pfad von *q* nach cp_3 kreuzen, so läßt sich die Klassenkontur *A* dennoch in zwei verschiedene Konturen A^1 und A^2 teilen. Nach der Teilung entsteht an den Erzeugungspunkten cp_1 und cp_2 dieselbe Klassenkontur A^1. Ließe man dort unterschiedliche Klassenkonturen entstehen, würden sich diese auf den sich kreuzenden Pfaden nach x_1 und x_2 wieder vermischen. Die unscharfe Instanzvariablenkontur *A.b* läßt sich also nur in eine weiterhin unscharfe Kontur $A^1.b$ und eine scharfe Kontur $A^2.b$ teilen. Dieses Abspalten der monomorphen Verwendung von $A^2.b$ mit dem Typ *D* ist in Abbildung 52 dargestellt. Durch diese Aufweichung der Voraussetzungen für die Teilung einer Kontur kann eine nicht auflösbare Unschärfe die Separierung anderer monomorpher Verwendungen nicht mehr verhindern.

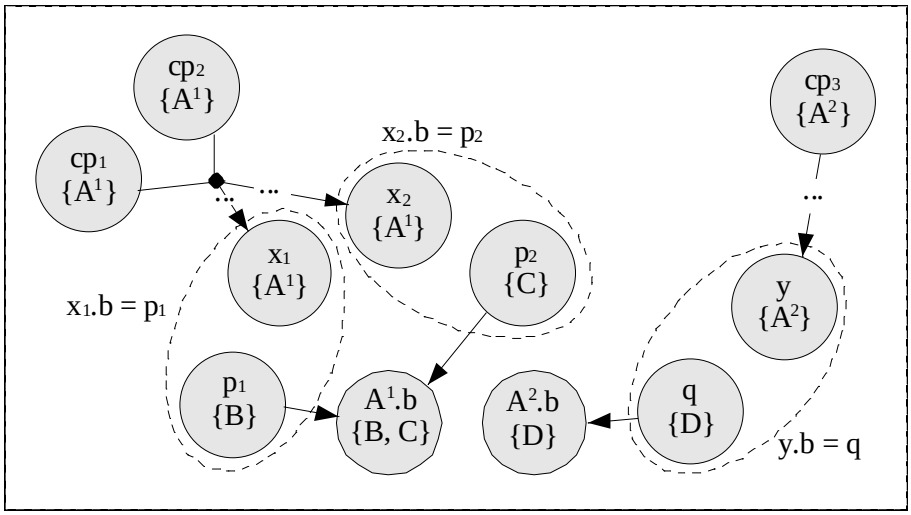

Abbildung 52
Teilweise aufgelöste Unschärfe

7 Architektur und Implementierung

Die in dieser Arbeit vorgestellte statische Lokalitätsoptimierung ist in *jpc*, den JavaParty-Übersetzer, integriert [Phi97b]. Dieser wiederum basiert auf *Espresso Grinder*, einem der ersten verfügbaren Java-Übersetzer [Ode96]. Die grobe Struktur von *Espresso Grinder* und damit auch von *jpc* ist dem Beitrag von Matthias Zenger in [Phi96] zu entnehmen.

Die Erweiterungen für die Lokalitätsoptimierung sind in nachfolgender Aufstellung grau schattiert in den Programmablauf des Übersetzers eingefügt:

- Zerteilung der Eingabe und Aufbau des abstrakten Syntaxbaums
- Semantische Analyse (Attributierung des Baums in mehreren Durchläufen)
- Analysephase der Lokalitätsoptimierung
- Transformation des abstrakten Syntaxbaums als Ergebnis der Lokalitätsoptimierung
- erneute Semantische Analyse
- Transformation zur Implementation von Fernklassen
- erneute Semantische Analyse
- Codegenerierung

Abbildung 53 gibt einen Überblick über die Strukturen, mit denen die Analyse in den JavaParty-Übersetzer integriert wurde. Der grau schattierte Bereich markiert dabei den von *jpc* stammenden Teil, auf den die Erweiterung aufbaut. Der JavaParty-Übersetzer führt alle Transformationen direkt auf dem attributierten Syntaxbaum durch. Die Knoten des Syntaxbaums sind Instanzen von Klassen, die von der Klasse *AST* abgeleitet sind. Der abstrakte Syntaxbaum wird während der semantischen Analyse zum Zweck der Typüberprüfung mit den Identitäten des Programms (Klassen, Funktionen und Variablen) attributiert. Diese sind durch Instanzen der Klassen *ClassObj*, *FunObj* und *VarObj* identifiziert.

Für die in Kapitel 2 beschriebene Typinferenz ist der abstrakte Syntaxbaum keine gute Repräsentation des Programms, da dafür eine kleine Teilmenge der Sprache (das abstrakte Java aus 2.5) ausreicht. Die Selektion und das Teilen von Klassen- und Funktionskonturen läßt sich auf der Baumstruktur ebenfalls schlecht durchführen. Eine neue Zwischenrepräsentation des Programms kommt deswegen nicht in Frage, weil davon sonst die anschließende Transformation von Fernklassen und die Codeerzeugung betroffen wäre. In einen rekursiven Baumdurchlauf wird daher aus dem Syntaxbaum jeder Methode eine Sequenz von abstrakten Java-Anweisungen generiert und bei der diese Methode repräsentierenden Instanz der Klasse *FunObj* gespeichert. Die abstrakten Anweisungen einer Funktion operieren weiterhin auf den Variablen des Programms, die durch Instanzen der Klasse *VarObj* identifiziert sind. Die extrahierten Anweisungen des abstrakten Java dienen dabei lediglich der Analyse. Vor den weiteren Phasen des Übersetzers wird der Syntaxbaum des Programms entsprechend den Ergebnissen der Analyse transformiert und neu attributiert.

