FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme
Prof. Dr. Friedrich Steimann

Constraint-basierte Typinferenz für Java 5

Diplomarbeit

im Studiengang Informatik

vorgelegt von

Hannes Kegel
Jakob-Klar-Str. 7
80796 München
Matrikelnummer 6614248
hannes.kegel@gmx.de

betreut durch

Prof. Dr. Friedrich Steimann

29. Juni 2007

Danksagung

Mein besonderer Dank gilt Herrn Prof. Dr. Friedrich Steimann für die sehr intensive Betreuung dieser Arbeit und die zahlreichen weiterführenden Hinweise.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die interface-basierte Programmierung, d. h. die Deklaration von Variablen mit Interfaces anstelle von konkreten Klassen, ist eine vielfach empfohlene objektorientierte Programmiertechnik [Gamma et al. 1994; Fowler 1999; Steimann 2001]. Die Nutzung von Interfaces dient der Entkopplung von Klassen und verbessert damit die Anpassbarkeit eines Programms.

Der größtmögliche Nutzen wird erreicht, wenn die verwendeten Interfaces kontextspezifisch sind, d. h. wenn sie ausschließlich die Methoden enthalten, die für die Referenz benötigt werden [Steimann & Mayer 2005]. Dies bietet zwei wesentliche Vorteile [Steimann 2007]:

- Der Zugriff auf das Objekt wird durch die Reduzierung der angebotenen Methoden spezifisch für die jeweilige Referenz eingeschränkt. Dies ist wesentlich exakter als durch die üblichen Access modifier, die nicht auf einzelne Klienten abgestimmt werden können.
- Durch die Minimierung der benötigten Methoden und damit der Anforderungen an einsetzbare Klassen wird die Flexibilität maximiert.

Obwohl die Verwendung von Interfaces also von Vorteil ist, kommen diese vergleichsweise selten zum Einsatz (siehe auch [Steimann et al. 2003]). Ein wesentlicher Hinderungsgrund liegt in der Schwierigkeit, diejenigen Methoden zu bestimmen, die ein Interface enthalten sollte. In der Regel wird deswegen häufig die Implementierung selbst (in Form einer Klasse) als Typ einer Referenz gewählt. Dies kann für abgeschlossene, monolithische Anwendungen zunächst durchaus akzeptabel sein — gerade im Rahmen agiler Prozessmodelle wie Extreme Programming, die anstelle von Planung auf Anpassung des Quelltextes nach Bedarf setzen [Beck & Andres 2004].

Wird zu einem späteren Zeitpunkt die Flexibilität der Deklaration mit Interfaces benötigt, stellen die aktuellen Entwicklungsumgebungen für Java wie Eclipse und IntelliJ IDEA jedoch nur eingeschränkt Hilfsmittel bereit: Das Refactoring „Extract Interface“ lässt den Programmierer zwar ein neues Interface erzeugen und wie „Use Supertype Where Possible“ automatisiert für die Deklaration von Referenzen verwenden [Tip et al.

2003], die Auswahl der im Interface deklarierten Methoden bleibt jedoch dem Nutzer überlassen.

In [Steimann et al. 2006] und [Steimann 2007] wird deswegen ein neues Refactoring namens „Infer Type“ vorgeschlagen, das — angewendet auf ein Deklarationselement[1] eines Programms — einen maximal verallgemeinerten Typ berechnet (d. h. einen Typ, der nur die benötigten Methoden deklariert), wenn nötig erzeugt und für das Deklarationselement einführt. In Abbildung 1.1 wird ein Programm (a) ohne Verwendung von Interfaces gezeigt. Durch die Anwendung von „Infer Type“ auf das Feld `x` soll Programm (b) erzeugt werden. Die Klassen `Client` und `Server` werden entkoppelt, der Typ der Variable `x` deklariert nur noch die benötigte Methode `n()`. Diese Inferenz maximal verallgemeinerter Typen und das darauf aufbauende Refactoring „Infer Type“ sind Gegenstand dieser Arbeit.

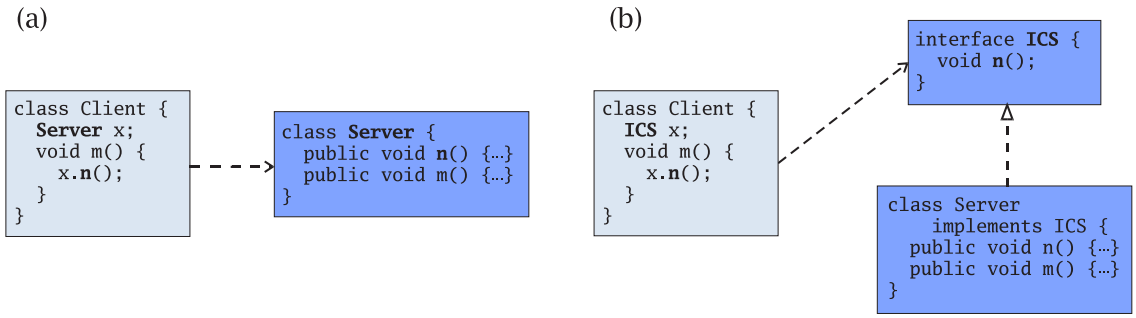

Abbildung 1.1: (a) Herkömmliche Programmierweise ohne Interfaces.
(b) Programm nach Anwendung des Refactorings mit spezifischem Interface für `x` (aus [Steimann 2007]).

1.2 Beitrag der Arbeit

Es gibt bereits eine Reihe von Arbeiten, die sich mit Typinferenz befassen. Ziel ist üblicherweise die Inferenz von Typinformationen für untypisierte Sprachen. Dies dient der Verbesserung des Laufzeitverhaltens oder der Lesbarkeit eines Programms (siehe z. B. [Palsberg & Schwartzbach 1994]). Anwendungsgebiete für typisierte Sprachen sind beispielsweise die Verifizierung von Downcasts [Wang & Smith 2001], die automatische Parametrisierung generischer Typen [Fuhrer et al. 2005] und die automatische Typisierung lokaler Variablen [Microsoft 2005].

Tip et al. [2003] beschreiben einen constraint-basierten Ansatz zur Verallgemeinerung von Typdeklarationen. Alle Bedingungen für die Typkorrektheit eines Programms werden dabei in einem System von Constraints erfasst. Die Lösung baut jedoch ausschließ-

[1] Als Deklarationselemente werden zusammenfassend Methoden (als Lieferanten eines Rückgabewerts), Parameter, Felder und lokale Variablen bezeichnet.

lich auf bestehenden Typen auf, für die Inferenz maximal verallgemeinerter Typen ist die Erzeugung neuer Typen erforderlich. Obwohl eine Verwandtschaft besteht, unterscheidet sich die Inferenz maximal verallgemeinerter Typen deutlich von den genannten Arbeiten (vgl. [Steimann et al. 2006; Steimann 2007]).[2]

Bisherige Implementierungen von „Infer Type“ unterstützen Java nur bis zur Version 1.4 [Steimann 2007]. Die Einführung von Generics in Java 5 erfordert beträchtliche Anpassungen — insbesondere reicht die Analyse von direkten und indirekten Zuweisungen für die gewünschte Unterstüzung nicht mehr aus.[3] Der Zeitpunkt für eine grundlegend neue Implementierung ist deswegen günstig.

Diese Arbeit untersucht, wie der constraint-basierte Ansatz nach [Tip et al. 2003] für die Inferenz maximal verallgemeinerter Typen angepasst werden kann. Dabei muss insbesondere ein neues Verfahren zur Lösung des Constraintsystems eingesetzt werden. Die neuen Sprachkonstrukte von Java 5 werden im Rahmen dieser Arbeit so durch Constraints repräsentiert, dass im Vergleich zur Modellierung von [Tip et al. 2003] eine Vielzahl zusätzlicher Anwendungen ermöglicht wird. Dies könnte ebenso den in [Tip et al. 2003] genannten Refactorings zu Gute kommen.

Auf Grundlage dieser Überlegungen wurde „Infer Type“ als Refactoring für Eclipse neu implementiert. Für die Unterstützung von Java 5 musste das Constraint-Framework von Eclipse erweitert werden. Neben der Unterstützung von Java 5 bietet die Implementierung weitere neue Möglichkeiten wie Vorschau und Undo sowie die Auswahl zusätzlicher Methoden für ein berechnetes Interface durch den Benutzer und die Erweiterung des Anwendungsbereichs auf komplette Kompilierungseinheiten oder Pakete. Ausführliche Tests durch automatisierte Anwendung auf alle Deklarationselemente größerer Projekte stellen sicher, dass die Implementierung weitgehend korrekt und alltagstauglich ist.

1.3 Aufbau der Arbeit

In Kapitel 2 wird zunächst eine kurze Einführung in die constraint-basierte Typinferenz gegeben, zudem werden die in dieser Arbeit verwendeten Begriffe für die Sprachkonstrukte von Java 5 definiert. Kapitel 3 arbeitet die Anforderungen an das neue Refactoring heraus. Dabei wird insbesondere die gewünschte Unterstützung von Java 5 erläutert.

In Kapitel 4 werden die Sprachkonstrukte von Java 5 vorgestellt und die daraus resultierenden Typconstraints erläutert. Anschließend wird beschrieben, wie auf Basis dieser Constraints ein maximal verallgemeinerter Typ berechnet und typkorrekt eingesetzt werden kann. Die Erzeugung der zusätzlichen Constraints für die Unterstützung der Generics unterliegt sehr vielen Bedingungen, deswegen wird sie in diesem Kapitel eher

[2] Zum Vergleich mit anderen Formen der Typinferenz siehe auch Abschnitt 2.1 und Abschnitt 7.4.
[3] Siehe hierzu Abschnitt 3.3.1.

beispielhaft dargestellt. Im folgenden Kapitel 5 werden die Einstiegspunkte in die Implementierung aufgeführt, die die exakte Bearbeitung beinhaltet. Das Kapitel erläutert zudem die Architektur und weitere wichtige Details der Implementierung des Refactorings.

In Kapitel 6 werden einige Anwendungsszenarien für das Refactoring vorgestellt, zudem werden die Tests der Implementierung beschrieben. Kapitel 7 enthält eine Diskussion der Einsatzmöglichkeiten und der Implementierung des Refactorings und stellt es in den Kontext verwandter Arbeiten. Kapitel 8 schließt die Arbeit mit einer kurzen Zusammenfassung und einem Ausblick ab.

2 Grundlagen und Begriffe

In diesem Kapitel werden Grundlagen und Begriffe vorgestellt, die in den folgenden Kapiteln benötigt werden. Zunächst wird die in dieser Arbeit betrachtete Form der Typinferenz eingeordnet und der constraint-basierte Ansatz vorgestellt. Anschließend werden die Begriffe definiert, die in dieser Arbeit für die Sprachkonstrukte von Java 5 verwendet werden.

2.1 Grundlagen der Typinferenz

Unter Typinferenz versteht man die automatische Bestimmung von Typinformationen für Programmelemente. Wichtiges Anwendungsgebiet sind dabei Sprachen, die eine statische Typisierung nicht vorschreiben oder nicht gestatten, wie z. B. Smalltalk [Palsberg & Schwartzbach 1994]. In einer solchen Sprache muss zur Laufzeit überprüft werden, ob der Empfänger einer Nachricht eine entsprechende Methode zur Verfügung stellt. Mittels Typinferenz können diese Sicherheitsüberprüfungen wesentlich optimiert werden. Die inferierten Typen werden dabei dem Programm nicht als Typannotationen hinzugefügt (d. h. Variablen werden im Quelltext nicht explizit mir einem Typ deklariert), sondern vom Compiler oder der Laufzeitumgebung in speziellen Strukturen festgehalten und können nach erfolgreicher Sicherheitsüberprüfung eines Programms wieder verworfen werden.

Auch für Java spielt diese Form der Typinferenz eine wesentliche Rolle: Der Bytecode sieht keine Typisierung für Variablen oder den Stack vor [Lindholm & Yellin 1999]. Zur Verifizierung, ob alle aufgerufenen Methoden für die Einträge des Stacks vorhanden sind, inferiert die JVM deswegen die Typen der jeweiligen Objekte beim Laden einer Klasse.[4] Eine spätere Überprüfung ist dann nicht mehr erforderlich.

Für eine stark typisierte objektorientierte Sprache wie Java gibt es ein weiteres Anwendungsgebiet für Typinferenz: Es können damit bestehende Typannotationen geändert werden. Dies kann zum einen eine Änderung hin zu spezifischeren Typen sein: Beispielsweise kann eine mit `Object` typisierte Variable oder ein Raw type[5] mit einem

[4] Seit Java 6 werden allerdings vom Compiler Typinformationen im zusätzlichen Class-file-Attribut `StackMapTable` hinterlegt. Der neue Verifier muss diese Typen dann nur noch überprüfen, nicht mehr inferieren.

[5] Das ist eine Verwendung eines generischen Typs ohne Typargumente. Siehe auch Abschnitt 2.3.

spezifischen Typ bzw. mit einem parametrisierten Typ redeklariert werden. Diese Form der Typinferenz wird von Refactorings wie „Infer Generic Type Arguments“ zur Verbesserung der Typsicherheit und der Lesbarkeit eines Programms durchgeführt [Fuhrer et al. 2005]. Zum anderen ist es häufig sinnvoll, einen allgemeineren Typ zu inferieren — dies steigert die Wiederverwendbarkeit eines Programmteils. Refactorings wie „Use Supertype Where Possible“, „Extract Interface“ oder „Generalize Declared Type“ verfolgen diesen Ansatz [Tip et al. 2003] und werden auch als Generalisierungsrefactorings bezeichnet.

Die Möglichkeit, einen allgemeineren Typ für eine Referenz zu verwenden, hängt davon ab, welche Felder und Methoden für diese Referenz benötigt werden. Die genannten Refactorings ziehen für die Änderung ausschließlich existierende Typen in Betracht. Werden also z. B. in einem Java-Programm auf einer Variablen des Typs `java.util.List` nur die Methoden `size()` und `iterator()` aufgerufen, kann der Typ der Variablen zu `java.util.Collection` geändert werden. In der Regel enthält der allgemeinere Typ Methoden und Felder, die für die Referenz nicht benötigt werden. Zudem kann kein allgemeinerer Typ eingesetzt werden, auch wenn nur wenige Methoden des deklarierten Typs benötigt werden, sobald eine der benötigten Methoden ausschließlich im deklarierten Typ vorkommt. Will man also die Wiederverwendbarkeit durch Einsatz allgemeinerer Typen verbessern, ist die Bestimmung eines maximal verallgemeinerten Typs wünschenswert, der ausschließlich die benötigten Methoden und Felder enthält [Steimann 2007]. Dieser Typ muss in der Regel neu erzeugt und der Typhierarchie hinzugefügt werden.

2.2 Constraint-basierte Typinferenz

Es gibt verschiedene Herangehensweisen an statische Programmanalysen, Nielson et al. [2005] geben hierzu eine detaillierte Einführung. Grundlage dieser Arbeit ist der constraint-basierte Ansatz der Refactorings „Extract Interface“ und „Use Supertype Where Possible“ [Tip et al. 2003] aus Eclipses Java Development Tools (JDT). Dieser Ansatz beruht auf [Palsberg & Schwartzbach 1994]. Das genaue Vorgehen von „Infer Type“ wird in Kapitel 4 dargestellt, an dieser Stelle soll eine kurze, informelle Einführung erfolgen:

Aus jedem Sprachkonstrukt von Java lassen sich Bedingungen (Constraints) für die Typen der beteiligten Elemente ableiten. Diese Bedingungen sind zweistellige Relationen der Art „Typgleichheit“ und „Subtyp von oder Typgleichheit“. Für eine Zuweisung muss zum Beispiel gelten, dass der Typ der rechten Seite ein Subtyp oder der gleiche Typ wie der der linken Seite ist. Aus der Zuweisung `E1 = E2` folgt also die Bedingung [E2] ≤ [E1]. Hierbei wird die Funktion, die einen Ausdruck auf Ihren Typ abbildet, mit eckigen

Klammern notiert, die Relation „Subtyp von oder Typgleichheit“ mit $\leq$. Eine vollständige Beschreibung der Notation findet sich in Tabelle 4.1. Weniger triviale Bedingungen werden zum Beispiel für Methodenaufrufe benötigt, um dynamisches Binden zu unterstützen, oder für die Unterstützung von Generics, wie sie im nächsten Kapitel dargestellt ist. Mittels dieser Constraints können alle Typbedingungen eines Programms erfasst werden und darauf basierend allgemeinere Typen für Typdeklarationen inferiert werden.

2.3 Begriffsdefinitionen für Java 5

Im Rahmen von Generics gibt es eine Reihe neuer Sprachkonstrukte, deren Bezeichnungen nicht gänzlich eindeutig sind. Als Grundlage werden für diese Arbeit — soweit möglich — die Bezeichnungen in der Eclipse-API und in [Gosling et al. 2005] in eingedeutschter Form verwendet. Die Begriffe bedingen sich teilweise gegenseitig, deswegen werden sie im Folgenden nicht in Listenform präsentiert.

Ausgangspunkt sind *generische Typen*, wie sie durch *generische Klassen* und *Interfaces* definiert werden, sowie *generische Methoden*. Zunächst werden die generischen Typen beschrieben, anschließend werden die Konzepte auf generische Methoden übertragen.

Als generischer Typ wird ein Typ bezeichnet, der *Typvariablen* deklariert. Diese Typvariablen sind nicht-qualifizierte Identifier, die im Rahmen der Typdeklaration auch als *(formaler) Typparameter* bezeichnet werden. Die Typparameter stehen nach dem Typnamen und sind durch spitze Klammern eingeschlossen. Beispielsweise deklariert `interface Collection<E>` ... die Typvariable `E`, diese ist der Typparameter des Interfaces `Collection`.

Die Bezeichnung Typvariable wird grundsätzlich für die Nutzung des Identifiers innerhalb der Deklaration des generischen Typs verwendet, z. B. für wird für die Deklaration `T t` die Variable `t` mit der Typvariable `T` deklariert. Eine namentliche Unterscheidung wäre nicht zwingend notwendig, wird aber in dieser Arbeit zur einheitlichen Darstellung mit der Eclipse-API und [Gosling et al. 2005] verwendet.