Für die Typinferenz sind die Knoten des Kontrollflußgraphen allerdings nicht die Methoden des Programms sondern deren Konturen. Die Kontur einer Methode ist durch eine Instanz der Klasse *FunContour* repräsentiert. Jede Kontur einer Funktion besitzt die gleichen Anweisungen, mit dem Unterschied, daß Anweisungen einer Funktionskontur nicht auf Variablen sonder auf deren Konturen operieren. Von einer Variable (repräsentiert durch eine Instanz der Klasse *VarObj*) kann über den Besitzer die entsprechende Variablenkontur selektiert werden. Der Besitzer einer Kontur einer lokalen Variable ist die Kontur der sie enthaltenden Funktion. Entsprechend ist der Besitzer einer Instanzvariablenkontur die Kontur derjenigen Klasse, in welcher die Instanzvariable deklariert ist. Von den apply-Anweisungen einer Funktionskontur gehen die Kanten der interprozeduralen Kontrollflußgraphen aus. Diese Kanten zeigen auf andere Funktionskonturen. Bei jedem Anlegen einer

neuen Funktionskontur während der Analyse findet ein Spezialisierungsschritt statt, der die abstrakten Anweisungen der Funktion in die entsprechenden Anweisungen der Funktionskontur übersetzt. Dabei werden von den lokalen Variablen der Funktion entsprechend neue Variablenkonturen selektiert.

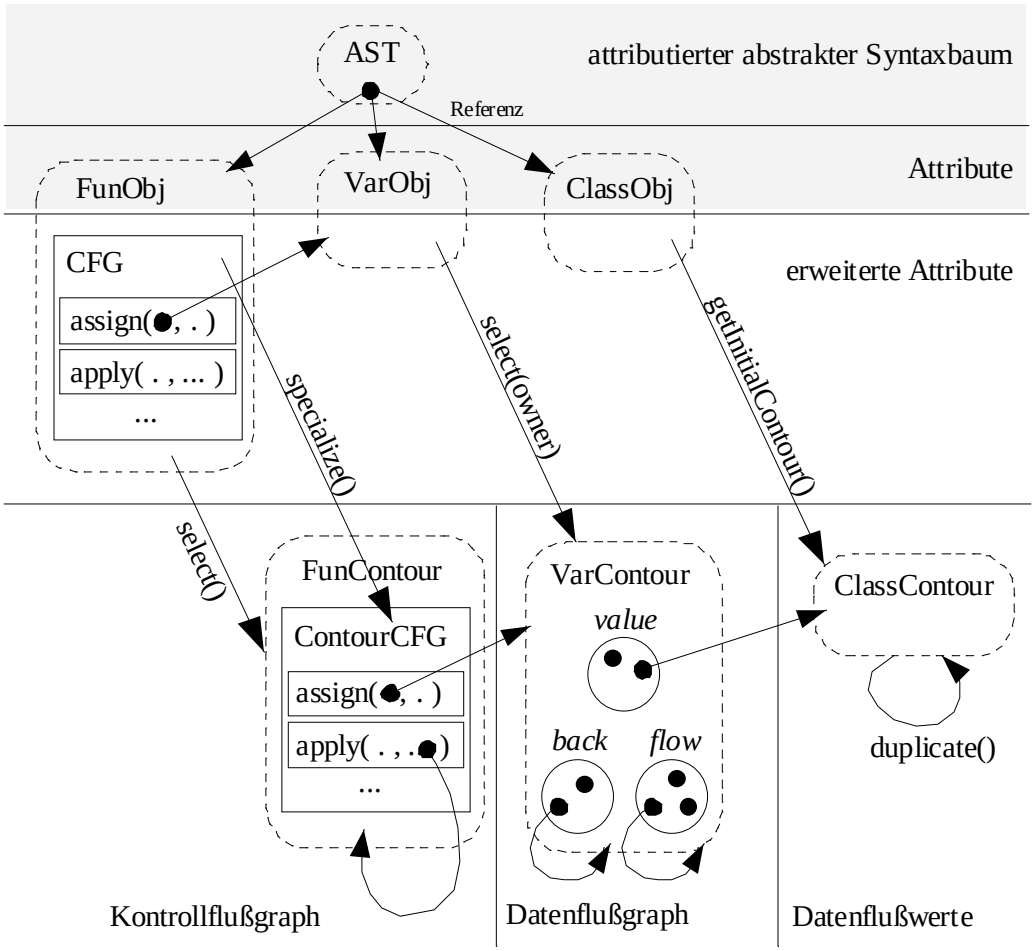

Abbildung 53
Überblick über beteiligte Strukturen

Die Variablenkonturen bilden die Knoten des Datenflußgraphen des Programms. Eine Kante im Datenflußgraphen ist durch zwei Verweise repräsentiert. Die Variablenkontur, von der die Kante ausgeht, enthält in der Menge *flow* eine Verweis auf die Zielkontur. Die Variablenkontur, in der die Kante mündet, enthält in der Menge *back* einen Verweis auf die Ausgangskontur. Auf diese Weise läßt sich der Datenflußgraph vorwärts und rückwärts traversieren.

Klassenkonturen sind durch Instanzen der Klasse *ClassContour* repräsentiert. Sie bilden die konkreten Typen, die als Datenflußwerte entlang der Kanten des Datenflußgraphen durch die Variablenkonturen fließen. Bei der Selektion von Instanzvariablenkonturen dienen die Klassenkonturen als Besitzer. Für jede Klasse existiert eine initiale Klassenkontur, die bei einem Teilungsschritt dupliziert werden kann.

Während der Typinferenz finden alle Transformationen auf der Kontur-Ebene statt. Das weiterhin als Syntaxbaum repräsentierte Ausgangsprogramm bleibt unverändert. An Typinferenz und Verteilungsanalyse, die ebenfalls auch auf der Kontur-Ebene stattfindet, schließt sich eine Transformation des Syntaxbaums des Programms an. Diese Transformation generiert aus dem Syntaxbaum des Ausgangsprogramms und den daran annotierten Konturinformationen einen neuen Syntaxbaum.

8 Ergebnisse

Wie an einigen für JavaParty verfügbaren Benchmarks nachgewiesen werden kann, ist das Auflösungsvermögen der in dieser Arbeit verwendeten Typinferenz nebst ihren Erweiterungen oft nicht ausreichend, um eine gute Objektverteilung für ein gegebenes Programm berechnen und implementieren zu können. In diesem Kapitel soll diskutiert werden, unter welchen Voraussetzungen eine gute Objektverteilung zu erwarten ist und welche Ursachen zu Grunde liegen, wenn die berechnete Verteilungsstrategie die Erwartungen nicht erfüllt.

8.1 Wann die Informationen nicht ausreichen

Es gibt mehrere Ursachen, warum die Analyse durch die Typinferenz nicht genug Information über ein Programm gewinnen kann, um eine gute Objektverteilung zu berechnen. In den folgenden Abschnitten sollen Ursachen angesprochen werden, die für das Scheitern einer brauchbaren Verteilungsstrategie verantwortlich gemacht werden konnten.

8.1.1 Problemstruktur

Ist das durch das Programm beschriebene Problem so strukturiert, daß mehrere gleichartige Berechnungen parallel durchgeführt werden, resultiert das in der Erzeugung mehrerer für die Analyse identischer Aktivitäten. Sie werden alle durch eine einzige Klassenkontur einer Threadklassse beschrieben. Die fehlende Unterscheidung wirkt sich dann so aus, daß alle Aktivitäten auf derselben virtuellen Maschine angelegt werden. Die mögliche Parallelität kann dann nicht ausgenutzt werden, da sich die Aktivitäten die Rechenzeit einer einzigen virtuellen Maschine teilen müssen, anstatt parallel auf unterschiedlichen virtuellen Maschinen ausgeführt zu werden.

Erschwerend kommt hinzu, daß gleichartige Aktivitäten meist in einer Schleife erzeugt werden. Die sie repräsentierenden Threadobjekte werden dann in einen Feld gespeichert. Die Erzeugung in einer Schleife hat zur Folge, daß alle Threadobjekte an derselben new-Anweisung entstehen. Die Analyse teilt aber Klassenkonturen nur nach unterschiedlichen Erzeugungspunkten. Auch die Teilung der betreffenden Methodenkontur kann keine Abhilfe schaffen, da die Erzeugung der Aktivitäten nicht in verschiedenen Aufrufen derselben Methode, sondern innerhalb eines einzigen Aufrufs der Methode geschieht. Die Analyse differenziert aber den Kontrollfluß innerhalb einer Methode nicht weiter, womit alle in einer Schleife erzeugten Objekte nicht unterscheidbar sind und immer derselben Klassenkontur angehören. Die Speicherung der Aktivitäten in ein Feld würde aber ohnehin eine getrennte Analyse verhindern, da Zugriffe auf verschiedene Indizes eines Feldes nicht unterschieden werden. In Kapitel 9 werden Vorschläge für Erweiterungen der Analyse gemacht, um besser mit solchen gleichartigen Aktivitäten umgehen zu können.

8.1.2 Entwurfsmuster „Arbeitsvorrat“

Die Verwendung des Entwurfsmusters „Arbeitsvorrat“ in einem JavaParty Programm ist ein weiterer Grund, warum die Analyse statisch keine ausreichenden Information über den Programmablauf gewinnen kann. Dabei werden zuerst eine Reihe von Aktivitäten erzeugt, die auf zu erledigende Arbeit warten. Die anfallenden Berechnungen werden in einen Arbeitsvorrat gespeichert, woraus sich die zuvor angelegten Aktivitäten bedienen. Hat eine Aktivität eine Berechnung beendet, greift sie sich ein neues Arbeitsobjekt aus dem Vorrat zur Abarbeitung heraus. Die hier vorliegende dynamische Zuordnung von Aktivitäten zu den durchzuführenden Berechnungen kann durch eine statische Programmanalyse nicht aufgelöst werden. Für die Analyse bleiben die Beziehungen zwischen den Aktivitäten und den durchzuführenden Berechnungen verborgen, was sich darin manifestiert, daß alle Arbeitsthreads derselben Klassenkontur angehören. Werden von ihrem konkreten Typ her unterschiedliche Berechnungen in den Arbeitsvorrat abgelegt, resultiert das in einer nicht auflösbaren Unschärfe im Kontrollfluß der Arbeitsthreads.