Ein generischer Typ ist kein Typ im eigentlichen Sinne, d. h. man kann keine Variable mit einem generischen Typ deklarieren. Für eine Variablendeklaration muss der generische Typ parametrisiert werden. Ein *parametrisierter Typ* besteht aus dem Namen eines generischen Typs und einer Liste von *(aktuellen) Typargumenten* in spitzen Klammern. Die Typargumente stellen die konkrete Belegung der Typparameter dar. Für die Variablendeklaration `Collection<String> c` ist `String` das Typargument für den Typparameter `T` des generischen Typs `Collection<T>`, die Variable `c` ist mit dem parametrisierten Typ `Collection<String>` deklariert.

Der Name eines generischen Typs kann auch ohne Typargumente für eine Variablendeklaration verwendet werden: In diesem Fall spricht man von einem *Raw type*. Der

Compiler verwendet dabei die Erasure des generischen Typs. Von der Verwendung von Raw types wird nachdrücklich abgeraten [Gosling et al. 2005], das Konstrukt existiert nur aus Kompatibilitätsgründen. Für diese Arbeit bedarf es keiner speziellen Darstellung von Raw types, deswegen sei für weitere Informationen auf [Gosling et al. 2005] verwiesen.

3 Anforderungen an das neue „Infer Type“-Refactoring

Dieses Kapitel arbeitet die Anforderungen an das neue „Infer Type“-Refactoring heraus und stellt in Beispielen die gewünschten Ergebnisse dar.

3.1 Einführung

Das neue „Infer Type“-Refactoring soll für ein ausgewähltes Deklarationselement einen maximal verallgemeinerten Typ [Steimann 2007] bestimmen und einführen können. Unter einem maximal verallgemeinerten Typ wird dabei ein Typ verstanden, der ausschließlich die für das Deklarationselement benötigten Methoden enthält. Eine genaue Definition, welche Methoden benötigt werden, wird in der Lösungsbeschreibung in Abschnitt 4.3.1 herausgearbeitet.

Im Allgemeinen gibt es einen solchen maximal verallgemeinerten Typ im Programm noch nicht. „Infer Type“ muss deswegen ein neues Interface mit den benötigten Methoden erstellen, in die Typhierarchie des ursprünglichen Typs einfügen und für das Deklarationselement einsetzen. Gibt es Zuweisungen zu anderen Deklarationselementen, ist es in der Regel erforderlich, diese ebenfalls zu ändern. Unter Umständen müssen hierzu weitere neue Interfaces erzeugt werden. In Abschnitt 4.3.3 wird die Lösung orientiert am grundsätzlichen Vorgehen von [Steimann 2007] dargestellt. Das neue Refactoring soll das bisherige „Infer Type“-Refactoring[6] ablösen, es muss also bezüglich der Erzeugung der maximal verallgemeinerten Typen genauso leistungsfähig sein. Als weitere funktionale Anforderungen ergeben sich folgende Punkte:

- Das Refactoring soll auf den Eclipse Java Development Tools (JDT) basieren und wie die Refactorings „Extract Interface“ und „Use Supertype Where Possible“ sowie „Infer Generic Type Arguments“ und „Generalize Declared Type“ einen constraintbasierten Ansatz [Tip et al. 2003; Fuhrer et al. 2005] verfolgen. Dies führt Abschnitt 3.2 weiter aus.

[6] Das bisherige „Infer Type“-Refactoring kann unter `http://www.fernuni-hagen.de/ps/prjs/InferType/` heruntergeladen werden.

- Zudem sollen die neuen Sprachkonstrukte von Java 5 so unterstützt werden, dass die Anwendbarkeit des Refactorings möglichst weitgehend ist. Insbesondere sollen dafür — soweit nötig — auch Typargumente parametrisierter Klassen durch das Refactoring geändert werden können. Die wesentliche Aufgabe besteht dabei in der Unterstützung von generischen Typen (Generics) — hierzu mehr in Abschnitt 3.3. Weiterhin erfordern auch kovariante Rückgabetypen und die erweiterte For-Schleife (Enhanced for loop) eine besondere Behandlung.

- Es soll Gebrauch von Eclipses Refactoring-Framework gemacht werden, um dem Benutzer eine einheitliche Darstellung zu bieten und von dessen Vorteilen wie einer Vorschau zu profitieren.

- Eine neue Möglichkeit, die gesamte Verwendung eines Typs in einer Übersetzungseinheit oder in einem Paket zu betrachten, soll durch Anwendung auf Importdeklarationen geschaffen werden. Dies wird in Abschnitt 3.4 näher beleuchtet.

- Nach Bestimmung der benötigten Methoden soll „Infer Type“ dem Benutzer ähnlich wie „Extract Interface“ die Möglichkeit bieten, dem neuen Interface zusätzliche Methoden aus der Hierarchie des ursprünglichen Typs hinzuzufügen.

3.2 Inferenz mittels Typconstraints

Eine kurze Einführung der Typconstraints von „Extract Interface“, „Generalize Declared Type“ und „Use Supertype Where Possible“ wurde in Abschnitt 2.2 gegeben. Die Constraints und deren Lösung bauen allerdings ausschließlich auf existierenden Typen auf [Tip et al. 2003]. Für „Infer Type“ muss der zugrundeliegende Ansatz so erweitert werden, dass auch neue Typen auf Basis der verwendeten Methoden erzeugt werden können. Insbesondere muss ein neuer Algorithmus zum Einsatz kommen. Für die Bestimmung der benötigten Protokolle werden den Constraintvariablen dabei nicht Mengen von Typen sondern Mengen von Methoden als Wertebereich zugeordnet. Anschließend soll für das ausgewählte Deklarationselement unter Erzeugung möglichst weniger weiterer neuer Typen ein maximal verallgemeinerter Typ eingesetzt werden. Dies soll ebenfalls auf Basis der erzeugten Typconstraints erfolgen.

Einige eher technische Klassen der genannten Refactorings wie eine eigene Typumgebung, um Speicher zu sparen, oder einige Klassen für die Constraintvariablen können wiederverwendet werden. Die Constraintgenerierung und -lösung muss jedoch neu erstellt werden. Die Implementierungen der constraint-basierten Refactorings der Java Development Tools stellen dabei eine Vorlage dar.

3.3 Unterstützung von Generics

3.3.1 Grundlagen

Mit Java 5 wurden generische Typen (Generics) Teil der Sprache. Durch diese ergeben sich eine Reihe neuer Regeln für die Typkorrektheit eines Programms. „Infer Type“ muss diese Regeln mit den beschriebenen Typconstraints erfassen, um ein korrektes Ergebnis zu liefern. Die Anforderungen an die Repräsentation gehen hierbei aber über die reine Typkorrektheit hinaus. „Infer Type“ soll in der Lage sein, Typargumente von parametrisierten Klassen zu ändern, wenn dies für die Redeklaration des ausgewählten Deklarationselements erforderlich ist. Der folgende Quelltext gibt ein einfaches Beispiel, in dem ein Typargument bei Anwendung von „Infer Type“ auf `obj` geändert werden müsste:

```
1  class Client {
2      private List<ClassOne> container = new ArrayList<ClassOne>();
3      public void action(ClassOne obj) {
4          obj.m1();
5          container.add(obj);
6      }
7  }
8  class ClassOne implements IfOne {
9      public void m1() { /* ... */ }
10     public void m2() { /* ... */ }
11     public void m3() { /* ... */ }
12 }
13 class IfOne {
14     public void m1() { /* ... */ }
15 }
```

Quelltext 3.1: Einfache Verwendung parametrisierter Container

Auf dem Deklarationselement `obj` in Quelltext 3.1 wird ausschließlich die Methode `m1()` aufgerufen. Mit `IfOne` existiert also bereits ein maximal verallgemeinerter Typ. Der Typ kann jedoch für `obj` nur eingesetzt werden, wenn das Typargument von `container` ebenso geändert wird. Die Verwendung von `container` muss in diesem Fall weiter verfolgt werden.

Das Refactoring „Use Supertype Where Possible“ kann für `obj` den Typ `ClassOne` nicht durch `IfOne` ersetzten, da es Typargumente als Konstanten betrachtet. Auch das entsprechende Refactoring von IntelliJ IDEA [Jetbrains s.r.o. 2007] kann für Quelltext 3.1 keine Änderung durchführen. Für ein äquivalentes Programm, das das Collections-Framework unparametrisiert verwendet (d. h. so wie in Java 1.4), stellt sich dieses Problem nicht. Die Verwendung von Generics schränkt also die Nutzbarkeit der Generalisierungsrefactorings im Vergleich zu äquivalenten nicht-generischen Klassen signifikant[7]

[7] Die tatsächliche Anzahl an verhinderten Änderungen wurde im Rahmen dieser Arbeit für einige Projekte untersucht. Die Ergebnisse finden sich in Abschnitt 6.4.

ein. Das neue „Infer Type“-Refactoring soll die in Quelltext 3.2 angegebene Lösung erzeugen:

```
1 class Client {
2     private List<IfOne> container = new ArrayList<IfOne>();
3     public void action(IfOne obj) {
4         obj.m1();
5         container.add(obj);
6     }
7 }
8 /* ... */
```

Quelltext 3.2: Einfache Verwendung parametrisierter Container — Lösung

Wird `container` an weiteren Stellen verwendet, können zusätzliche Methoden für den maximal verallgemeinerten Typ erforderlich sein: Quelltext 3.3 zeigt ein Beispiel, in dem `Client` die zusätzliche Methode `checkFirst()` enthält. Das Deklarationselement `first` muss in diesem Fall ebenfalls mit dem neuen Typ deklariert werden. Als Konsequenz daraus muss der maximal verallgemeinerte Typ auch die Methode `m2()` enthalten. „Infer Type“ muss in diesem Fall also ein neues Interface erstellen.

```
1 class Client {
2     private List<ClassOne> container = new ArrayList<ClassOne>();
3     public void action(ClassOne obj) {
4         obj.m1();
5         container.add(obj);
6     }
7     public void checkFirst() {
8         ClassOne first = container.get(0);
9         first.m2();
10    }
11 }
12 /*...*/
13 }
```

Quelltext 3.3: Mehrfache Verwendung parametrisierter Container

Kapitel 4 beinhaltet weitere Beispiele, die den nötigen Umfang der Unterstützung zeigen. Insbesondere müssen Wildcards, Schranken von Typparametern, verschachtelt parametrisierte Typen und Subtyping von parametrisierten Typen behandelt werden.

Das in den Java Development Tools vorhandene Refactoring „Infer Generic Type Arguments“ erzeugt ebenfalls Constraints, deren Variablen Typargumente repräsentieren [Fuhrer et al. 2005]. Die Behandlung der Konstrukte von Java 5 ist jedoch nicht vollständig: Schranken von Typparametern oder die erweiterte For-Schleife werden zum Beispiel in der aktuellen Implementierung nicht erfasst, da dies für „Infer Generic Type Arguments“ nicht erforderlich ist. Auch verschachtelt parametrisierte Typen werden von der

aktuellen Implementierung nicht vollständig unterstützt. In der Regel werden dadurch nur einige Deklarationselemente nicht vollständig parametrisiert, im Zusammenhang mit generischen Methoden kann dies aber auch zu Kompilierungsfehlern führen. Für Quelltext 3.4 wird so das nicht kompilierbare Ergebnis in Quelltext 3.5 erzeugt.

```
1 class Client {
2     public void addNestedLists() {
3         List container = new ArrayList();
4         container.add(Collections.singletonList(
5                         Collections.singletonList("")));
6     }
7 }
```

Quelltext 3.4: Verschachtelte Parametrisierung

```
1 class Client {
2     public void addNestedLists() {
3         List<List<List>> container = new ArrayList<List<List>>();
4         container.add(Collections.singletonList(
5                         Collections.singletonList("")));
6     }
7 }
```

Quelltext 3.5: Ergebnis von „Infer Generic Type Arguments“

Auch wenn „Infer Generic Type Arguments“ einen möglichst spezifischen Typ anstelle eines möglichst generellen Typs bestimmt, kann das grundsätzliche Verfahren zur Constrainterzeugung — nicht jedoch das der Lösung des Constaintsystems — von „Infer Type“ verwendet werden. Da „Infer Type“ im Gegensatz zu „Infer Generic Type Arguments“ Typargumente von bereits parametrisierten Klassen ändern soll, muss es zur korrekten Behandlung allerdings vervollständigt werden, so dass alle Sprachkonstrukte von Java 5 durch Constraints repräsentiert werden.

Abgesehen von der erforderlichen Korrektheit des Refactorings haben komplexere Konstrukte durchaus praktische Relevanz. So soll „Infer Type“ auch die Parametrisierung von Supertypen — falls erforderlich — ändern können. Folgendes Beispiel zeigt einen solchen Fall für eine anonyme Klasse:

```
1 class Client {
2     private List<ClassOne> container = new ArrayList<ClassOne>();
3     public void action(ClassOne obj) {
4         container.add(obj);
5     }
6     public void sortByName() {
7         Collections.sort(container, new Comparator<ClassOne>() {
8             public int compare(ClassOne o1, ClassOne o2) {
9                 return o1.getName().compareTo(o2.getName());
```

```
10              }
11          });
12      }
13  }
14  class ClassOne {
15      public void m1() { /* ... */ }
16      public void m2() { /* ... */ }
17      public String getName() { /* ... */ }
18  }
```

Quelltext 3.6: Parametrisierter Supertyp

Das von „Infer Type“ gewünschte Ergebnis bei Anwendung auf `obj` ist in Quelltext 3.7 angegeben. Hierbei muss nicht nur das Typargument von `container` geändert werden, sondern ebenso das Argument des Supertyps der anonymen `Comparator`-Klasse und die Methodenparameter von `compare`. Es müssen also auch Deklarationselemente geändert, die nicht durch direkte oder indirekte Zuweisungen mit `obj` verbunden sind. Dies ist eine grundsätzliche Neuerung, die durch den Einsatz von Generics entsteht.

```
1  class Client {
2      private List<InferredIf> container = new ArrayList<InferredIf>();
3      public void action(InferredIf obj) {
4          container.add(obj);
5      }
6      public void sortByName() {
7          Collections.sort(container, new Comparator<InferredIf>() {
8              public int compare(InferredIf o1, InferredIf o2) {
9                  return o1.getName().compareTo(o2.getName());
10             }
11         });
12     }
13 }
14 class ClassOne implements InferredIf { /* ... */
15 }
16 interface InferredIf {
17     void String getName();
18 }
```

Quelltext 3.7: Parametrisierter Supertyp — Lösung

Das Beispiel zeigt außerdem die Anwendung einer generischen Methode (`Collections.sort`). Für die Typparameter generischer Methoden inferiert der Compiler einen möglichst spezifischen Typ. „Infer Type“ muss die Argumente generischer Methoden so ändern, dass der Compiler wieder einen gültigen Typ inferieren kann. Für den Ausgangsquelltext inferiert der Compiler `ClassOne` als Typargument, für das Ergebnis `InferredIf`. Damit dies möglich ist, muss — nach Änderung des Typs von `container` — auch die Parametrisierung der anonymen `Comparator`-Klasse

geändert werden. Bei einer ursprünglichen Parametrisierung mit `Object` wäre keine Änderung erforderlich, da dann direkt `InferredIf` als neues Typargument inferiert werden könnte.

3.3.2 Anwendung auf generische Typen

„Infer Type“ soll auf generische Typen genau so wie auf einfache Typen angewendet werden können. Der neue Typ ist in diesem Fall ebenfalls ein generischer Typ. Die wesentliche Aufgabe ist die korrekte Handhabung der Typparameter sowie die richtige Parametrisierung des neuen Typs im Deklarationskopf des alten. Komplexe Fälle entstehen hier durch Schachtelung parametrisierter Typen, die sich auf verschiedene Hierarchiestufen verteilt. Hier ein etwas einfacheres Beispiel mit einer generischen Typhierarchie:

```
1  class Client {
2      public static void main(String[] args) {
3          GenA<Integer, String> a = new GenA<Integer, String>();
4          a.a1(1, "");
5          GenB<String> b = a;
6          b.b1("");
7      }
8  }
9  class GenA<E, T> extends GenB<T> {
10     public void a1(E e, T t) { }
11     public void a2(E e) { }
12 }
13 class GenB<B> {
14     public void b1(B b) { }
15     public void b2(B b) { }
16 }
```

Quelltext 3.8: Refactoring eines generischen Typs

Das gewünschte Ergebnis bei Anwendung auf a wird im nächsten Quelltext dargestellt. Für diesen Beispiel müssen zwei neue Typen eingeführt werden (dies wird in Abschnitt 4.3.3 erläutert). Das Parametrisierungsverhältnis entspricht dem der ursprünglichen Typen.

```
1  class Client {
2      public static void main(String[] args) {
3          IfA<Integer, String> a = new GenA<Integer, String>();
4          a.a1(1, "");
5          IfB<String> b = a;
6          b.b1("");
7      }
8  }
9  class GenA<E, T> extends GenB<T> implements IfA<E, T> { /* ... */
10 }
```

```
11 class GenB<B> implements IfB<B> { /* ... */
12 }
13 interface IfA<E, T> extends IfB<T> {
14     void a1(E e, T t);
15 }
16 public interface IfB<B> {
17     void b1(B b);
18 }
```

Quelltext 3.9: Refactoring eines generischen Typs — Lösung

3.3.3 Anwendung auf Typargumente

In den bisherigen Beispielen wurde die Anwendung für den Haupttyp eines Deklarationselements gezeigt. Um Typen zu entkoppeln, kann es auch nötig sein, Typargumente auszuwählen, wie das folgende Beispiel zeigt:

```
1 class Client {
2     private List<ClassOne> container = new ArrayList<ClassOne>();
3     public void action(int index) {
4         container.get(index).m1();
5     }
6 }
7 class ClassOne {
8     public void m1() { /* ... */ }
9     public void m2() { /* ... */ }
10 }
```

Quelltext 3.10: Anwendung auf Typargumente

Hier gibt es kein Deklarationselement vom Typ `ClassOne`, trotzdem besteht eine Abhängigkeit. Damit die Klassen voneinander entkoppelt werden können, soll deswegen das Typargument `ClassOne` von `container` auswählbar sein und folgende Lösung liefern:

```
1 class Client {
2     private List<InferredIf> container = new ArrayList<InferredIf>();
3     public void action(int index) {
4         container.get(index).m1();
5     }
6 }
7 class ClassOne implements InferredIf { /* ... */
8 }
9 interface InferredIf {
10     void m1();
11 }
```

Quelltext 3.11: Anwendung auf Typargumente — Lösung

3.4 Refactoring von Importdeklarationen

Bei den bisher beschriebenen Anwendungsfällen wird immer ein einzelnes Deklarationselement ausgewählt. Für die Entkopplung zweier Klassen muss man jedoch alle Deklarationselemente eines Typs in der jeweils anderen Klasse durch ein Interface ersetzen. Hängen die Deklarationselemente nicht über Zuweisungen oder über Typargumente wie in Quelltext 3.3 zusammen, reicht eine einmalige Anwendung von „Infer Type“ auf eines der Deklarationselemente nicht aus. Man kann nun „Infer Type“ zur Abhilfe sukzessive auf die nicht zusammenhängenden Deklarationselemente anwenden. Dabei werden jedoch in der Regel verschiedene neue Typen erzeugt werden, je nachdem welche Methoden auf den einzelnen Deklarationselementen aufgerufen werden.