Unter Verwendung dieses Entwurfsmusters ist eine Verteilungsanalyse allerdings sowieso wenig sinnvoll, da eine gute Objektverteilung durch wenige Anweisungen an das JavaParty-Laufzeitsystem auch von Hand implementiert werden kann. Man legt dabei auf jeder zur Berechnung zur Verfügung stehenden virtuellen Maschine einen Arbeitsthread an und implementiert die Objekte, welche die Berechnungen repräsentieren, am besten durch lokale Klassen. Bei der Übergabe eines Arbeitsobjekts wird dieses dann automatisch in den Adreßraum kopiert, in dem sich der Thread befindet, welcher die Berechnung durchführen soll (vgl. 8.2.1).

8.2 Wann die Analyse nicht nötig ist

8.2.1 Gruppierung im Entwurf

Durch geschickter Ausnutzung der Unterschiede zwischen lokalen Klassen und Fernklassen läßt sich eine automatische Gruppierung einer Aktivität mit den sie betreffenden Objekten erreichen. Zur Beschreibung einer Aktivität leitet man dabei die Fernklasse *jp.lang.RemoteThread* ab. Alle diese Aktivität betreffenden Objekte implementiert man als lokale Klassen. Nach der Erzeugung der Aktivität wird jedes Objekt, das an die Aktivität übergeben wird, automatisch in den Adreßraum dieser Aktivität kopiert, da es sich um ein Objekt einer lokalen Klasse handelt. Kommunikation zwischen den Aktivitäten wird dann ausschließlich über das Aktivitätsobjekt selbst abgewickelt, da es sich bei diesem um ein entferntes Objekt handelt und nur solche außerhalb des eigenen Adreßraums bekannt sind. Bei dieser Entwurfsmethode gibt es nur sehr wenige entfernte Objekte, die mit einer Vielzahl lokaler Objekte zusammenarbeiten. Da entfernte Objekte immer auch Aktivitäten repräsentieren, ergibt selbst eine willkürliche Objektverteilung eine brauchbare Ausnutzung der Parallelität und durch die inhärente Gruppierung eine gute Lokalität. Die Verteilungsanalyse kann daran nichts Entscheidendes verbessern und führt durch den Zusatzaufwand, der bei der Implementation der Methodenkonturen entsteht, zu einer längeren Laufzeit.

8.3 Wann eine gute Verteilung zu erwarten ist

Damit die Analyse überhaupt eine Chance hat, eine sinnvolle Verteilungsstrategie zu berechnen, ist primär wichtig, daß im Programm unterschiedliche Aktivitäten identifiziert werden können. Wie in 3.1 beschrieben, identifiziert die Analyse Aktivitäten über verschiedene Konturen einer Threadklasse. Eine Teilung einer Klassenkontur tritt nur dann auf, wenn es unterschiedliche monomorphe Verwendungen von Objekten dieser Klasse gibt, die in ihrer Summe zu polymorphen konkreten Typen von Instanzvariablen der Klasse führen. Angewendet auf Aktivitäten heißt das, daß nur solche Aktivitäten separat untersucht werden können, von denen unterschiedliche Konturen existieren, die sich also in ihrer Struktur unterscheiden. Sind die Aktivitäten des Programms identifiziert, trennt die Analyse auch Klassen in mehrere Konturen, deren Instanzen in unterschiedlichen Aktivitäten benutzt werden. Diese in 6.1 vorgestellte Erweiterung der Typinferenz erlaubt, daß Objekte derselben Klasse und des gleichen monomorphen konkreten Typs jeweils auf der virtuellen Maschine erzeugt werden, auf der die sie benutzende Aktivität angelegt wurde.

8.3.1 Entwurfsmuster „Fabrik“

Bei der Verwendung des Entwurfsmusters „Fabrik“ werden Objekte nicht direkt über eine new-Anweisung erzeugt, sondern in einer Methode eines sog. Fabrik-Objekts. Ihre hauptsächliche Nutzung findet aber nach Rückgabe des erzeugten Objekts als Ergebnis außerhalb der Fabrik-Methode statt. Diese Fabrik-Methode kann in unterschiedlichen Kontexten verwendet werden; ohne Typinferenz wird darauf aber keine Rücksicht genommen. Der Bezug des in der Fabrik-Methode erzeugten Objekts zu seiner Verwendung kann erst durch die Typinferenz modelliert werden. Müssen Aufrufe der Fabrik-Methode getrennt analysiert werden, weil sie aus verschiedenen Aktivitäten heraus stattfinden, oder Objekte zur Verwendung in unterschiedlichen Aktivitäten erzeugen, kann erst die Typinferenz die Methodenkontur der Fabrik-Methode und mit ihr auch die Stelle der Objek-

terzeugung teilen. In den entstehenden verschiedenen Konturen kann dann die Verwendung des erzeugten Objekts in seiner Umgebung untersucht werden, was unter Umständen zur Erzeugung des Objekts auf verschiedenen virtuellen Maschinen in Abhängigkeit der Kontur der Fabrik-Methode führt.

8.4 Empirische Untersuchungen

8.4.1 Ein positives Beispiel

Das in diesem Abschnitt vorgestellte Testprogramm *Speed2* enthält keine der besprochenen für die Analyse kritischen Strukturen. Die durch die Typinferenz identifizierten statisch unterscheidbaren Aktivitäten sind auch diejenigen, welche sinnvollerweise zur Verteilung herangezogen werden sollten. Die Erweiterung der Typinferenz kann darauf aufbauend Konturen von mehreren Aktivitäten gemeinsam benutzter Klassen teilen. Dies ermöglicht der Analyse zur Generierung der Verteilungsstrategie, ausreichend Informationen aus dem Kontrollflußgraphen der Aktivitäten zu extrahieren, um für jede am Programm beteiligte Klassenkontur geeignete Plazierungsinformation zu berechnen.