Um diesen Anwendungsfall besser zu lösen, soll „Infer Type“ auf Typimportdeklarationen anwendbar sein. Es soll in diesem Fall einen gemeinsamen neuen Typ für alle Vorkommen in einer Kompilierungseinheit erstellen und einsetzen. Als zusätzliche Option soll der Anwendungsbereich auch auf das gesamte Paket oder auf das Paket inklusive Unterpaketen ausgeweitet werden können. Quelltext 3.12 zeigt einen solchen Fall: Die Klasse `Target` ist Parameter der beiden Methoden `actionOne` und `actionTwo`, wobei unterschiedliche Methoden benötigt werden. Bei einzelner Anwendung auf die beiden Parameter werden so zwei neue Interfaces erzeugt. Häufig wird das nicht erwünscht sein, da eine hohe Zahl von Typen nicht zur Übersichtlichkeit beiträgt und der gewünschte Effekt auch mit einem einzigen neuen Typ möglich wäre.

```
1  package mypackage;
2
3  import otherpackage.Target;
4
5  public class Server {
6      public void actionOne(Target t) {
7          t.m1();
8      }
9      public void actionTwo(Target t) {
10         t.m2();
11     }
12 }
13 ----------------------------------------
14 package otherpackage;
15
16 class Target {
17     public void m1() { /* ... */ }
18     public void m2() { /* ... */ }
19     public void m3() { /* ... */ }
20 }
```

Quelltext 3.12: Anwendung auf Importdeklarationen

Angewendet auf die Importdeklaration, die die Verwendung des Typs in der ganzen Kompilierungseinheit repräsentiert, soll „Infer Type“ folgende Lösung liefern:

```
1  package mypackage;
2
3  import otherpackage.TargetIf;
4
5  public class Server {
6      public void actionOne(TargetIf t) {
7          t.m1();
8      }
9      public void actionTwo(TargetIf t) {
10         t.m2();
11     }
12 }
13 ----------------------------------------
14 package otherpackage;
15
16 class Target implements TargetIf {
17     public void m1() { /* ... */ }
18     public void m2() { /* ... */ }
19     public void m3() { /* ... */ }
20 }
21 interface TargetIf {
22     void m1();
23     void m2();
24 }
```

Quelltext 3.13: Anwendung auf Importdeklarationen — Lösung

4 Die Typconstraints und ihre Lösung

Abschnitt 4.1 gibt eine kurze Einführung und erläutert den Ablauf der Analyse. In Abschnitt 4.2 werden die relevanten Sprachkonstrukte von Java und die dazugehörigen Typconstraints besprochen. Abschnitt 4.3 erläutert den Algorithmus, der auf Basis der Typconstraints einen maximal verallgemeinerten Typ berechnet und für die Auswahl einführt.

4.1 Einführung

Das Ziel von „Infer Type“ ist es, für ein ausgewähltes Deklarationselement einen maximal verallgemeinerten Typ zu bestimmen und so einzusetzen, dass das Programm danach wieder typkorrekt ist. Dazu werden alle Bedingungen (Constraints), die für die Typkorrektheit erfüllt sein müssen, als System von Relationen erfasst. Die Variablen dieses Systems repräsentieren die Typen der Deklarationselemente und Ausdrücke eines Programms. Ein bestimmtes Programm ist typkorrekt, wenn alle daraus bestimmten Constraints erfüllt sind.

Der constraint-basierte Ansatz wird bereits für mehrere Refactorings von Eclipse eingesetzt. Tip et al. [2003] beschreiben dies für Generalisierungsrefactorings (insbesondere für „Use Supertype Where Possible“ und „Extract Interface“). Fuhrer et al. [2005] zeigen, wie Typconstraints für das Refactoring „Infer Generic Type Arguments“ eingesetzt werden.

„Infer Type“ kann grundsätzlich dieselbe Art von Constraints verwenden, ein wesentlicher Unterschied besteht jedoch in der Zielsetzung. Die oben genannten Refactorings bestimmen zu verwendende Typen auf Basis der bestehenden Typhierarchie. „Infer Type“ soll einen neuen maximal verallgemeinerten Typ bestimmen und einführen, der ausschließlich die verwendeten Methoden enthält. Die Lösung der Constraints unterscheidet sich deshalb stark. Für „Infer Type“ wird ein zweistufiger Algorithmus verwendet: Zunächst werden dabei die Constraintvariablen mit Mengen von Methoden belegt und eine minimale Lösung gesucht, d. h. die Belegung, bei der aus keiner der Mengen eine Methode entfernt werden kann, ohne dass ein Constraint verletzt wird. Anschließend wird nach einem existierenden Typ gesucht oder ein neuer Typ erstellt, der nur die benötigten Methoden deklariert. Im zweiten Schritt des Algorithmus wird dann — ausge-

hend von der Verwendung des maximal verallgemeinerten Typs für die Auswahl — nach einer korrekten Belegung der Constraintvariablen mit Typen gesucht. Da die Constraints selbst jedoch nur die Typisierungsregeln von Java repräsentieren, unterscheiden sich diese ansonsten im Wesentlichen durch die vollständige Unterstützung von Java 5.[8]

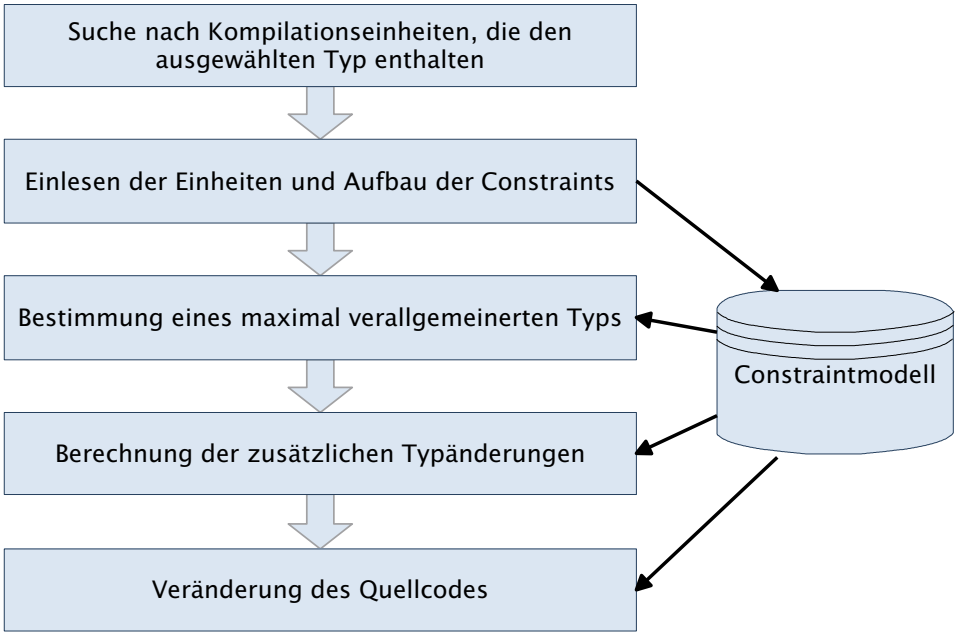

Abbildung 4.1: Ablauf von „Infer Type“

Der Ablauf von „Infer Type“ ist schematisch in Abbildung 4.1 dargestellt. „Infer Type“ sammelt zunächst alle Kompilationseinheiten, aus denen relevante Constraints erzeugt werden könnten. Dies geschieht in der aktuellen Implementierung durch Suche nach dem Typ der Auswahl und dessen Supertypen, die im Quelltext vorliegen. Davon ausgehend wird nach der Verwendung von Feldern mit einem dieser Typen und Methoden, die einen dieser Typen zurückgeben, gesucht. Wird der Typ als Typargument eines Supertyps nach `extends` oder `implements` verwendet (z. B. wie in `class MyList extends ArrayList<Target>`), muss auch nach dem Subtyp gesucht werden (also nach `MyList`). Dies ist erforderlich, da z. B. für `MyList myList` und `t1` als Methode von `Target` der Ausdruck `myList.get(0).t1()` den Typ `Target` ohne explizite Nennung verwendet. Damit werden alle erforderlichen Kompilationseinheiten erfasst. Auf Verbesserungsmöglichkeiten dieser Suche wird in Abschnitt 7.2 eingegangen.

Anschließend werden die Kompilationseinheiten eingelesen und daraus die Constraints erstellt. Dies wird in Abschnitt 4.2 erläutert. Die erste Stufe des Lösungsalgorithmus bestimmt dann aus den Constraints den maximal verallgemeinerten Typ. In der zweiten Stufe berechnet der Algorithmus die nötigen Typänderungen, so dass alle

[8] Schranken von Typparametern werden allerdings in dieser Arbeit (noch) so in Constraints gefasst, dass diese nicht geändert werden können (siehe Abschnitt 4.2.7).

Constraints wieder erfüllt sind. Abschließend wird für alle Constraintvariablen, die ein Deklarationselement repräsentieren und deren Typ geändert wurde, der entsprechende Knoten im Abstract syntax tree bestimmt und geändert.

4.2 Aufbau der Constraints

4.2.1 Grundlagen

Für jedes Vorkommen eines Sprachkonstrukts von Java werden beim Einlesen einer Kompilationseinheit Typconstraints erstellt, die der Semantik von Java entsprechen. Die Sprachkonstrukte und die dazugehörigen Constraints werden im Folgenden vorgestellt. Hierzu wird eine an [Tip et al. 2003] und [Fuhrer et al. 2005] angelehnte Notation verwendet; diese ist in Tabelle 4.1 angegeben.

Constraintvariablen	
$[E]$	Typ eines Ausdrucks oder eines Deklarationselements *E*.
$[M]$	Typ der Methode *M* (Der Typ einer Methode ist ihr Rückgabetyp).
$[Param(i, M)]$	Der Typ des i-ten formalen Parameters der Methode *M*.
$Decl(F)$	Typ, der das Feld *F* deklariert.
$Decl(M)$	Typ, der die Methode *M* deklariert.
$Elem(E, T)$	Das tatsächliche Typargument des Ausdrucks *E* für den Typparameter *T*.
$Elem(C, T)$	Das tatsächliche Typargument des Typs *C* für den Typparameter *T*.
C	Konstanter Typ *C*.
Constraints	
$T_1 \leq T_2$	T_1 muss Subtyp von T_2 oder mit T_2 identisch sein. Dies wird im Folgenden als Subtypconstraint bezeichnet.
$T_1 = T_2$	T_1 muss mit T_2 identisch sein. Dies wird im Folgenden als Gleichheitsconstraint bezeichnet.
Funktion zur Methodenbestimmung	
$RootDefs(M)$	Alle Methoden, die von M überschrieben bzw. implementiert werden.

Tabelle 4.1: Notation von Typconstraints

Die erzeugten Typconstraints sind Relationen zwischen zwei Constraintvariablen der Art „Typgleichheit“ (=) oder „Subtyp oder Typgleichheit“ (≤). Die Relation ≤ ist eine Halbordnung, d. h. sie ist reflexiv, transitiv und antisymmetrisch. Für $[E_2] \leq [E_1]$ und $[E_1] \leq [E_2]$ gilt also $[E_2] = [E_1]$. Constraintvariablen sind Typen von Ausdrücken

Sprachkonstrukt	*Constraints*
Zuweisung $E_1 = E_2$	$[E_2] \leq [E_1]$
Methodenaufruf $E.m(E_1, ..., E_n)$ einer Methode M	$[E.m(E_1, ..., E_n)] = [M]$ $[E_i] \leq [Param(i, M)]$ $[E] \leq Decl(M_1)$ oder ... oder $[E] \leq Decl(M_k)$ mit $RootDefs(M) = \{M_1, ..., M_k\}$
Feldzugriff $E.f$ auf das Feld F	$[E.f] = [F]$ $[E] \leq Decl(F)$
`return` E in einer Methode M	$[E] \leq [M]$
Deklaration einer Methode M	$[Param(i, M_j)] = [Param(i, M)]$ $[M] = [M_j]$ (bis Java 1.4) oder $[M] \leq [M_j]$ (ab Java 5) für alle $M_j \in RootDefs(M)$
Konstruktoraufruf `new` $C(...)$	[`new` $C(...)$] = C Parameterbehandlung wie bei Methodenaufruf
Cast $(C)E$	$[(C)E] = C$
Implizite oder explizite Verwendung von `this`	[`this`] = $Decl(M)$

Tabelle 4.2: Behandlung grundlegender Sprachkonstrukte von Java

oder Deklarationselementen ([*E*], [*M*], [*Param*(*M, i*)]), deklarierende Typen (*Decl*(*F*) und *Decl*(*M*)), Elementvariablen, die Typargumente repräsentieren (*Elem*(*E, T*)) und konstante Typen (*C*). Es sind also nicht — wie im eigentlichen Sinne zu erwarten wäre — alle Constraintvariablen tatsächlich veränderlich. Weitere konstante Constraintvariablen sind die deklarierenden Typen von Feldern und von Methoden, da der Ort der Deklaration von „Infer Type“ nicht geändert wird.[9] Einige weitere, spezielle Constraintvariablen, die als Hilfskonstrukte dienen, werden in den kommenden Abschnitten vorgestellt.

Tabelle 4.2 führt grundlegende Sprachkonstrukte von Java und die zugehörigen Constraints auf. Quelltext 4.1 zeigt ein einfaches Programm, Tabelle 4.3 die dazugehörigen Constraints.

```
1 class Sample extends Base {
2     public void m1() { }
3     public void m2() { }
4 }
5
6 class Base {
7     public void m1() { }
8     public void m2() { }
```

[9] Wenn „Infer Type“ einen neuen Typ mit benötigten Methoden einführt, bekommt RootDefs(M) ein weiteres Element. Es würde also für die dargestellte Behandlung in Tabelle 4.2 ein zusätzlicher Constraint erstellt und keine bestehende Constraintvariable geändert.

```
9  }
10
11 class Client {
12     public void client() {
13         Sample sample = new Sample(); /* Auswahl */
14         sample.m2();
15
16         Base base = sample;
17         base.m1();
18     }
19 }
```

Quelltext 4.1: Beispiel für Typconstraints

Zeile	*Constraints*
13	[*new Sample*()] = *Sample*, [*new Sample*()] ≤ [*sample*]
14	[*sample*] ≤ *Decl*(*Base.m2*())
16	[*sample*] ≤ [*base*]
17	[*base*] ≤ *Decl*(*Base.m1*())

Tabelle 4.3: Relevante Constraints zu Quelltext 4.1

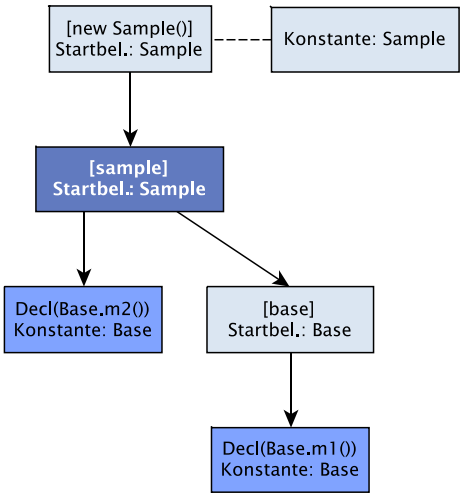

Abbildung 4.2: Constraintgraph zu Quelltext 4.1

Die Typconstraints können auch als Graph mit den Variablen als Knoten und den Constraints als Kanten verstanden werden. Da der Lösungsalgorithmus von „Infer Type“ diesen Ansatz verwendet, wird für Quelltext 4.1 in Abbildung 4.2 auch eine Graphendarstellung der relevanten Constraints angegeben. Ungerichtete Kanten repräsentieren Gleichheitsconstraints, gerichtete Kanten Subtypconstraints, wobei der Pfeil auf die rech-

te Seite, also den Supertyp, zeigt. Der dunkle Knoten entspricht dem ausgewählten Deklarationselement. Für alle Constraintvariablen wird zudem die Startbelegung, d. h. der Wert für das Ausgangsprogramm, angegeben.

4.2.2 Bestimmung überschriebener Methoden

Um die Überschreibungsregeln von Java zu beachten, muss „Infer Type“ für jede Methodendeklaration alle überschriebenen bzw. implementierten Methoden bestimmen und Gleichheitsconstraints zwischen den Parametern sowie Subtypconstraints zwischen den Rückgabewerten anlegen. „Use Supertype Where Possible“ erzeugt dazu für jede Methodendeklaration Constraints zwischen der Methode und den in der Hierarchie am höchsten gelegenen überschriebenen oder implementierten Methoden. Für Fälle wie in Abbildung 4.3 werden so über die Methode aus `BaseIf` alle Methoden miteinander verknüpft.

Die Verwendung der in der Hierarchie am höchsten gelegenen Methoden reicht jedoch für manche Fälle nicht aus.[10] Abbildung 4.4 zeigt eine Klassenhierarchie, in der überschriebene Methoden nicht über einen gemeinsamen Supertyp zusammenhängen. Für eine Verwendung dieser Klassen wie in Quelltext 4.2 ergibt sich mit dieser Behandlung ein fehlerhaftes Ergebnis.

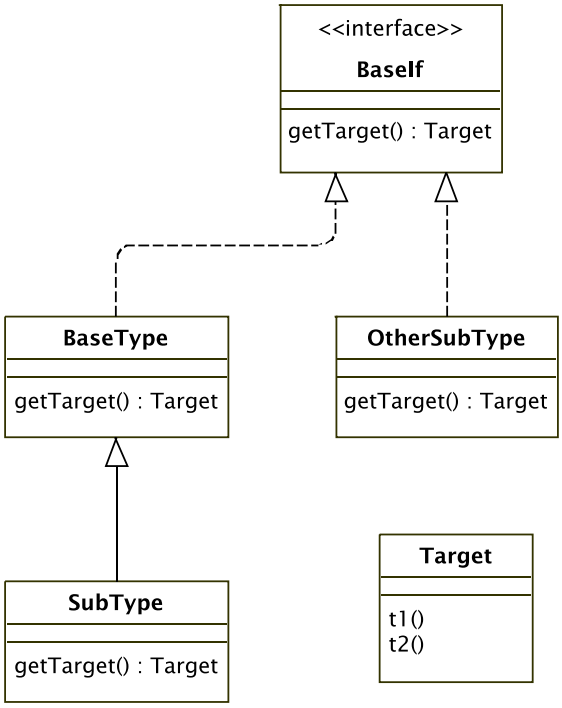

Abbildung 4.3: Beispiel einer Klassenhierarchie mit überschriebenen Methoden

[10] Siehe auch Bugreport Nr. 187034 auf `https://www.eclipse.org/bugs/`.

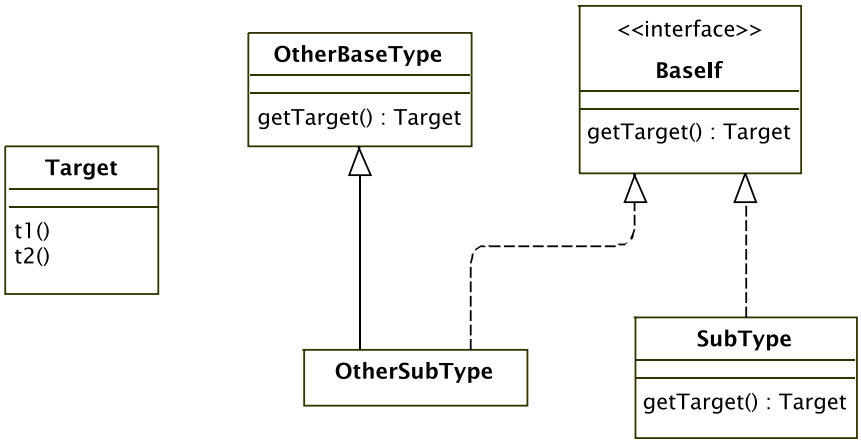

Abbildung 4.4: Beispiel einer Klassenhierarchie, in der die Bestimmung verwandter Methoden über die Supertypen hinausgehen muss

```
1 class Client {
2     public void client() {
3         new OtherSubType().getTarget().t1();
4         new SubType().getTarget().t2();
5     }
6 }
```

Quelltext 4.2: Verwendung der Hierarchie aus Abbildung 4.4

Für eine öffentliche, nicht-statische Methode *M*, die in einer Klasse *C* deklariert ist, führt „Infer Type“ deswegen folgende zusätzliche Suche durch:

1. Bestimme die Blätter des Subtypbaums.
2. Bestimme die Supertypen dieser Blätter, die Interfaces sind und in der bisherigen Suche noch nicht behandelt wurden.
3. Lege für alle Methoden *N* dieser Supertypen, die dieselbe bzw. kovariante Signatur wie *M* haben, zusätzliche Constraints an.

Die Bestimmung dieser zusätzlichen Methoden wird z. B. auch von Eclipses „Rename“-Refactoring durchgeführt. Dabei müssen jedoch in einem Schritt alle auf diese Weise zusammenhängenden Methoden bestimmt werden. Für „Infer Type“ ergibt sich die Verkettung dieser „Heiratsfälle“[11] (`OtherSubType` verheiratet `OtherBaseType` und `BaseIf` für die Hierarchie aus Abbildung 4.4) dadurch, dass die beschriebene Suche unabhängig für alle relevanten Methodendeklarationen durchgeführt wird. Eine Wiederverwendung

[11] Die Bezeichung „marriage cases“ wird in den Kommentaren von `org.eclipse.jdt.internal.corext.refactoring.rename.RippleMethodFinder2` verwendet.

des Suchalgorithmus des „Rename“-Refactorings würde deswegen zu erheblichen Geschwindigkeitsnachteilen führen.[12]

Für Rückgabewerte überschriebener Methoden werden Gleichheitsconstraints anstelle von Subtypconstraints erzeugt, wenn im Ausgangsprogramm Typgleichheit der Rückgabetypen vorliegt. Für den in Abschnitt 4.3 beschriebenen Algorithmus ergibt sich daraus keinen Unterschied. Für zukünftige Anwendungen der Constraints kann damit eine unerwünschte Abhängigkeit von Java 5 vermieden werden — dieses Vorgehen kann aber auch jederzeit geändert werden.

4.2.3 Deklarationselemente mit parametrisierten Typen

Im Vergleich zu „Use Supertype Where Possible“ erfasst „Infer Type“ die neuen Sprachkonstrukte von Java 5 so, dass das Refactoring auch angewendet werden kann, wenn Typargumente geändert werden müssen (siehe Abschnitt 3.3). Die in Tabelle 4.2 genannten Regeln müssen dafür um eine Reihe weiterer Regeln ergänzt werden. Im Folgenden wird das Vorgehen beschrieben, die Einstiegspunkte in die algorithmische Behandlung finden sich im Rahmen der Implementierungsbeschreibung in Abschnitt 5.4.

Der wesentliche Ansatzpunkt ist die Zerlegung einer Constraintvariablen für einen parametrisierten Typ in mehrere Constraintvariablen einfacher Typen. Der ursprünglichen Constraintvariablen werden dabei Elementvariablen zugeordnet, die die Belegung eines Typparameters mit einem Typargument repräsentieren. Durch Subtyping kann es hierbei mehrere Typparameter geben, die mit demselben Typargument belegt sind.

Abbildung 4.5 zeigt, wie dies für den Typ `Map<String, List<String>>` mit den ent-

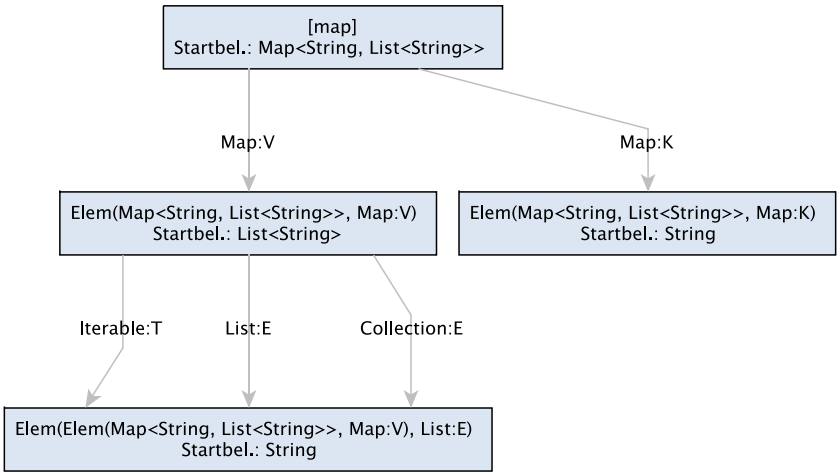

Abbildung 4.5: Constraintvariablen eines parametrisierten Typs

[12] Dies wurde mit einem Profiler (JProfiler: `http://www.ej-technologies.com/products/jprofiler/overview.html`) überprüft.

sprechenden Klassen des JDK aussieht. Die grauen, gerichteten Kanten verbinden die Constraintvariable eines parametrisierten Typ mit der Constraintvariable seines Typarguments. Die Kante ist mit dem Typparameter beschriftet, dessen Belegung sie repräsentiert.

Wird nun ein Constraint erzeugt, bei dem den Constraintvariablen Elementvariablen zugeordnet sind, werden für diese ebenfalls Constraints angelegt. Dazu werden zwischen Elementvariablen, die den gleichen Typparametern zugeordnet sind, Gleichheitsconstraints[13] erstellt. Für die Zuweisung in `Map<String, List<String>> map = new HashMap<String, List<String>>()` ergeben sich somit die Constraints in Abbildung 4.6. Es werden ein Subtypconstraint für den Haupttyp sowie drei Gleichheitsconstraints für die Elementvariablen angelegt. Hier sieht man auch die Notwendigkeit, alle entsprechenden Typparameter der Supertypen zu bestimmen. Auf diese Weise funktioniert die Zuordnung zwischen `HashMap` und `Map`.

Für generische Typen, die als Supertypen verschachtelt parametrisierte Typen deklarieren, muss die Identität von Elementvariablen über mehrere Ebenen festgestellt

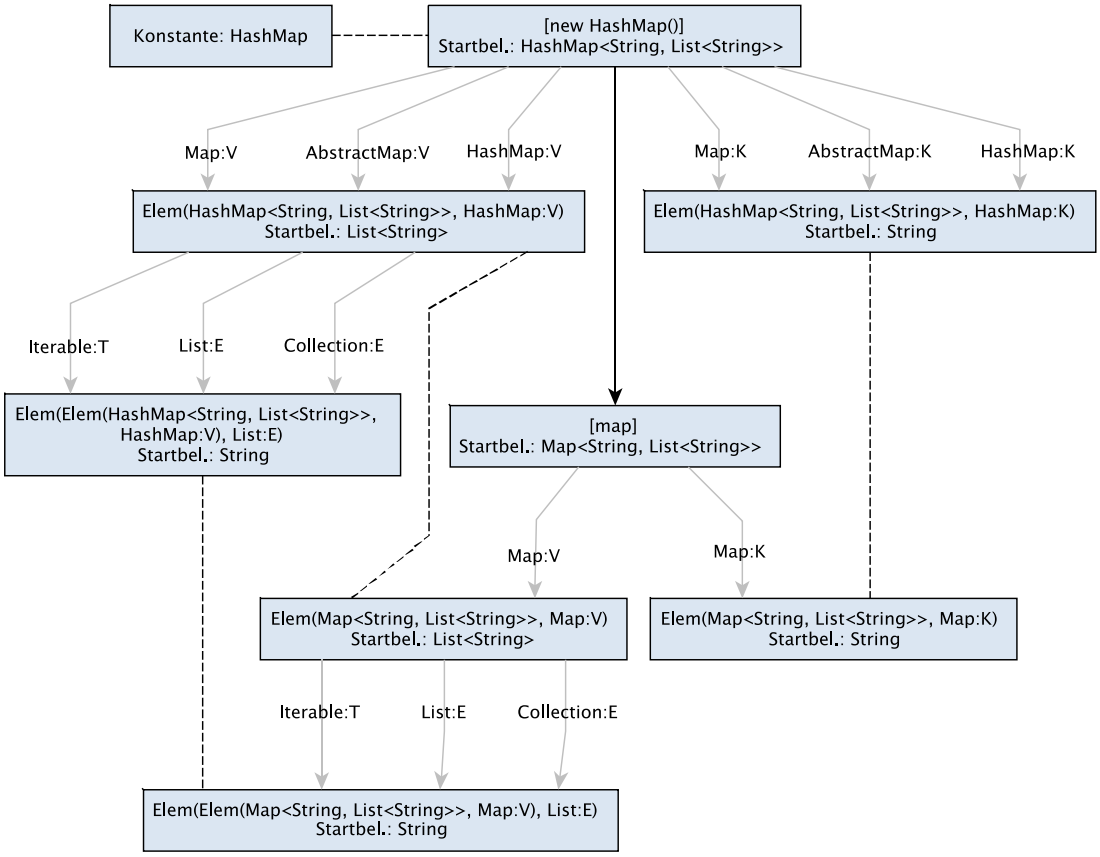

Abbildung 4.6: Constraints für eine Zuweisung zwischen parametrisierten Typen

[13] Im Zusammenhang mit Wildcards können das auch Subtypconstraints sein. Hierzu mehr in Abschnitt 4.2.5.

werden. Für ein Deklarationselement mit Typ `ListOfLists<String>` aus Quelltext 4.3 werden Constraintvariablen und ein zusätzliches Gleichheitsconstraint wie in Abbildung 4.7 erzeugt, damit eine Zuweisung zu einem Deklarationselement mit Typ `List<List<String>>` korrekte Ergebnisse liefert. Dies wird aus Implementierungsgründen der eleganteren Variante vorgezogen, bei der die Zuordnung für den Typparameter von `ListOfLists` direkt zur Elementvariablen in der zweiten Ebene führt.

```
1 class ListOfLists<T> implements List<List<T>> {
2    /* ... */
3 }
4
5 class Client {
6     void client(ListOfLists<String> ls1) {
7         List<List<String>> ls2 = ls1;
8     }
9 }
```

Quelltext 4.3: Beispiel für geschachtelt parametrisierte Supertypen

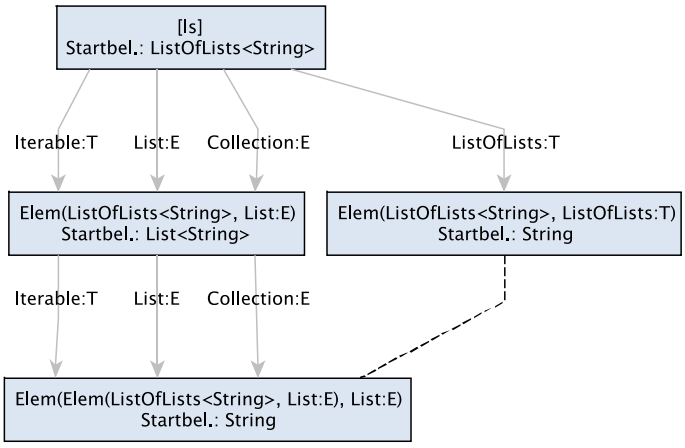

Abbildung 4.7: Constraintvariablen für geschachtelt parametrisierte Supertypen

Ein wesentlicher Punkt bei der Behandlung parametrisierter Typen ist die Frage nach der Identität der Constraintvariablen von Rückgabewerten, Parametern und Feldern. Über die Identität der Constraintvariablen wird festgelegt, welche Constraints miteinander verknüpft sind. Quelltext 4.4 zeigt als Beispiel die Klasse `GenClass<T>`: Für alle möglichen Aufrufe der Methode `addAll` muss der erste Parameter den gleichen Haupttyp haben, dessen Typargument kann sich je nach Empfänger — der ja die Typargumente festlegt — unterscheiden.

Für alle Verwendungen von Typvariablen müssen die Constraintvariablen dafür pro Empfänger einer aufgerufenen Methode eine eigene Identität haben, da ansonsten für

alle Deklarationen mit einer bestimmten Parametrisierung ein einziger neuer Typ berechnet würde. Für alle Parameter oder Rückgabetypen (oder Teile davon), die nicht mit Typvariablen deklariert sind, darf es allerdings genauso wie bei nicht-generischen Klassen nur eine Variable pro Methode geben. Wird ein solcher Rückgabetyp geändert, muss das ja für alle Verwendungen der Methode berücksichtigt werden.

Für „Infer Type“ werden deswegen bei einem Aufruf einer Methode, die zu einer parametrisierten Klasse gehört,[14] zweimal Constraints für Parameter- und Methodentyp erzeugt: zum einen mit unabhängigen (d.h. pro Methodenaufruf neuen) Variablen und den Typen der parametrisierten Form, zum anderen mit einer einzigen Variable pro Methode und den Typen aus der Deklaration der Methode. Für letzteren Fall werden keine Constraints erzeugt, bei denen eine Constraintvariable mit einer Typvariablen belegt wäre. Ist der Parameter selbst mit einer Typvariablen deklariert, wird auf die unabhängige Parametervariable verzichtet und das Argument direkt mit der entsprechenden Elementvariablen des Empfängers verknüpft. Feldzugriffe werden wie Rückgabewerte behandelt.

Abbildung 4.8 zeigt die Constraints als Graph für die Zeilen 5 und 6 von Quelltext 4.4. Hierbei werden die Verbindungen zwischen Variablen und ihren Elementen zusätzlich als graue Kante angezeigt. Man sieht, dass über die Elementvariable der unabhängigen Parametervariable das Typargument von `targetIterable` mit dem deklarierenden Typ von `m1` verbunden ist.

```
1  class Client {
2      public void client(Iterable<Target> targetIterable,
3                         GenClass<Target> gen, String str)
4      {
5          gen.addAll(targetIterable, str);
6          gen.getFirst().m1();
7      }
8  }
9  class GenClass<T> {
10     public void addAll(Iterable<T> t, String str) { /*...*/ }
11     public T getFirst() { /*...*/ }
12 }
13 class Target {
14     public void m1() { /*...*/ }
15     public void m2() { /*...*/ }
16 }
```

Quelltext 4.4: Beispiel für den Aufruf einer Methode einer parametrisierten Klasse

[14] Die umständliche Formulierung dient der Abgrenzung von parametrisierten Methoden als Instanzen generischer Methoden. Hierzu mehr in Abschnitt 4.2.6.