Das Testprogramm *Speed2* demonstriert die Notwendigkeit einer Strategie zur Objektplazierung in einem verteilten Adreßraum und weist nach, daß eine statische Programmanalyse mit Hilfe von Typinferenz dies unter gewissen Voraussetzungen leisten kann. Das Programm besteht ohne verwendete Bibliotheken aus 7 Klassen mit insgesamt 15 Methoden. Zur Laufzeit werden mehrere nebenläufige Aktivitäten erzeugt, die ihrerseits wieder eine Vielzahl von Objekten instanzieren und Methoden dieser Objekte aufrufen.

```
Speed2
    7 Klassen
       6 Fernklassen
       1 lokale Klasse
    15 Methoden

Datenflußanalyse
    9 Unschärfen
       4 Kontrollflußunschärfen
       5 Aktivitätsunschärfen

Typinferenz
    26 Teilungsoperationen
       6 Klassenkontur-Splitting
       20 Methodenkontur-Splitting
       17 Klassenkonturen
       41 Methodenkonturen

Laufzeit
    3 virtuelle Maschinen
       257.927ms ohne Objektverteilung
       117.476ms mit Objektverteilung
```

Abbildung 54
Daten von Speed2

Die Datenflußanalyse identifiziert 4 Kontrollfluß- und 5 Aktivitätsunschärfen, bei denen unterschiedliche Aktivitäten dieselbe Kontur einer Klasse benutzen. Diese Unschärfen veranlassen die Typinferenz zu insgesamt 26 Teilungsoperationen. Dabei können zwar nicht alle Unschärfen aufgelöst werden, die erreichte Genauigkeit reicht aber aus, um eine gute Objektverteilung berechnen und implementieren zu können. Die Laufzeit von ca. 250 Sekunden bei zufälliger Objektverteilung halbiert sich mit der durch die statische Analyse des Programms berechneten Verteilungsstrategie ungefähr. Die den Messungen zugrunde liegende Plattform bestand dabei aus 3 virtuellen Maschinen, die auf 3 über Ethernet gekoppelten Arbeitsstationen ausgeführt wurden. Abbildung 54 faßt die Daten von *Speed2* zusammen.

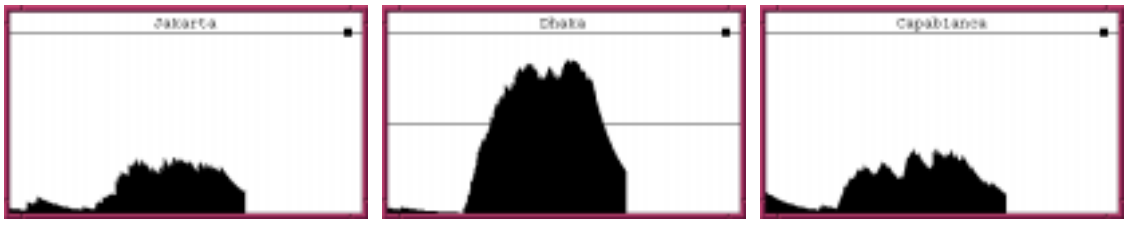

Abbildung 55
xload Bildschirmabzüge der Ausführung von Speed2 mit zufälliger Objektverteilung

Abbildung 55 stellt Bildschirmabzüge von Lastmessungen mittels *xload* auf den Arbeitsstationen gegenüber, die an der Ausführung von *Speed2* mit zufälliger Objektverteilung beteiligt waren. Man

erkennt, daß die Last sehr ungleich auf die drei Maschinen verteilt ist. Da die Objekte zufällig verteilt werden, kann das aber nicht daran liegen, daß auf der Maschine *Dhaka* überdurchschnittlich viele Objekte angelegt wurden. Daß *Dhaka* mit fast doppelter Überlast läuft (1 Querstrich bedeutet Last 1, volle Last) kann nur so erklärt werden, daß die auf ihr angelegten Objekte in unterschiedlichen Aktivitäten verwendet werden. Versuchen mehrere Aktivitäten gleichzeitig Methoden von Objekten auszuführen, die alle auf *Dhaka* liegen, müssen alle diese Aufrufe durch das Betriebsystem für einen Prozessor serialisiert werden, was sich in der dargestellten Überlast ausdrückt. Die zufällig gewählte Objektverteilung läßt also nur eine sehr schlechte Ausnutzung der durch das Programm vorgegebenen Parallelität zu.

Die Typinferenz kann in *Speed2* alle für die Verteilungsstrategie relevanten Unschärfen auflösen und die Aktivitäten des Programms identifizieren. Das Laufzeitverhalten von *Speed2* mit der generierten Objektverteilung ist in Abbildung 56 wieder anhand von Bildschirmabzügen von *xload* dargestellt. Man erkennt, daß bei Programmstart auf allen drei Maschinen die Last gleichmäßig stark ansteigt. Das Fehlen der Querstriche im *xload*-Bild auf *Capablanca*, die die Überlast andeuten, liegt daran, daß auf *Capablanca* wegen eines nicht installierten Solaris-Patches keine *native* Threads verwendet werden konnten. Alle Java-Threads müssen sich auf *Capablanca* daher einen Betriebsystem-Thread teilen, weswegen die Last nicht über 1 anwachsen kann. Man sieht daß die berechnete Objektverteilung nicht nur zu einer guten Lastverteilung auf den drei beteiligten Maschinen führt, sondern auch die im Programm enthaltene Parallelität gut ausgenutzt wird. Anders wäre die halbierte Ausführungszeit gegenüber der zufälligen Objektverteilung nicht zu erklären.