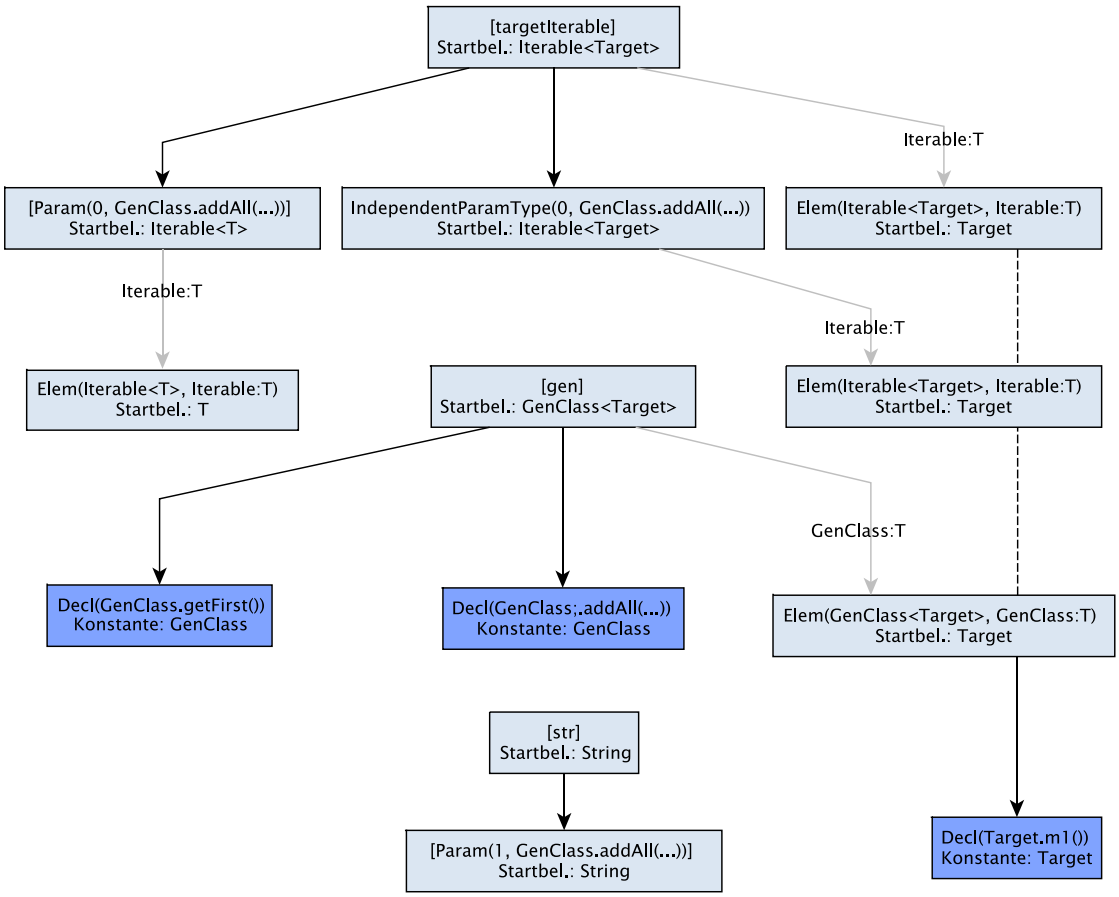

Abbildung 4.8: Constraints für den Aufruf einer Methode einer parametrisierten Klasse

4.2.4 Parametrisierte Typen als Supertypen

Es gibt zwei Parametrisierungsformen von Supertypen, die unterschieden werden müssen: Zum einen kann die Parametrisierung durch Typvariablen erfolgen (wie z. B. in `interface List<E> extends Collection<E>`). Wie im vorhergehenden Abschnitt erläutert, wird in diesem Fall für beide Typparameter auf dieselbe Constraintvariable verwiesen. Zum anderen kann die Parametrisierung durch Klassen oder Interfaces erfolgen (wie z. B. in `class MyComparator implements Comparator<ClassOne>`). Diese Form muss separat behandelt werden und wird im Folgenden beschrieben.

An zwei Stellen müssen zusätzliche Constraints erzeugt werden: Zum einen müssen Constraints für implementierende oder überschreibende Methoden generiert werden. Die genannte Beispielklasse `MyComparator` muss eine Methode `int compare(ClassOne o1, ClassOne o2)` enthalten, die die Methode `int compare(T o1, T o2)` implementiert. Die Typen von `o1` und `o2` und das Typargument von `Comparator<ClassOne>` müssen dabei identisch sein. „Infer Type“ legt deswegen zwischen Typargumenten und De-

klarationselementen von implementierenden und überschreibenden Methoden, die in der Basismethode mit der entsprechenden Typvariablen deklariert sind, Gleichheitsconstraints an.

Zum anderen müssen bei jeder Verwendung von Typen wie `MyComparator` Constraintvariablen und Constraints für die Typargumente ihrer Supertypen erzeugt werden. Diese müssen jedoch zusätzlich über ein Gleichheitsconstraint mit den Typargumenten der Supertypdeklaration in Verbindung gebracht werden. Für das Beispiel aus Quelltext 4.5 (mit `ClassOne` aus Quelltext 3.6) werden so die Constraints in Abbildung 4.9 erzeugt. Man sieht hier, wie `target` über die Elementvariablen von `comp` und `getComparator` mit der zur Supertypdeklaration von `MyComparator` gehörenden Elementvariablen und so mit beiden Parametern von `compare` verbunden ist.

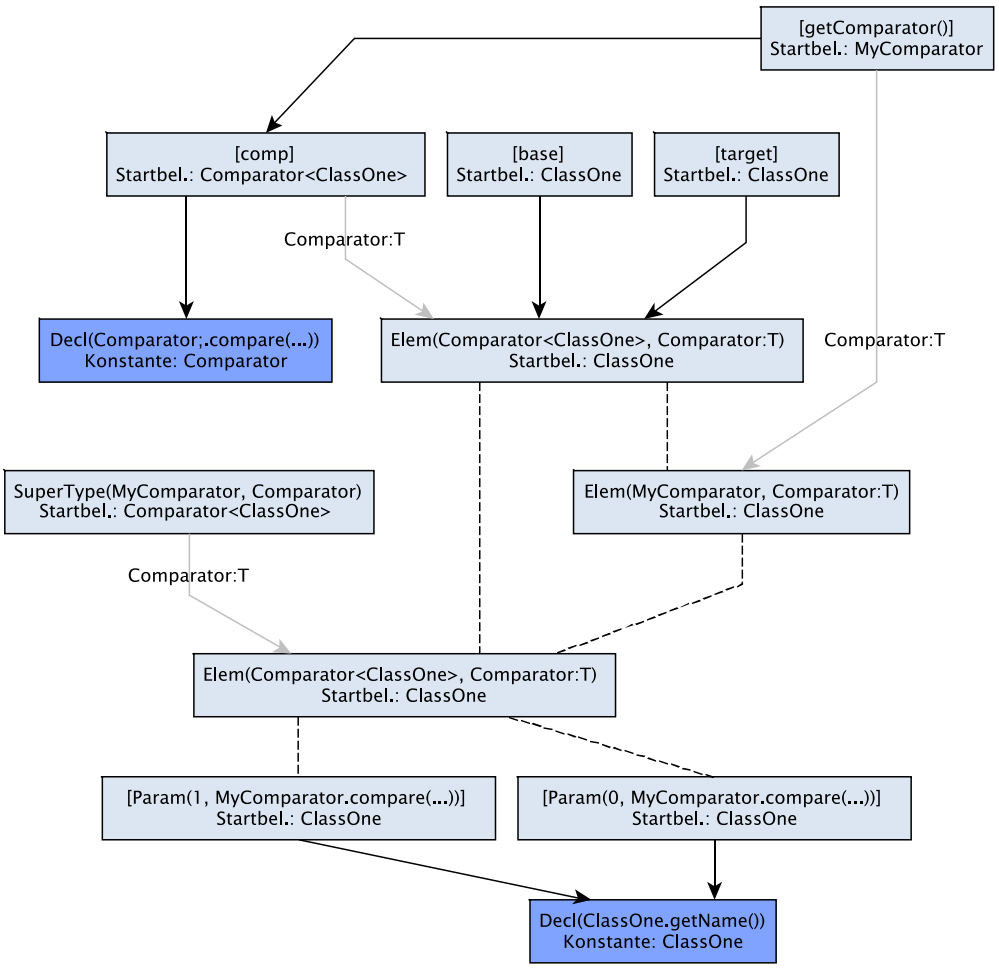

Abbildung 4.9: Constraints für einen parametrisierten Supertyp

```
1  class Client {
2      ClassOne base;
3
4      public boolean check(ClassOne target) {
5          Comparator<ClassOne> comp = getComparator();
6          return comp.compare(base, target) > 0;
7      }
8
9      public Comparator<ClassOne> getComparator() {
10         MyComparator myComp = new MyComparator();
11         return myComp;
12     }
13 }
14 class MyComparator implements Comparator<ClassOne> {
15     public int compare(ClassOne o1, ClassOne o2) {
16         return o1.getName().compareTo(o2.getName());
17     }
18 }
```

Quelltext 4.5: Beispiel für einen parametrisierten Supertyp

4.2.5 Wildcards

Für die bisherigen Beispiele wurden zwischen den Elementvariablen ausschließlich Gleichheitsconstraints angelegt, da im Normalfall zwischen den Typargumenten subtypkompatibler parametrisierter Typen keine Subtyprelation besteht, sondern Typgleichheit erforderlich ist (siehe z. B. [Bracha 2004]). Für viele interessante Anwendungsfälle ist eine Subtyprelation jedoch wichtig. Das Typsystem von Java 5 unterstützt deswegen Wildcards für Typargumente (auch dies wird von Bracha [2004] erläutert). Alle für „Infer Type“ erforderlichen Zusammenhänge lassen sich dabei durch Anpassung des jeweiligen Constrainttyps der bisher dargestellten Constraints zwischen Elementvariablen erfassen.

Im Folgenden wird erläutert, welche Constraints bei einer Zuweisung $E_1 = E_2$ mit Wildcards erstellt werden. Dabei ist *C<T>* eine generischen Klasse, $[E_1]$ = *C<M>* und $[E_2]$ = *C<N>,* wobei *M* ein Wildcardtyp und *N* beliebig typkompatibel ist. Zur besseren Übersichtlichkeit wird der Haupttyp *C* auf beiden Seiten der Zuweisung identisch gewählt, die Ausführungen sind aber ebenso gültig, wenn der Haupttyp auf der rechten Seite ein Subtyp des Haupttyps auf der linken Seite ist. Die Behandlung ist auch genauso gültig für alle anderen Sprachkonstrukte, bei denen zwischen den Haupttypen Subtypconstraints bestehen (wie zum Beispiel zwischen Parametern und Argumenten eines Methodenaufrufs). Es wird zwischen drei Formen von Wildcards unterschieden:

Wildcards mit oberer Schranke wie zum Beispiel in `List<? extends JComponent>`.[15]

[15] Die Beispiele beziehen sich auf die entsprechenden Klassen aus den Paketen `javax.swing` und `java.util`.

Hier wird zwischen $Elem(E_1, T)$ und $Elem(E_2, T)$ das Subtypconstraint $Elem(E_2, T) \leq Elem(E_1, T)$ angelegt. Steht auf der rechten Seite also zum Beispiel ein Ausdruck von Typ `List<JTextField>`, wird damit gefordert, dass `JTextField` ein Subtyp von oder identisch mit `JComponent` sein muss. Die Bedingung erfasst ebenso korrekt Zuweisungen zwischen zwei Elementen mit Wildcardtypen. So muss für eine Zuweisung mit rechter Seite vom Typ `List<? extends JTextComponent>` für die Typkorrektheit ebenso nur `JTextComponent` $\leq$ `JComponent` gelten.

Wildcards mit unterer Schranke wie zum Beispiel in `List<? super JTextField>`. Hier wird zwischen $Elem(E_1, T)$ und $Elem(E_2, T)$ der spiegelbildliche Subtypconstraint $Elem(E_1, T) \leq Elem(E_2, T)$ angelegt. Steht auf der rechten Seite also zum Beispiel ein Ausdruck von Typ `List<JComponent>`, wird damit gefordert, dass `JTextField` ein Subtyp von oder identisch mit `JComponent` sein muss. Auch dieser Constraint erfasst ebenso korrekt Zuweisungen zwischen zwei Elementen mit Wildcardtypen. So muss für eine Zuweisung mit rechter Seite vom Typ `List<? super JTextComponent>` für die Typkorrektheit ebenso nur `JTextField` $\leq$ `JTextComponent` gelten.

Unbeschränkte Wildcards wie zum Beispiel in `List<?>` stellen im Wesentlichen eine Abkürzung für `? extends Object` dar. Hier müssen keine Constraints erstellt werden, da jeder Typ N die Typisierungsregeln erfüllt.

Quelltext 4.6 zeigt ein Beispiel für die Anwendung der beiden beschränkten Versionen. Hier wird zusätzlich auf die Identität des Haupttyps verzichtet. Abbildung 4.10 zeigt die dazu erzeugten Constraints.

```
1  public class Client {
2      Comparator<JComponent> componentComparator;
3
4      public void client(List<JTextField> textFields) {
5          addSortedTextComponents(textFields, componentComparator);
6      }
7      public void addSortedTextComponents(
8              Collection<? extends JTextComponent> textComponents,
9              Comparator<? super JTextComponent> comparator) { /*...*/ }
10 }
```

Quelltext 4.6: Verwendung von Wildcards

Wildcards können auch innerhalb von verschachtelt parametrisierten Typen auftreten. In diesem Fall hängt die Behandlung davon ab, ob der übergeordnete Constraint ein Subtyp- oder ein Gleichheitsconstraint ist. Für einen Typ `Map<String, List<? extends JComponent>>` muss bei einer Zuweisung für die mit `? extends`

`JComponent` belegte Elementvariable ein Gleichheitsconstraint erstellt werden. Für einen Typ `Map<String, ? extends List<? extends JComponent>>` wird auch für die mit `? extends JComponent` belegte Elementvariable ein Subtypconstraint erzeugt.

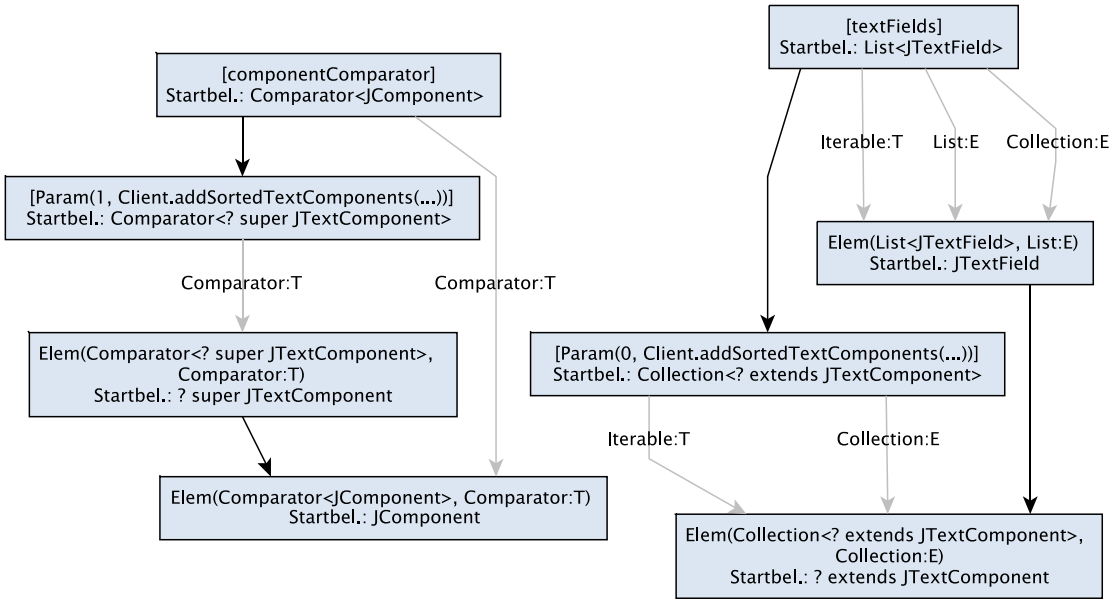

Abbildung 4.10: Constraints für Wildcardtypen

4.2.6 Generische Methoden

In Java 5 kann man nicht nur generische Klassen, sondern auch generische Methoden deklarieren. Für diese Methoden werden genauso wie für generische Klassen Typparameter angegeben. Für diese gibt es jedoch bei der Verwendung keine explizite Belegung mit einen Typargument, vielmehr inferiert der Compiler für alle Typparameter der Methode einen möglichst spezifischen Typ. Es müssen hier also zwar keine Typargumente geändert werden, jedoch führt die Typänderung einer Referenz, die als Argument einer generischen Methode verwendet wird, unter Umständen zu anderen vom Compiler für die Typparameter inferierten Typen. Dies führt in der Regel dazu, dass andere Argumente der Methode oder Deklarationselemente, denen der Rückgabewert der Methode zugewiesen wird, für ein typkorrektes Ergebnis ebenfalls geändert werden müssen.

Quelltext 4.7 (mit `ClassOne` aus Quelltext 3.6) zeigt eine einfache generische Methode und deren Verwendung. Hierbei werden der Parameter und die Methode selbst mit der Typvariablen deklariert. Um dies zu verknüpfen, werden für jede Verwendung einer generischen Methode eigene Constraintvariablen für alle Typparameter erzeugt und mit Methodenargumenten und -rückgabe verbunden. Für das Beispiel ergeben sich so die Constraints aus Abbildung 4.11.

```
class Client {
    public void client(ClassOne obj) {
        obj.m1();
        ClassOne genericResult = passThrough(obj);
        genericResult.m2();
    }
    public <T> T passThrough(T t) {
        return t;
    }
}
```

Quelltext 4.7: Einfaches Beispiel einer generischen Methode

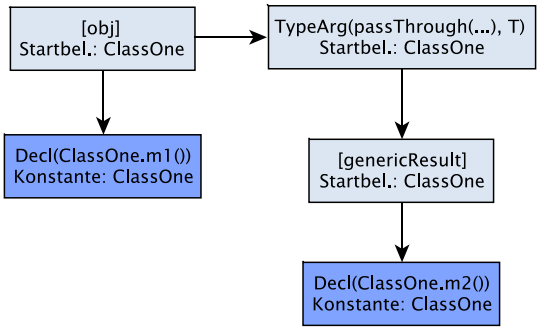

Abbildung 4.11: Constraints für den Aufruf einer generischen Methode

Werden parametrisierte Typen wie `List<T>` als Methodenparameter oder -rückgabetyp verwendet, müssen für jeden Aufruf — wie in Abschnitt 4.2.3 für generische Klassen beschrieben — zusätzlich eigene Parameter- und Rückgabeconstraintvariablen erzeugt werden, deren Elementvariablen mit den pro Methodenaufruf erstellten Constraintvariablen für die Typargumente verbunden werden. Durch die Art und Anordnung der Constraints können wie in Abschnitt 4.2.5 beschrieben Wildcards korrekt repräsentiert werden. In Quelltext 4.8 wird der Aufruf der generischen Methode `public static <T> void fill(List<? super T> list, T obj)` aus der Klasse `java.util.Collections` gezeigt.

```
class Client {
    List<ClassOne> ls;
    public void replaceAll(ClassOne obj) {
        Collections.fill(ls, obj);
    }
}
```

Quelltext 4.8: Aufruf einer generischen Methode mit parametrisierten Parametertypen

Abbildung 4.12 zeigt die dazu erstellten Constraints. Es werden zwei Constraintvariablen pro Parameter verwendet: Die eine dient der eindeutigen Zuordnung des Haupttyps

`List`, hierfür gibt es für das ganze Programm nur eine einzige Variable. Die andere dient der Zuordnung der Elementvariable von `List` zu der pro Methodenaufruf erstellten Typargumentvariable. Aus den Constraints ergibt sich so, dass der Typ der Variable `obj` ein Subtyp des Typarguments von `ls` sein muss.

Generische Methoden können auch als Instanzmethoden von generischen Klassen deklariert werden. In diesem Fall muss bei der Constraintgenerierung zusätzlich auf die richtige Zuordnung der Typvariablen geachtet werden.

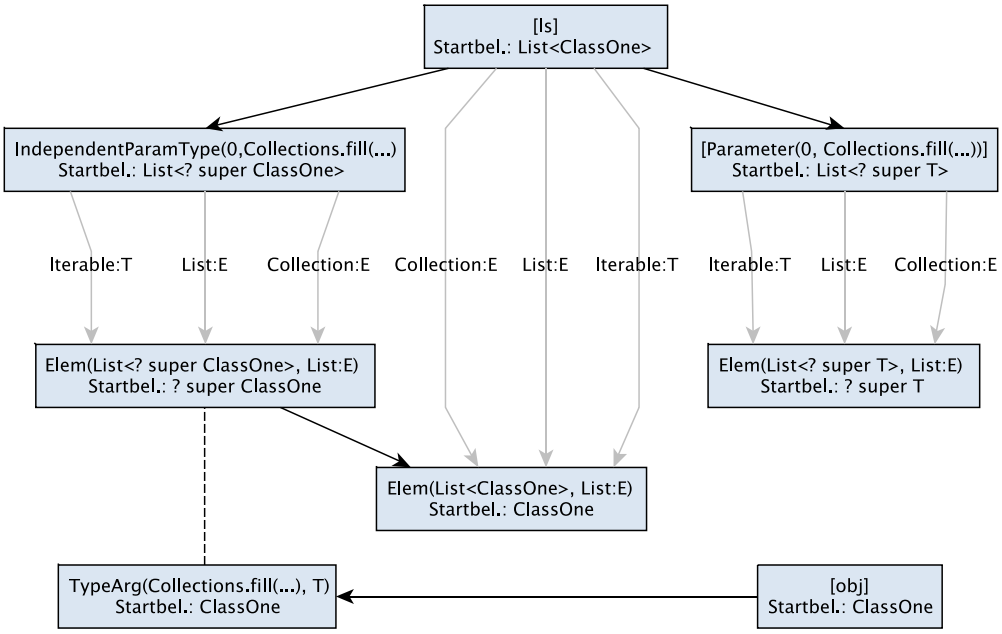

Abbildung 4.12: Constraints für den Aufruf von `Collections.fill()`

4.2.7 Schranken von Typparametern

Sowohl Typparameter von generischen Klassen als auch von generischen Methoden können Schranken haben. Diese werden in der Form T `extends` C & I_1 ... I_n für den Typparameter angegeben (siehe auch [Gosling et al. 2005, Abschnitt 4.4]). Für die Typvariable T können dann alle sichtbaren Methoden und Felder von C sowie von I_1 bis I_n verwendet werden. Man kann einer Typvariable mit Schranken also in der Regel keinen einzelnen Typ zuordnen, die bisher dargestellte Modellierung müsste dafür erweitert werden. Schranken von Typparametern werden deswegen von „Infer Type“ nicht so in Constraints gefasst, dass diese geändert werden können. Nichtsdestotrotz muss die Typkorrektheit des Ergebnisses sichergestellt werden — es werden dafür Constraints erstellt, bei denen die entsprechenden Constraintvariablen als konstant markiert werden.

„Infer Type“ legt für alle entsprechenden Typargumente $Elem(E, T)$ von generischen

Klassen folgende Subtypconstraints an: $Elem(E,T) \leq C$ und $Elem(E,T) \leq I_1$ bis $Elem(E,T) \leq I_n$. Für generische Methoden steht auf der linken Seite dieser Constraints die Constraintvariable aus Abschnitt 4.2.6, die für das vom Compiler inferierte Typargument angelegt wird.

Schranken von Typparametern können allerdings auch andere Typvariablen enthalten. Quelltext 4.9 (mit `ClassOne` aus Quelltext 3.6) zeigt ein Beispiel, in dem eine Typvariable als Typargument einer Schranke verwendet wird:

```
1  class Client {
2      SpecialContainer<ClassOne, HashSet<ClassOne>> specialContainer;
3      public void client(ClassOne obj) {
4          specialContainer.add(obj);
5      }
6  }
7  class SpecialContainer<E, C extends Collection<E>> {
8      C store;
9      public void add(E e) {
10         store.add(e);
11     }
12 }
```

Quelltext 4.9: Beispiel für Schranken von Typparametern

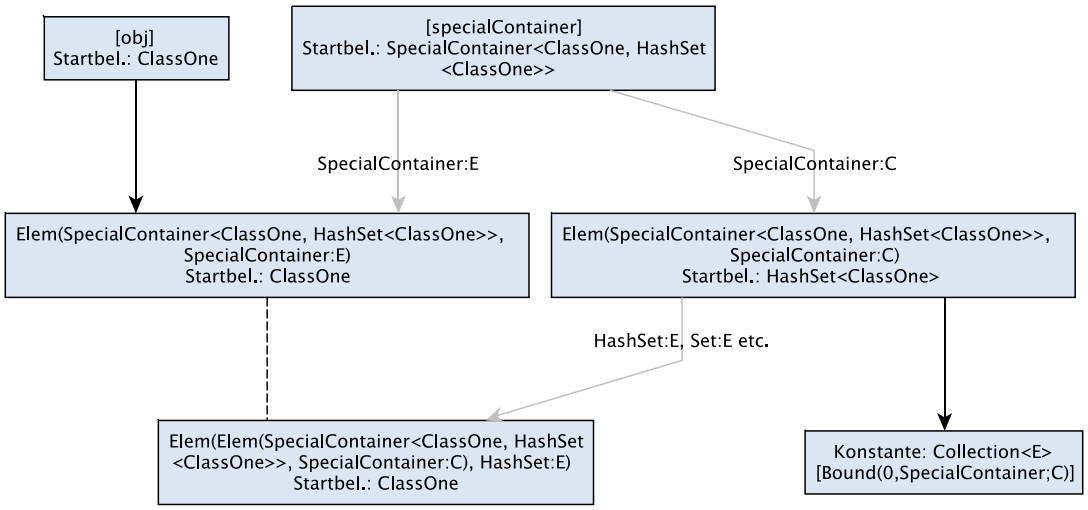

Abbildung 4.13: Constraints für Schranken von Typparametern

In diesem Fall müssen für alle Vorkommen des Typs oder der Methode beide Typargumente entsprechend miteinander verbunden werden. Abbildung 4.13 zeigt dies für Quelltext 4.9. Alle Typargumente zu Typparameter `C` müssen so ein Subtyp von `Collection` sein. Typargumente zu Typparameter `E` können jedoch unbeschränkt geändert werden, solange das Typargument von `HashSet` ebenso abgeändert wird.

4.2.8 Weitere neue Sprachkonstrukte in Java 5

Java 5 hat als spezielle Klassen Aufzählungstypen, sog. Enums, und als spezielle Interfaces die Deklaration von Annotations eingeführt. Für beide Fälle gibt es in „Infer Type“ keine spezielle Unterstützung, da Annotations keine expliziten Superklassen deklarieren können und Enums nicht besonders sinnvoll durch Einführung eines Interfaces generalisiert werden können. Da „Infer Type“ zudem nicht auf primitive oder externe Typen (d. h. Typen, deren Quelltext nicht modifiziert werden kann) angewendet werden kann, muss auch Autoboxing nicht explizit behandelt werden. Varargs können wie vom Compiler als Array behandelt werden, so dass hierfür — abgesehen von einigen technischen Details in der Implementierung — keine spezielle Behandlung erforderlich ist.

Zudem hat Java 5 mit der erweiterten For-Schleife eine neue Kontrollstruktur eingeführt, die stark mit Generics zusammenhängt und eine spezielle Behandlung von „Infer Type“ erfordert. Weiterhin gibt es einen Spezialfall beim Wildcard der Methode `Object.getClass()` (siehe [Gosling et al. 2005, Abschnitt 4.3.2]). Die Behandlung der beiden Konstrukte wird im Folgenden vorgestellt.

Die erweiterte For-Schleife

Die erweiterte For-Schleife erleichtert das Iterieren über Arrays und Ausdrücke vom Typ `Iterable`. Im ersten Fall muss die im Parameterteil der erweiterten For-Schleife deklarierte lokale Variable den Typ der Arrayelemente oder einen Supertyp davon haben. Im zweiten Fall bezieht der Compiler den Typ der Elemente über das generische Interface `java.lang.Iterable` (hier kann die Variable ebenso mit einem Supertyp deklariert werden). Dies kann jeweils durch ein Subtypconstraint dargestellt werden. Quelltext 4.10 (mit `ClassOne` aus Quelltext 3.6) zeigt ein Beispiel für den Fall mit Interface `Iterable`:

```
1 public class Client {
2     public void printAllKeyNames(Map<ClassOne, Object> map) {
3         for (ClassOne obj : map.keySet()) {
4             System.out.println(obj.getName());
5         }
6     }
7 }
```

Quelltext 4.10: Beispiel für die erweiterte For-Schleife

In Abbildung 4.14 sind die erstellten Constraints dargestellt. Man sieht hier, dass das Typargument von `Set` ein Subtyp des Typs von `obj` sein muss. Zudem wird durch die Subtypbedingung zwischen dem Typ von `keySet` und der Konstante `Iterable` sichergestellt, dass der Ausdruck in der erweiterten For-Schleife typkorrekt ist. Man sieht an dem Beispiel auch, wie die in Abschnitt 4.2.3 erläuterten unabhängigen Variablen für den Methodentyp verwendet werden.

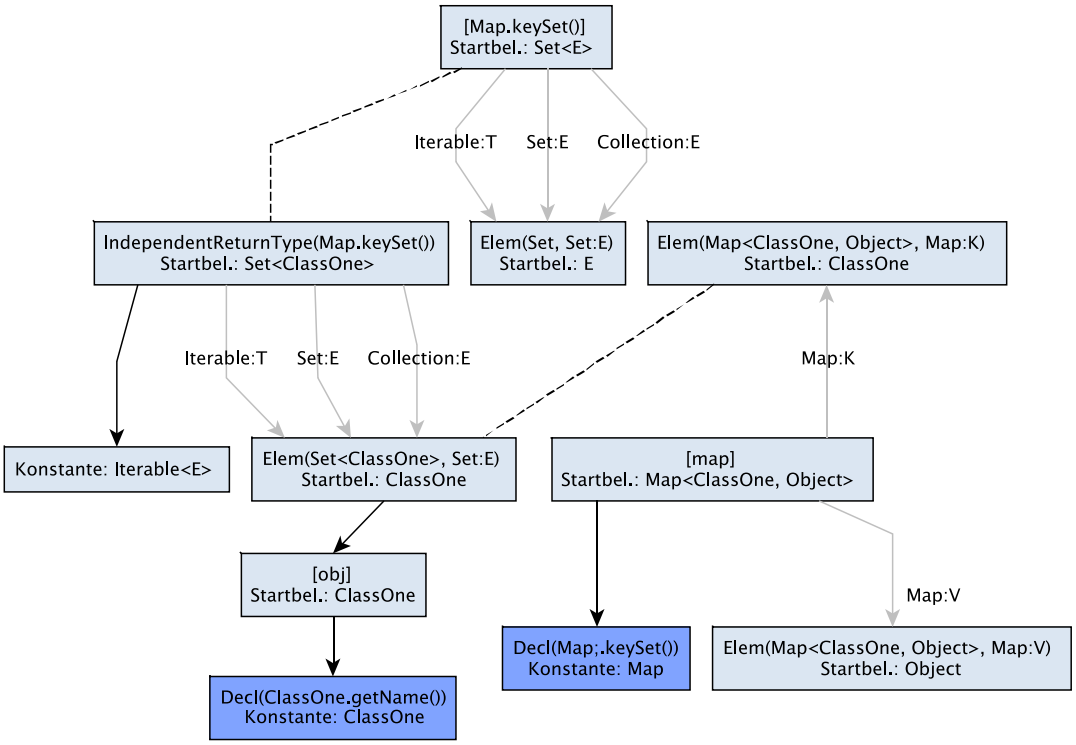

Abbildung 4.14: Constraints für die erweiterte For-Schleife

Wildcardbehandlung von `Object.getClass()`

Für die Wildcard des als `Class<?>` deklarierten Rückgabetyps von `Object.getClass()` sieht [Gosling et al. 2005, Abschnitt 4.3.2] eine spezielle Behandlung vor. Der vom Compiler tatsächlich verwendete Typ ist `Class<? extends` *T*`>`, wobei *T* dem (statischen) Typ des Empfängers entspricht. „Infer Type“ legt hierzu ein Subtypconstraint zwischen dem Typargument des Rückgabetyps und dem Typ des Empfängers an.

Quelltext 4.11 (mit `ClassOne` aus Quelltext 3.6) zeigt ein Anwendungsbeispiel des Spezialfalls im Zusammenhang mit einer generischen Methode. In Abbildung 4.15 sind die dazu erzeugten Constraints dargestellt. Für den Spezialfall wird die Subtypbeziehung zwischen `classOne` und der zum Rückgabetyp der Methode `Object.getClass()` gehörenden Elementvariable erstellt. Diese führt im Beispiel dazu, dass `classOne` (transitiv) ein Subtyp des deklarierenden Typs von `m1()` sein muss.

```
1  public class Client {
2      public void client(ClassOne classOne) throws Exception {
3          ClassOne otherInstance = createAndLog(classOne.getClass());
4          otherInstance.m1();
5      }
6      public <T> T createAndLog(Class<? extends T> cl) throws Exception {
7          T obj = cl.newInstance();
8          System.out.println(obj);
9          return obj;
10     }
11 }
```

Quelltext 4.11: Beispiel für die Verwendung von `Object.getClass()`

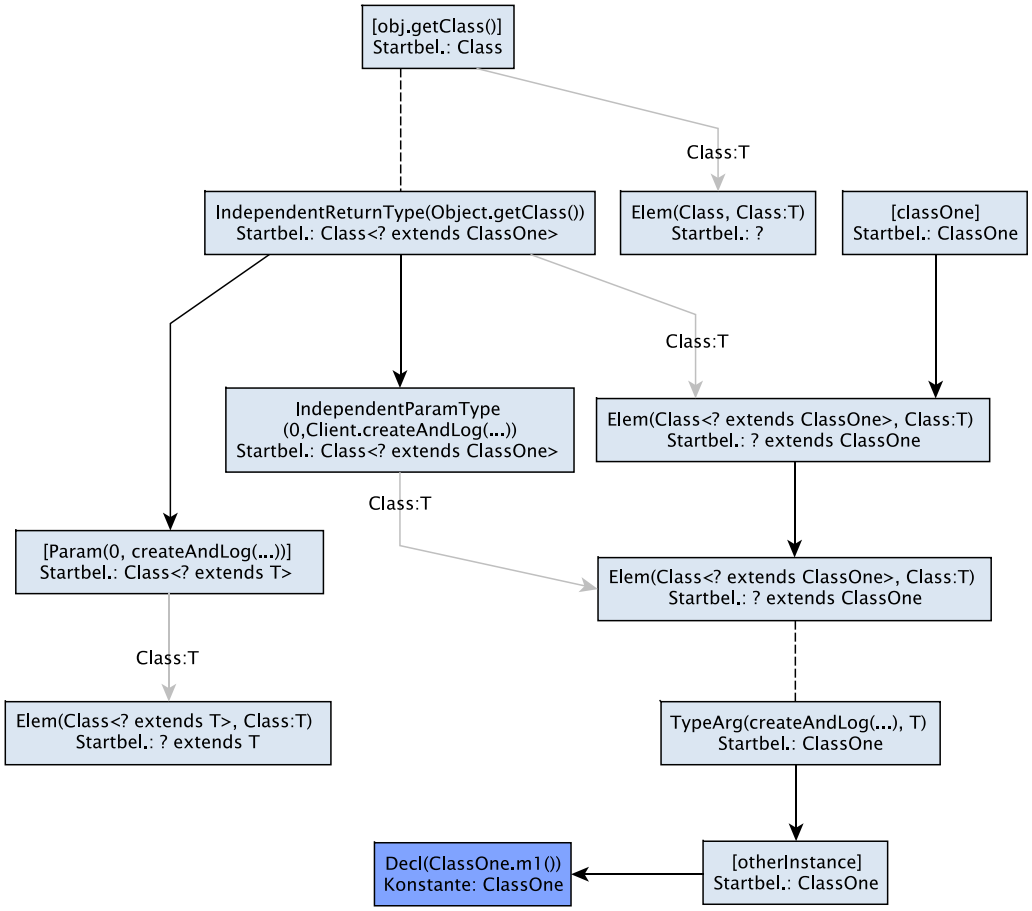

Abbildung 4.15: Constraints für die Verwendung von `Object.getClass()`

4.3 Lösung der Constraints

Nachdem „Infer Type“ alle Constraints wie im vorangehenden Abschnitt beschrieben erstellt hat, muss es die gewünschte Lösung[16] des Constraintsystems bestimmen. Dies erfordert ein zweistufiges Vorgehen: Zunächst wird das benötigte Protokoll [Steimann 2007] für das ausgewählte Deklarationselement berechnet. Dies wird in Abschnitt 4.3.1 und 4.3.2 dargestellt. Anschließend wird entweder ein existierender Typ mit genau diesem Protokoll oder ein neu zu erzeugender Typ für die Auswahl verwendet. Dies würde in der Regel dazu führen, dass Constraints nicht erfüllt sind. Deswegen muss im zweiten Schritt dafür gesorgt werden, dass alle Constraints erfüllt werden. Abschnitt 4.3.3 erläutert, wie „Infer Type“ hierzu vorgeht.