Abbildung 56
Laufzeitverhalten mit generierter Objektverteilung

8.4.2 Negative Beispiele

Für JavaParty stehen bereits mehrere Benchmarks zur Verfügung. Es handelt sich dabei um Implementierungen der unter dem Namen Salishan-Probleme [Feo92] bekannten Algorithmen *Hamming Problem*, *Doctors Office*, *Paraffins Problem*. In [Nes97] wird außerdem eine Parallelisierung des n-Körperproblems nach dem Algorithmus von Barnes und Hut für JavaParty vorgestellt.

Die Anwendung der Analyse auf oben genannte Programme führte jedoch nicht zu einer günstigen Objektverteilung. Dafür konnten folgende Ursachen verantwortlich gemacht werden.

Damit Aktivitäten überhaupt auf verschiedenen virtuellen Maschinen angelegt werden können, muß die statische Analyse diese Aktivitäten zuerst unterscheiden. Eine solche Unterscheidung findet statt, indem die Typinferenz die Klassenkontur einer Threadklasse (*jp.lang.RemoteThread* oder eine davon abgeleitete Klasse) in mehrere Konturen teilt. Wie in 2.10 gesehen, wird eine Klassenkontur nur dann geteilt, wenn dadurch eine Klassen-, Kontur- oder Pfadunschärfe aufgelöst werden kann, die irgendwo im Programm direkt oder indirekt eine Unsicherheit im Kontrollfluß verursacht. Für Threadkonturen heißt das, daß sie von der Typinferenz nur dann geteilt werden, wenn statisch ein Unterschied im Kontrollfluß von Instanzen dieser Kontur nachgewiesen werden kann.

Wird die Kontur einer Threadklasse geteilt, sind die durch die Konturen repräsentierten Aktivitäten statisch unterscheidbar. Nur statisch unterscheidbare Aktivitäten stehen für die anschließende Verteilungsanalyse auch zur Verfügung. Werden zur Laufzeit mehrere Instanzen einer Threadkontur erzeugt, so werden sie bei diesem Ansatz jedoch immer der selben virtuellen Maschine zugeordnet.

Bei der Untersuchung der oben genannten Programme kann die Teilung von Threadkonturen durch die Typinferenz beobachtet werden. Dadurch werden statisch unterscheidbare Aktivitäten im Programm identifizieren. Die Instanzen einer solchen Threadkontur zeichnen sich dann durch denselben Kontrollflußgraphen aus. In diesen Programmen ist es aber gerade nicht sinnvoll, alle Aktivitäten einer solchen Kontur auf der selben virtuellen Maschine anzulegen. Da zur Laufzeit viele Instanzen einer Threadkontur angelegt werden, und die einzelnen Instanzen weniger mit gleichartigen Instanzen der selben Kontur als mit Instanzen einer anderen Threadkontur kooperieren, macht die Verteilung ganzer Konturen auf virtuelle Maschinen hier keinen Sinn (vgl. 8.1.1).

Die in [Nes97] vorgestellte Parallelisierung des n-Körperproblems macht von dem Entwurfsmuster „Arbeitsvorrat“ Gebrauch, was zu einer völlig dynamischen Zuordnung von Aktivitäten und den von ihnen benutzten Objekten führt. Wo man bei den anderen Beispielen noch hoffen kann, durch genauere Analyse des lokalen Kontrollflusses von Methoden und ein besseres Verständnis von Schleifen und Feldern eine gute Objektverteilung mittels statischer Analyse berechnen zu können, scheint in diesem Fall statische Analyse zur Generierung einer Objektverteilung prinzipiell wenig aussichtsreich.

8.5 Aufwand

[Ple96] gibt die theoretische Komplexität des Typinferenzalgorithmus als exponentiell an. Die dort vorgenommenen empirischen Untersuchungen belegen allerdings, daß in der Praxis die Komplexität nur langsam mit der Programmgröße zunimmt.

In der vorliegender Implementierung führt die prototypisch implementierte Datenflußanalyse, die vor jedem Teilungsschritt eingesetzt wird, um die aktuelle konkrete Typisierung neu zu berechnen, zu extrem langen Laufzeiten. Erschwerend kommt hinzu, daß bei der Übersetzung realer Programme, die ausgiebig Bibliotheken benutzen, alle benutzten Bibliotheken mitanalysiert werden müssen. Außerdem wurde bei der Implementation kein Gebrauch von einer selektiven Neuberechnung nur der Datenflußinformationen gemacht, die durch einen Teilungsschritt ungültig geworden sind.

9 Ausblick

Die statische Analyse zur Berechnung der Verteilungsstrategie läßt sich noch auf mehreren Ebenen verbessern. Die größte Herausforderung stellt dabei sicher die detailliertere Analyse von Feldern und Strukturen des lokalen Kontrollflusses innerhalb einer Methode dar. Beides ist nicht voneinander zu trennen, da das Verständnis des Inhalts eines Feldes das Verständnis seiner Indizierung voraussetzt. Da diese Indizierung aber meist innerhalb einer Schleife stattfindet, ist das Vordringen der Analyse in Strukturen des lokalen Kontroll- und Datenflusses unerläßlich, um Beziehungen zwischen gleichartigen Aktivitäten herauszufinden, die in Schleifen erzeugt und in Felder gespeichert werden.