4.3.1 Definition des benötigten Protokolls

Ein Typ T deklariert eine Menge von Methoden und Feldern, die von Steimann [2007] als $\mu(T)$ bezeichnet wird. Als Protokoll eines Typs bezeichnet Steimann [2007] die Menge der öffentlichen, nicht-statischen Methoden eines Typs und definiert dies als:

$$\pi(T) := \{m \in \mu(T) \mid m \text{ ist öffentliche, nicht-statische Methode}\}$$

Weiterhin gibt es nach Steimann [2007] für jeden Ausdruck neben dem deklarierten Typ einen inferierten Typ. Für eine Referenz a mit deklariertem Typ A wird dabei der Typ I als inferierter Typ bezeichnet, der die kleinstmögliche Menge an Methoden und Feldern beinhaltet, auf die von a zugegriffen wird, vereinigt mit den entsprechenden Mengen aller Referenzen, denen a direkt oder indirekt zugewiesen wird. Dafür wird $\iota(a) := \mu(I)$ als Funktion von a mit I wie beschrieben definiert. $\iota(a)$ kann also Felder und nicht-öffentliche oder statische Methoden enthalten. Eine Entkopplung durch Interfaces ist dann nicht möglich, deswegen wird dieser Fall im Folgenden nicht betrachtet. Für die übrigen Fälle ist also $\iota(a) \subseteq \pi(A)$ das gesuchte benötigte Protokoll (oder auch Access set) eines Deklarationselements a.

Ziel von „Infer Type“ ist es, diesen Typ I für das ausgewählte Deklarationselement einzusetzen, so dass das transformierte Programm wieder typkorrekt ist. Wie in Abschnitt 3.3.1 beschrieben, reicht dafür die Analyse der direkten und indirekten Zuweisungen nicht mehr aus. Quelltext 4.12 (mit `ClassOne` aus Quelltext 3.6) zeigt ein Minimalbeispiel, in dem bewusst keine Zuweisung zwischen Eingabeparameter von `add` und Rückgabewert von `get` besteht.

Die Referenz `obj` wird nur dem Parameter von `add(E e)` zugewiesen. Auf beiden Referenzen wird keine Methode aufgerufen. Nach bisheriger Definition gilt $\iota(\texttt{obj}) = \emptyset$.

[16] Grundsätzlich gibt es viele Lösungen eines Constraintsystems. Insbesondere ist die Startbelegung eine solche, da das Ausgangsprogramm typkorrekt ist.

```
1  class Client {
2      NopContainer<ClassOne> nopContainer;
3      public void client(ClassOne obj) {
4          nopContainer.add(obj);
5          ClassOne other = nopContainer.get();
6          other.m1();
7      }
8  }
9  class NopContainer<E> {
10     public void add(E e) { }
11     public E get() { return null; }
12 }
```

Quelltext 4.12: Beispiel für Abhängigkeit ohne Zuweisung

Würde man nun aber den Typ von `obj` zu `Object` ändern, müsste das Typargument von `nopContainer` ebenfalls `Object` sein. Ohne einen Cast zu verwenden — was nicht gewünscht ist — müsste `other` ebenfalls vom Typ `Object` sein. Auf diesem wird jedoch die Methode `m1()` der Klasse `ClassOne` aufgerufen.

Im Rahmen dieser Arbeit wurde die Definition des inferierten Typs deswegen etwas abgewandelt. Für eine Referenz a wird der Typ I als inferierter Typ bezeichnet, der die kleinstmögliche Menge an Methoden und Feldern beinhaltet, auf die von a zugegriffen wird, vereinigt mit den entsprechenden Mengen aller Referenzen b, für die (direkt oder indirekt) ein Subtypconstraint $[a] \leq [b]$ existiert. Ein Gleichheitsconstraint $[a] = [b]$ wird dabei in Einklang mit der Antisymmetrie von $\leq$ als $[a] \leq [b]$ und $[b] \leq [a]$ aufgefasst. Für Java-Programme ohne Verwendung von Generics führen beide Definitionen zu demselben Ergebnis.

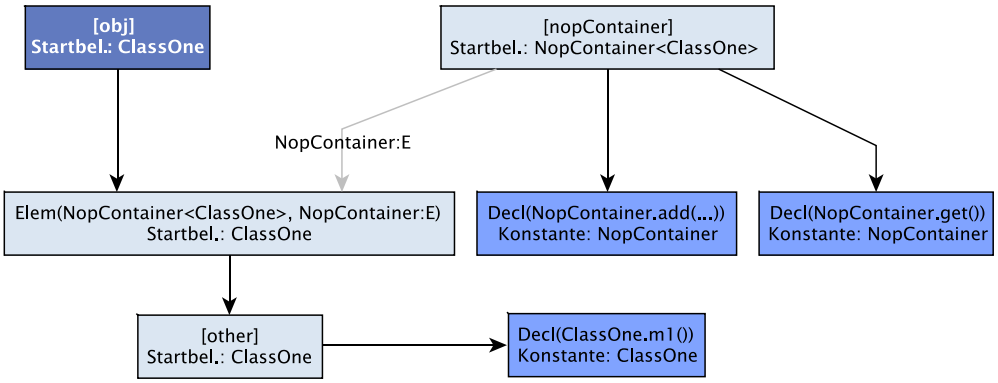

Abbildung 4.16: Constraints zu Quelltext 4.12

In Abbildung 4.16 sind die nach Abschnitt 4.2 für Quelltext 4.12 erzeugten Constraints dargestellt. Man sieht, dass — über das Typargument von `nopContainer`

— eine Subtypbeziehung zwischen `obj` und `other` besteht. Der Zugriff auf die Methode `ClassOne.m1()` wird durch den Constraint [`other`] ≤ Decl(`ClassOne.m1()`) repräsentiert. Für die erweiterte Definition des inferierten Typs gilt also ι(`obj`) = {`ClassOne.m1()`}. Ein Supertyp von `ClassOne`, der diese Methode deklariert, kann typkorrekt für `obj` verwendet werden, wenn das Typargument von `nopContainer` und der Typ von `other` ebenso angepasst werden.

4.3.2 Algorithmus zur Bestimmung des benötigten Protokolls

Legt man der Subtyprelation ≤ aus Abschnitt 4.2 die Teilmengenrelation ⊇ auf Mengen von öffentlichen Methoden[17] zu Grunde, kann man die Suche nach dem benötigten Protokoll als Suche nach einer minimalen Lösung ähnlich wie Palsberg & Schwartzbach [1994] formulieren. Eine Constraintvariable wird dabei mit einer Menge von Methoden belegt. Die Constraintrelation = entspricht der Mengengleichheit, die Constraintrelation ≤ der Teilmengenrelation ⊇. Die Variablen der Form $Decl(M)$, für die M eine öffentliche und nicht-statische Methode ist, werden konstant mit $\{M\}$ belegt.

Werden die Constraintvariablen mit dem Protokoll ihres deklarierten Typs belegt, ergibt sich somit eine Lösung des Constraintsystems — ansonsten wären die erstellten Constraints nicht korrekt, da ja von einem typkorrekten Ausgangsprogramm ausgegangen wird. Eine Belegung ist dann eine Lösung, wenn für alle Constraintvariablen gilt, dass $\iota(E)$ (mit E als zugehöriger Ausdruck) eine Teilmenge des Werts der jeweiligen Constraintvariable ist. Gesucht ist die Lösung, bei der alle Constraintvariablen genau mit $\iota(E)$ belegt sind.

Da für „Infer Type“ ein einzelnes Deklarationselement ausgewählt wird, kann man die Minimierung auf den relevanten Teil der Constraints beschränken.[18] Die Bestimmung des benötigten Protokolls wird deswegen als Tiefendurchlauf — auch als Tiefensuche bezeichnet — auf dem gerichteten Constraintgraph durchgeführt. Hierbei muss nur die Komponente des Graphen durchlaufen werden, die die zum ausgewählten Deklarationselement gehörende Constraintvariable enthält. Diese Variable wird dabei als Wurzel des Spannbaums ausgewählt.

Eine Beschreibung der Tiefensuche auf gerichteten Graphen findet sich z. B. in [Turau 2004] und [Brandstädt 1992], auf eine vollständige Darstellung wird hier verzichtet. Die Nummerierung der Kanten in Abbildung 4.17 entspricht einer möglichen Bearbeitungsreihenfolge bei der Tiefensuche. Die Suche teilt dabei die Kanten des Graphen (dies sind in diesem Fall die Constraints) in zwei Mengen auf: Baumkanten und Nicht-Baumkanten.

[17] Im Allgemeinen müsste man Methoden und Felder betrachten. Da für Felder jedoch — wie im vorhergehenden Abschnitt ausgeführt — keine Entkopplung durch Interfaces durchgeführt werden kann, kann die Minimierung auf Methoden beschränkt werden. In Abschnitt 4.3.5 wird erläutert, wie die Verwendung von Feldern erkannt wird.

[18] Warum auch nicht-relevante Constraints erstellt werden, wird in Abschnitt 7.2 besprochen.

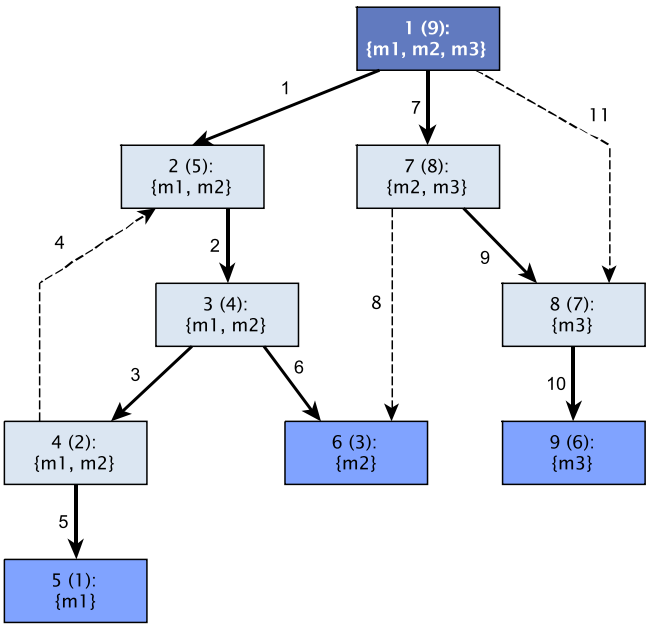

Abbildung 4.17: Beispielgraph für die Bestimmung des benötigten Protokolls

Die Reihenfolge, in der die Ausgangskanten eines Knoten ausgewählt werden, ist hierbei nicht festgelegt (es hätte z. B. auch Kante 7 vor Kante 1 gewählt werden können). Ändert man die Reihenfolge, ergibt sich eine andere Aufteilung der Kanten.

Die Baumkanten sind für die gewählte Reihenfolge in Abbildung 4.17 fett dargestellt. Dadurch ist auch die Reihenfolge der Abarbeitung der Knoten (Constraintvariablen) festgelegt. Für „Infer Type“ werden bei der Suche sowohl vorgeordnet (pre order) als auch nachgeordnet (post order) Aktionen für den Knoten durchgeführt. Abbildung 4.17 enthält deswegen zwei Knotennummerierungen. Die erste Nummer gibt die Reihenfolge an, in der mit der Bearbeitung der Knoten begonnen wird. Diese wird im Folgenden auch für die Bezeichnung der Knoten verwendet. Die zweite Nummer (in Klammern) gibt die Reihenfolge an, in der die Bearbeitung abgeschlossen wird.

Die Nicht-Baumkanten — d. h. die Kanten, die zu einem bereits besuchten Knoten führen — unterteilen sich wiederum in drei Gruppen [Turau 2004]. Eine Kante, die von Knoten e zu einem schon besuchten Knoten f führt, heißt:

Rückwärtskante, wenn f im Spannbaum[19] ein Vorgänger von e ist. In Abbildung 4.17 ist Kante 4 eine Rückwärtskante.

Vorwärtskante, wenn f im Spannbaum ein Nachfolger von e ist. In Abbildung 4.17 ist Kante 11 eine Vorwärtskante.

[19] Im Allgemeinen handelt es sich um einen Tiefensuchwald. Da jedoch nur eine zusammenhängende Komponente des Graphen ausgewertet wird, gibt es auch nur einen Baum.

Querkante, wenn f im Spannbaum weder Vorgänger noch Nachfolger von e ist. In Abbildung 4.17 ist Kante 8 eine Querkante.

Sobald mit der Bearbeitung eines Knoten begonnen wird, wird diesem ein Protokollobjekt zugeordnet — dieses speichert eine Menge von Methoden und stellt damit die Belegung einer Constraintvariable dar. Die Knoten 5, 6 und 7 entsprechen Variablen des Typs $Decl(M)$. Ist M in `java.lang.Object` deklariert, wird diesen konstant die leere Menge zugeordnet,[20] ansonsten werden sie konstant mit der Menge $\{M\}$ belegt. Allen anderen Knoten wird zunächst eine leere Menge zugeordnet. Die verschiedenen Kantentypen erfordern dann eine unterschiedliche Behandlung:

- Für Baumkanten wird nach Bearbeitung des Kindknotens dessen Methodenmenge dem Vaterknoten hinzugefügt.
- Die Bearbeitung des Zielknotens von Querkanten ist bereits abgeschlossen. Dessen Methoden können also direkt hinzugefügt werden.
- Für Vorwärtskanten müssen keine Methoden hinzugefügt werden, da diese auch über den Pfad innerhalb des Baumes erreicht werden. Um keine Unterscheidung zwischen Vorwärts- und Querkanten treffen zu müssen, können diese aber auch genauso wie Querkanten behandelt werden.
- Für Rückwärtskanten können die Methoden des Zielknotens noch nicht hinzugefügt werden, da dessen Bearbeitung noch nicht abgeschlossen ist. Deren spezielle Behandlung wird im Folgenden besprochen.

Eine Rückwärtskante von e nach f induziert einen Kreis, der aus der Kante selbst und dem Pfad von f nach e besteht. Innerhalb eines solchen Kreises müssen die Typen der Ausdrücke und deren benötigte Protokolle identisch sein. Um in einem Durchlauf korrekte benötigte Protokolle für alle besuchten Constraintvariablen berechnen zu können, ordnet „Infer Type“ allen Knoten eines Kreises ein gemeinsames Protokollobjekt zu, das die bisher bestimmten Methoden aller Knoten enthält. Das Protokollobjekt speichert zudem die Knoten, denen es zugeordnet ist. Befindet sich ein Knoten in einem weiteren Kreis, kann so allen beteiligten Knoten ein neues gemeinsames Protokollobjekt zugeordnet werden.

In Abbildung 4.17 zeigen also die Knoten 2, 3 und 4 nach Bearbeitung der Kante 4 auf dasselbe Protokollobjekt. Durch Bearbeitung von Knoten 3 wird so automatisch die Methode `m2` auch für das benötigte Protokoll von Knoten 4 bestimmt, obwohl dessen Bearbeitung bereits abgeschlossen ist. Bestünde noch eine weitere Rückwärtskante von

[20] Diese Methoden spielen für das benötigte Protokoll im Rahmen des „Infer Type“-Refactorings keine Rolle, da sie für jeden Typ verfügbar sind.

Knoten 5 zu Knoten 3, so würde durch die Speicherung der beteiligten Knoten auch Knoten 2 das gemeinsame Protokollobjekt des Kreises 3, 4 und 5 zugeordnet.

Die Tiefensuche von „Infer Type“ muss also für Kanten, die zu bereits besuchten Knoten führen, bestimmen, ob diese Rückwärtskanten sind, und den zugehörigen Pfad im Spannbaum kennen. Dies wird durch Speichern des aktuellen Pfades zur Wurzel in einem `LinkedHashSet` erreicht.

Bisher wurden nur gerichtete Kanten besprochen. Die Gleichheitsconstraints entsprechen jedoch ungerichteten Kanten. Diese können wie in Abschnitt 4.3.1 als zwei gerichtete Kanten mit entgegengesetzter Richtung betrachtet werden. Dabei entstünde pro Gleichheitsconstraint eine Rückwärtskante. Dies kann in der Implementierung vereinfacht werden, indem für einen Gleichheitsconstraint direkt die Protokollobjekte beider Knoten zusammengeführt werden.

Das „Infer Type“-Refactoring würde auch ohne die beschriebene Behandlung von Rückwärtskanten korrekte Ergebnisse liefern. Nur innerhalb von Kreisen könnten in diesem Fall zu kleine Protokolle bestimmt werden. Den Knoten von Kreisen, die den kürzesten Pfad zur Wurzel haben (also auch die Wurzel selbst), werden aber korrekte benötigte Protokolle zugeordnet. Abgesehen von der Wurzel werden für das „Infer Type“-Refactoring minimale Protokolle — wie im kommenden Abschnitt 4.3.3 beschrieben — nur benötigt, wenn im Ausgangsquelltext eine Zuweisung zu einem Supertyp erfolgt. Das kann innerhalb von Kreisen aber nicht vorkommen.

Die Basis von „Infer Type“ soll zukünftig aber auch für Analysen oder Refactorings eingesetzt werden, die alle Deklarationselemente eines Programms berücksichtigen. Für diese Anwendungsfälle ist eine korrekte Berechnung des benötigten Protokolls aller Deklarationselemente in einem Durchlauf wünschenswert. Mit der beschriebenen (und implementierten) Behandlung von Rückwärtskanten kann dies durch Ausweitung der Tiefensuche auf den ganzen Graph erreicht werden.

4.3.3 Einführung des maximal verallgemeinerten Typs

Nachdem „Infer Type“ wie im vorhergehenden Abschnitt beschrieben das benötigte Protokoll für das ausgewählte Deklarationselement berechnet hat, muss daraus ein maximal verallgemeinerter Typ erzeugt und für das Deklarationselement eingesetzt werden.