Ein erster Schritt in Richtung des Verständnisses des lokalen Datenflusses ist die Transformation von zu analysierenden Methoden in *single static assignment* Form (SSA-Form). Diese Transformation, nach der jede Variable innerhalb einer Methode nur an genau einer Stelle zugewiesen wird, ist für das Verständnis von Schleifen unentbehrlich. Sie hat den erwünschten Nebeneffekt, daß Mehrfachverwendungen derselben lokalen Variablen, die in dieser zu nicht auflösbaren Unschärfen führen, durch Vervielfachung der Variable beseitigt werden. In der vorliegenden Implementierung ist diese Transformation nur deswegen nicht enthalten, da der zugrunde liegende Übersetzer Espresso Grinder alle Transformation quasi auf Quellcode-Ebene durchführt und Bytecode direkt aus dem Strukturbaum erzeugt. Da die Transformation in SSA-Form (und die nach der Analyse notwendige Rücktransformation) auf Quellcode-Ebene sehr mühsam ist, wurde auf sie verzichtet, um nicht tiefere Eingriffe in die Interna des Compilers vornehmen zu müssen.

Als Nebenprodukt der Analyse des lokalen Datenflusses in Verbindung mit der Typinferenz, könnte eine verbesserte Konstantenfaltung durchgeführt werden, die ihrerseits wieder ein detaillierteres Bild des lokalen Kontrollflusses und damit auch eine schärfere konkrete Typisierung liefert.

Am anderen Ende des Spektrums steht die Durchführung der Objektverteilung an die zur Verfügung stehenden virtuellen Maschinen. In dieser Arbeit wird eine absolute Zuordnung von Klassenkonturen zu virtuellen Maschinen vorgeschlagen. Das heißt, daß die Analyse für jede Kontur einer Fernklasse die Nummer der virtuellen Maschine bestimmt, auf der Objekte dieser Kontur angelegt werden. Diese Zuordnung ist starr; man ist deswegen darauf angewiesen, daß die Klassen genügend fein in Konturen unterteilt werden können, um eine sinnvolle Verteilung durchführen zu können, da immer alle Objekte einer Kontur auf derselben virtuellen Maschine angelegt werden. Gäbe man diese absolute Objektplazierung zu Gunsten einer relativen auf, würden die Plazierungsentscheidungen mit einem echten Laufzeitobjekt parametrisiert. Die Anweisung „erzeuge Objekt der Klasse *A* auf virtueller Maschine *n*“ würde zu einer Anweisung „erzeuge Objekt der Klasse *A* auf derselben virtuellen Maschine wie das Objekt in der Variablen *x*“. Das Laufzeitobjekt in *x* fungiert dann als Parameter für die Plazierung. Die Analyse müßte nur geeignete Plazierungsparameter für eine Objekterzeugung ausfindig machen. Die Plazierungsparameter bilden dann die Mittelpunkte von Objekt-Clustern; Objekte eines Clusters werden auf derselben Maschine angelegt, entweder weil sie wegen ausgiebiger Kommunikation eng benachbart gespeichert werden müssen oder sie in einer Aktivität verwendet werden und durch Trennung der Objekte aus unterschiedlichen Clustern die Parallelität des Entwurfs erst nutzbar wird. Werden dann voneinander unabhängige Clustermittelpunkte unabhängig von ihrer Klassenkontur reihum auf verschiedenen virtuellen Maschinen angelegt, folgen die zugehörigen Objekte automatisch nach. Es besteht die Hoffnung, daß durch eine solche relative Objektplazierung das Problem von mehreren gleichartigen Aktivitäten adäquater gelöst werden kann. In diesem Fall könnten die Threadobjekte selbst als Clustermittelpunkte dienen. Leider ist der benötigte Clustermittelpunkt nicht an allen Stellen sichtbar, an denen Objekte erzeugt werden, die die Analyse dem Cluster zugeordnet hat. In Schleifen und Feldern bleibt das Problem bestehen, daß das Objekt mit dem richtigen Index, bzw. das Objekt aus der richtigen Schleifeniteration als Clustermittelpunkt gewählt wird. Auch hierzu ist eine detaillierte Analyse des lokalen Kontrollflusses nötig.

Verwendet man als Zielplattform ein heterogenes Cluster aus Arbeitsstationen, in dem auch symmetrische Multiprozessormaschinen enthalten sind, kann dies bei der Berechnung der Verteilungsstrategie berücksichtigt werden. In diesem Fall ist es sinnvoll die Tatsache ausnutzen, daß auf solchen Rechnern mehrere Threads echt parallel ablaufen können. Dort könnten dann mehrere Aktivitäten angelegt werden, die viele gemeinsame Objekte benutzen, um von dem gemeinsamen Adreßraum zu profitieren.

Auch die Transformation zur Implementation der Methodenkonturen bietet noch Optimierungspotential. In der vorliegenden Version wird diese Transformation immer durchgeführt, auch wenn nur eine einzige Kontur der Methode selektiert wurde. Auch wird nicht geprüft, ob sich nicht vielleicht die selektierten Konturen bei der Implementation wieder zusammenlegen lassen, da die Information über die Nummer der Methodenkontur eventuell gar nicht benötigt wird (weil in der Methode kein Objekt erzeugt wird, keine andere Methode aufgerufen wird oder zumindest kein Unterschied in den einzelnen Methodenkonturen vorliegt).

Abschließend kann man sagen, daß Typanalyse zur statischen Berechnung einer Verteilungsstrategie eine notwendige Voraussetzung darstellt. Allerdings sind der statischen Analyse Grenzen gesetzt, wo eine dynamische Zuordnung von Aktivitäten und zu bearbeitenden Aufgaben im Programm stattfindet. Die Untersuchung von gleichartigen (monomorphen) Aktivitäten und deren Beziehungen untereinander scheint prinzipiell möglich. Der vorliegende Typinferenzalgorithmus müßte dazu aber um eine differenziertere Untersuchung von Feldern und Strukturen des lokalen Kontrollflusses innerhalb einer Methode erweitert werden. In jedem Fall ist die Lokalitätsoptimierung durch statische Typanalyse ein spannendes Feld, auf dem weitere Verbesserungen zu erwarten sind.

Literaturverzeichnis

[Age94] Ole Agesen. Constraint-Based Type Inference and Parametric Polymorphism. In Proc. of the First International Static Analysis Symposium, SAS'94, Namur, Belgium, September 1994. Lecture Notes in Computer Science Vol. 864, Seiten 78-100, Springer Verlag 1994, ISBN 3-540-58485-4.

[Age95] Ole Agesen. The Cartesian Product Algorithm, Simple and Precise Type Inference of Prametric Polymorphism. In Proc. of ECOOP'95, European Conference on Object-Oriented Programming, Århus, Dänemark, August 1995. Lecture Notes in Computer Science Vol. 952, Seiten 2-26, Springer Verlag, ISBN 3-540-60160-0.

[Bel93] U. Bellur, G. Craig, K.Shank, D. Lea. DIAMONDS: Principles and Philosophy. Technical Report 9313, CASE Center, Syracuse University, Juni 1993.

[Bel94] U. Bellur. A Methodology for Statistically Clustering Active Objects in Distributed Systems. Doctoral Dissertation, Syracuse University, Mai 1994.

[Cha90] Craig Chambers, David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90), Juni 1990, White Plains, New York, USA. SIGPLAN Notices 25(6), Juni 1990, Seiten 150-162.

[Dol97] Julian Dolby. Automatic Inline Allocation of Objects. In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97), Juni 1997, Las Vegas, Nevada, USA. SIGPLAN Notices 32(5), Mai 1997, Seiten 7-17.

[Feo92] John Feo, Editor. A comparative study of parallel programming languages: the Salishan problems. Special topics in supercomputing, Amsterdam 1992, ISBN 0-444-88135-2.

[Lea97] Doug Lea. Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley, 1997, ISBN 0-201-69581-2.

[Mil78] Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Science 17, Dezember 1978, Seiten 348-375.

[Nes97] Christian Nester. Parallelisierung rekursiver Benchmarks für JavaParty mit expliziter Datenobjekt- und Threadverteilung. Studienarbeit, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe (TH), 1997.

[Ode96] Martin Odersky, Michael Philippsen. Espresso Grinder.
http://wwwipd.ira.uka.de/~espresso, 1996.

[Ode97] Martin Odersky, Philip Wadler. Pizza into Java: Translating theory into practice. In Proc. of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France, Januar 1997.

[Oxh92] N. Oxhøj, J. Palsberg, M. Schwartzbach. Making Type Inference Practical. In Proc. of ECOOP'92, Sixth European Conference on Object-Oriented Programming, Utrecht, Niederlande, 1992. Springer-Verlag (LNCS 615), Seiten 329-349.

[Pal91] Jens Palsberg, Michael I. Schwartzbach. Object Oriented Type Inference. In Proc. of OOPSLA'91 Object-Oriented Programming Systems, Languages and Applications, Oktober 1991, Phoenix, Arizona, USA. SIGPLAN Notices 26(11), November 1991, Seiten 146-161.

[Pal92] J. Palsberg, M. Schwartzbach. Polyvariant Analysis of the Untyped Lambda Calculus. Technical Report, Daimi PB-386, Computer Science Department, Aarhus University, Dänemark, 1992.

[Phi96] Michael Philippsen, Editor. Java Seminarbeiträge. Technical Report No. 24/96, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe (TH), Juli 1996.

[Phi97a] Michael Philippsen, Matthias Zenger. JavaParty: Transparent Remote Objects in Java. Concurrency: Practice and Experience 9(11), Seiten 1225-1242, November 1997.

[Phi97b] Michael Philippsen, Matthias Zenger. JavaParty: A distributed companion to Java, http://wwwipd.ira.uka.de/JavaParty/, 1997.

[Ple93] John Plevyak, Andrew A. Chien. Incremental Inference of Concrete Types. Technical Report UIUCDCS-R-93-1829, Department of Computer Science, University of Illinois, Urbana, Illinois, USA, Juni 1993.

[Ple94] John Plevyak, Andrew A. Chien. Precise Concrete Type Inference for Objekt-Oriented Languages. In Proc. of OOPSLA'94, Object-Oriented Programming Systems, Languages and Architecture, Oktober 1994, Portland, Oregon, USA. SIGPLAN Notices 29(10), Oktober 1994, Seiten 324-340.

[Ple96] John Plevyak. Optimization of Object-Oriented and Concurrent Programs. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[Rmi97] Sun Microsystems, Inc., Mountain View, CA. Java Remote Method Invocation Specification. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html, 1997.

[Wai84] W. Waite, Gerhard Goos. Compiler Construction. Springer Verlag, Berlin, Heidelberg, New York, Berlin-New York, 1984.

[Zen97] Matthias Zenger.Transparente Objektverteilung in Java. Studienarbeit, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe (TH), Februar 1997.