Zunächst prüft „Infer Type“, ob der deklarierte Typ bereits maximal verallgemeinert ist. Dies ist in der Standardeinstellung des Refactorings gegeben, wenn das Protokoll des Typs mit dem berechneten minimalen Protokoll übereinstimmt und der Typ keine öffentlichen oder geschützten (protected) Felder oder geschützte Methoden deklariert.

Diese Festlegung beinhaltet eine gewisse Willkür, da innerhalb eines Pakets auch bei paketsichtbaren Feldern oder Methoden ein neuer Typ gewünscht sein könnte — eine paketsichtbare Methode ist in diesem Fall ja sichtbar. Mit derselben Begründung könnte

eine Entkopplung von privaten Feldern oder Methoden innerhalb einer Klasse gewünscht sein. Es gibt deswegen die Möglichkeit, das Verhalten von „Infer Type“ mittels eines Schalters so zu ändern, dass auch bei paketsichtbaren oder privaten Feldern und Methoden im ursprünglichen Typ ein neuer Typ erstellt wird.

Ist der aktuelle Typ nicht maximal verallgemeinert, wird nach einem bestehenden Supertyp gesucht, der dieses Kriterium erfüllt. Wird auch kein solcher Supertyp gefunden, muss ein neues Interface bzw. eine abstrakte Klasse mit dem benötigten Protokoll erstellt und als Supertyp des bisherigen Typs deklariert werden.

Der so bestimmte Typ wird für die Constraintvariable, die die Auswahl repräsentiert, als neuer Typ festgelegt. Dadurch sind in der Regel Constraints nicht mehr erfüllt. Mit einem weiteren Tiefendurchlauf werden neue Typen für zusätzliche Constraintvariablen ausgewählt (und eventuell erzeugt), so dass alle Constraints wieder erfüllt sind. Dabei werden vorgeordnet (pre order) alle verbundenen Constraints überprüft.

Für einen Gleichheitsconstraint $[E_1] = [E_2]$ wird der neue Typ für die andere Variable übernommen. Für einen Subtypconstraint $[E_1] \leq [E_2]$ mit $[E_1] = A$ und $[E_2] = B$ als Startbelegung sowie neu bestimmtem Typ I für E_1 wird wie von Steimann et al. [2006] für Zuweisungen beschrieben vorgegangen.

Dazu führen Steimann et al. [2006] auf der Menge $\mathbf{T}$ aller Typen eines Programms für die Teilmengenrelation von Protokollen die reflexive und transitive Relation $\leq_S$[21] ein: Mit $A, B \in \mathbf{T}$ ist $A \leq_S B$ definitionsgemäß gdw. $\pi(A) \supseteq \pi(B)$. A wird in diesem Fall als möglicher Subtyp von B bezeichnet. Für $A \leq_S B$ und $\pi(A) \neq \pi(B)$ wird $A <_S B$ geschrieben. $\leq_S$ wird auch als „Structural type conformance“ im Gegensatz zur „Type conformance by name“ bezeichnet. Die „Type conformance by name“ entspricht der deklarierten Subtypbeziehung, in Java wird dies durch `A extends B` oder `A implements B` ausgedrückt. Dafür wird das Symbol :< verwendet. Von Steimann et al. [2006] werden nun vier Fälle unterschieden:

1. $A \equiv B$. In diesem Fall kann I direkt für B verwendet werden.
2. $A :< B$ und $B <_S I$. In diesem Fall kann $B :< I$ gesetzt werden und I für B verwendet werden.
3. $A :< B$ und $I \leq_S B$. Hier muss im Normalfall nur $I :< B$ gesetzt werden, B kann dann unverändert als Typ beibehalten werden. Dies ist in Java aber nicht immer möglich, da Interfaces keine Subtypen von Klassen und Klassen nur Subtyp höchstens einer anderen Klasse sein können. In diesem Fall kann auf die Behandlung von Fall 4 ausgewichen werden. Zudem gibt es für „Infer Type“ eine Option, immer die Behandlung von Fall 4 durchzuführen, wenn B eine Klasse ist und der neue Typ I damit statt eines Interfaces eine abstrakte Klasse sein müsste.

[21] Von Steimann [2007] wird die Relation ohne Subskript S als $\leq$ eingeführt. Das Subskript wird hier jedoch zur Unterscheidung von der Notation der Subtypconstraints benötigt.

4. $A :< B$ und weder $B \leq_S I$ noch $I \leq_S B$. In diesem Fall wird für E_2 ein neuer Typ J benötigt, der Supertyp von I und von B ist. Für J wird $\pi(J) = \iota(E_2)$ gewählt und $B :< J$ und $I :< J$ gesetzt.

Auch für diesen Tiefendurchlauf muss überlegt werden, welche Bedeutung Nicht-Baumkanten zufällt. Bei diesen wurden beiden Constraintvariablen für E_1 und E_2 bereits neue Typen I_1 und I_2 zugewiesen.

- Vorwärtskanten stellen kein Problem dar, da über den Pfad des Baumes bereits sichergestellt ist, dass I_2 Subtyp von I_1 oder damit identisch ist. Die Bedingung ist also bereits erfüllt.
- Aus Rückwärtskanten ergeben sich Kreise, in denen alle Constraintvariablen denselben Typ haben. Für alle Constraints im Kreis gilt somit $A \equiv B$ und damit auch $I_1 \equiv I_2$. Die Bedingung ist also ebenso erfüllt.
- Bei Querkanten ist die Bedingung im Allgemeinen noch nicht erfüllt, jedoch kann man — außer das Typsystem von Java lässt es wie für Fall 3 beschrieben nicht zu — $I_1 :< I_2$ setzen, wenn nicht bereits $I_1 \equiv I_2$ oder $I_1 :< I_2$ gilt.

Konnten alle Bedingungen erfüllt werden, muss „Infer Type“ nur noch die neuen Typen im Quelltext einsetzen (siehe Abschnitt 5.6).

4.3.4 Konstruktion generischer Typen

Für den beschriebenen Algorithmus müssen in der Regel neue Interfaces erzeugt und einem bestehenden Typ als Supertyp hinzugefügt werden. Für Fall 4 aus Abschnitt 4.3.3 wird ein weiteres Interface benötigt, das sowohl Supertyp eines Typs aus der Ausgangshierarchie als auch Supertyp des ursprünglichen neuen Typs sein muss. Für nichtgenerische Typen können solche Interfaces ohne Probleme erzeugt und die entsprechenden Subtypbeziehungen deklariert werden.

Ersetzt ein Typ einen generischen Typ, muss er in der Regel ebenfalls ein generischer Typ sein. In diesem Fall stellt sich zusätzlich die Aufgabe, Typparameter für den neuen Typ festzulegen und bei der Deklaration der Subtypbeziehung zu belegen. Durch die vielfältigen Möglichkeiten, wie generische Hierarchien zusammenhängen können, erscheint dies insbesondere für Fall 4 zunächst recht aufwendig. Quelltext 4.13 zeigt eine Ausgangssituation, in der eine Subtypbeziehung verschachtelt parametrisiert ist und Fall 4 vorliegt. Zudem wird für das Deklarationselement b eine Wildcard verwendet.

```
1 class Client {
2         public static void main(String[] args) {
3             GenA<Integer, String> a = // Auswahl
4                         new GenA<Integer, String>();
```

```
5             a.a1(1, ""); // ohne diese Zeile liegt Fall 2 vor
6             a.b2(null);
7             GenB<? extends List<String>> b = a;
8             b.b1(null);
9         }
10 }
11
12 class GenA<E, T> extends GenB<List<T>>  {
13     public void a1(E e, T t) { }
14     public void a2(E e) { }
15 }
16 class GenB<B> {
17     public void b1(B b) { }
18     public void b2(B b) { }
19     public void b3(B b) { }
20 }
```

Quelltext 4.13: Beispiel für die Konstruktion verschachtelter generischer Supertypen

Um hier korrekte neue Typen einzuführen, kann sich „Infer Type“ aber an bestehenden Subtypbeziehungen orientieren: Einem neuen Typ kann immer ein existierender, parametrisierter Typ zugeordnet werden, der als Basis für die Parametrisierung dient (dieser wird im Folgenden auch als Basistyp bezeichnet). Das Subtypverhältnis zwischen den Basistypen kann dann auf die neuen Typen übertragen werden. Quelltext 4.14 zeigt das Ergebnis von „Infer Type“.

```
1 class Client {
2         public static void main(String[] args) {
3             InfA<Integer, String> a = // Auswahl
4                         new GenA<Integer, String>();
5             a.a1(1, ""); // ohne diese Zeile liegt Fall 2 vor
6             a.b2(null);
7             InfB<? extends List<String>> b = a;
8             b.b1(null);
9         }
10 }
11
12 class GenA<E, T> extends GenB<List<T>> implements InfA<E, T>  {
13 /*...*/
14 }
15 class GenB<B> implements InfB<B> {
16 /*...*/
17 }
18
19 interface InfA<E, T> extends InfB<List<T>> {
20     void a1(E e, T t);
21     void b2(List<T> b);
22 }
```

```
23 interface InfB<B> {
24     void b1(B b);
25 }
```

Quelltext 4.14: Ergebnis für die Konstruktion verschachtelter generischer Supertypen

Dem neuen Typ `InfA` wurde `GenA<Integer, String>` zugeordnet, dem Typ `InfB` `GenB<List<String>>` (dies ergibt sich aus der Analyse der Supertyphierarchie des parametrisierten Typs `GenA<Integer, String>`). Die neuen Typen erhalten dieselben Typparameter wie ihre Basistypen und können für diese dadurch einfach als Supertyp deklariert werden. Die Subtypbeziehung zwischen `InfA` und `InfB` ist genauso parametrisiert wie zwischen `GenA` und `GenB`. Durch die Verwendung der existierenden parametrisierten Hierarchie kann eine Methode wie `b2`, die in einem Supertyp deklariert ist, korrekt in einen neuen Typ auf einer anderen Hierarchiestufe übernommen werden.

Beim Einsetzen eines neuen Typs wird für die korrekten Typargumente dessen Basistyp im bisherigen Typ eines Deklarationselements gesucht und verwendet. Auf diese Weise bleiben Wildcards wie bei `b` erhalten. Wenn Fall 2 vorliegt — wie dies für Quelltext 4.13 der Fall wäre, wenn Zeile 5 entfernt wird — wird der neue Supertyp auch als Basistyp festgelegt. Die Typargumente, die für `a` verwendet werden, müssen dann an den neuen Basistyp angepasst werden. Dies geschieht wieder durch Suche des neuen Basistyps in der parametrisierten Hierarchie des bisherigen Typs. Quelltext 4.15 zeigt das Ergebnis, wenn Zeile 5 im Beispiel entfernt wird:

```
1  class Client {
2      public static void main(String[] args) {
3          InfB<List<String>> a = // Auswahl
4                           new GenA<Integer, String>();
5          a.b2(null);
6          InfB<? extends List<String>> b = a;
7          b.b1(null);
8      }
9  }
10 class GenA<E, T> extends GenB<List<T>> {
11 /*...*/
12 }
13 class GenB<B> implements InfB<B> {
14 /*...*/
15 }
16 interface InfB<B> {
17     void b2(B b);
18     void b1(B b);
19 }
```

Quelltext 4.15: Einführung generischer Typen bei Fall 2

4.3.5 Bestimmung von nicht änderbaren Deklarationselementen

„Infer Type“ muss in der Regel ein neues Interface erstellen und dem bestehenden Typ als Supertyp hinzufügen. Folgende Gründe verhindern deswegen (transitiv), dass das Refactoring durchgeführt werden kann:

- Es wird auf ein Feld zugegriffen. Dieses kann einem Interface nicht hinzugefügt werden.
- Eine private, geschützte oder paketsichtbare Methode wird aufgerufen. Auch diese kann ein Interface nicht enthalten.
- Die Auswahl wird einem externen Supertyp (d. h. einem Typ, der nicht im Quelltext vorliegt und somit nicht geändert werden kann) außer `java.lang.Object` zugewiesen. Das neue Interface kann dann nicht als Supertyp des externen Typs deklariert werden, dies verhindert in der Regel die Durchführung des Refactorings.
- Die Auswahl wird einem formalen Parameter einer externen Methode zugewiesen. Hier kann der Parameter der Methode nicht geändert werden. Auch dadurch kann das Refactoring nicht durchgeführt werden.

Alle diese Fälle werden bei der Constraintgenerierung durch Konstanten repräsentiert. Um eine genaue Rückmeldung an den Benutzer geben zu können, wird den Konstanten eine Beschreibung der Ursache hinzugefügt.

Während der Bestimmung des benötigten Protokolls nach Abschnitt 4.3.2 werden auf dieselbe Art, wie die benötigten Methoden berechnet werden, alle Konstanten bestimmt und die Beschreibung der jeweiligen Ursache dem Protokollobjekt hinzugefügt.

Zwei weitere Fälle im Zusammenhang mit Generics verhindern ebenso die Anwendung:

- Die Schranke eines Typparameters müsste geändert werden. Dies wird ebenso durch ein Konstante dargestellt (siehe Abschnitt 4.2.7).
- Das vom Compiler inferierte Typargument einer generischen Methode müsste wegen deren Rückgabetyp geändert werden. Dies muss „Infer Type“ während der Bestimmung des benötigten Protokolls separat bestimmen und wird im Folgenden näher erläutert.

Quelltext 4.16 zeigt ein einfaches Beispiel. Angewendet auf das Typargument `ClassOne` der Methode `getList()` kann „Infer Type“ keine Änderung durchführen. Da eine generische Methode nicht explizit sondern durch Inferenz basierend auf den Methodenargumenten parametrisiert wird, kann der Typ des Aufrufs von `singletonList(...)`

in diesem Beispiel nicht geändert werden, ohne dass auch der Konstruktoraufruf geändert wird. Dieser Fall kann nur auftreten, wenn eine Typvariable der Methode als Typargument des Rückgabewerts verwendet wird (wie für `public static <T> List<T> singletonList(T o)` in `List<T>`).

```
1 class Server {
2     public List<ClassOne> getList() {
3         return Collections.singletonList(new ClassOne());
4     }
5 }
```

Quelltext 4.16: Nicht änderbare Typargumente durch Verwendung generischer Methoden

Eine Beeinflussung des inferierten Typs einer generischen Methode von „Infer Type“ über den Rückgabetyp wird deswegen nicht in Betracht gezogen. „Infer Type“ stellt dafür während der Bestimmung des minimalen Protokolls über die Art der Constraintvariablen fest, ob von der Auswahl ausgehend ein Typargument einer generischen Methode über ihren Rückgabetyp erreicht wird und fügt eine entsprechende Beschreibung dem Protokollobjekt hinzu.

4.3.6 Anwendung auf Importdeklarationen

In den bisherigen Ausführungen wurde von der Auswahl eines einzelnen Deklarationselements ausgegangen. „Infer Type“ kann zusätzlich auf Importdeklarationen angewendet werden, um das benötigte Protokoll für alle Deklarationselemente eines Typs innerhalb einer Kompilierungseinheit, eines Pakets oder eines Pakets und dessen Unterpakete zu bestimmen und einzuführen.

Um diese Anwendung auf den bisher beschriebenen Fall der Auswahl eines einzelnen Deklarationselements zurückzuführen, legt „Infer Type“ Gleichheitsconstraints zwischen allen Deklarationselementen des Typs im gewünschten Anwendungsbereich an. Ein beliebiges dieser Deklarationselemente kann dann als Auswahl (bzw. Start) des Refactorings betrachtet werden. Alle Deklarationselemente erhalten so einen gemeinsamen neuen Typ, der als Protokoll die Vereinigungsmenge der benötigten Protokolle der einzelnen Deklarationselemente erhält.

Für generische Typen kann dieses Verfahren allerdings nicht verwendet werden, da diese in der Regel unterschiedlich parametrisiert werden und somit im Ausgangsprogramm keine Typgleichheit vorliegt. In diesem Fall können die zusätzlichen Gleichheitsconstraints zu falschen Ergebnissen führen. Mit einem anderen Ansatz könnte dieser Fall aber auch behandelt werden (siehe auch Abschnitt 8.2).

5 Architektur und Implementierung des „Infer Type“-Refactorings

Dieses Kapitel beschreibt, wie die im vorhergehenden Kapitel erläuterte Vorgehensweise und die weiteren Vorgaben aus Kapitel 3 umgesetzt wurden. Das Kapitel richtet sich primär an Leser, die sich mit dem Quelltext der Implementierung auseinandersetzen wollen. Es erläutert die Architektur, liefert alle wichtigen Einstiegspunkte und erklärt grundlegende Vorgehensweisen. Es werden dabei nur kurze Ausschnitte des Quelltextes präsentiert, der vollständige Quelltext findet sich auf der beiliegenden CD.

5.1 Voraussetzungen

Die Architektur des Refactorings ist stark von den Vorgaben geprägt. Es soll den Ansatz des Eclipse-Refactorings „Use Supertype Where Possible“ verfolgen und das Eclipse-Refactoring-Framework verwenden. Die grundsätzliche Plugin-Entwicklung für Eclipse wird z. B. von Gamma & Beck [2004], Clayberg & Rubel [2006] und von Bach [2007] ausführlich beschrieben, deswegen wird auf eine Darstellung in dieser Arbeit verzichtet. Das Eclipse-Refactoring-Framework wird jedoch — soweit es das „Infer Type“-Refactoring betrifft — in Abschnitt 5.1.1 erläutert. Abschnitt 5.1.2 erläutert kurz die grundsätzliche Architektur der constraint-basierten Refactorings von Eclipse. Dies stellt sich zwar nur eingeschränkt als wiederverwendbares Framework dar, der Architektur wurde aber — soweit für „Infer Type“ möglich — gefolgt.

5.1.1 Das Eclipse-Refactoring-Framework

Das Eclipse-Refactoring-Framework ist ein sprachunabhängiges Framework, das die Erstellung von Refactorings für die Eclipse-Plattform wesentlich erleichtert. Hauptaufgabe des Frameworks ist die Ablaufsteuerung eines Refactorings, zudem stellt es Dialoge und eine Vorschaufunktion bereit. Im Paket `org.eclipse.ltk.core.refactoring` befinden sich die grundlegenden Klassen des Frameworks, das Paket `org.eclipse.ltk.ui.refactoring` enthält die Komponenten für die Nutzerschnittstelle.

Ein Refactoring, das das Framework nutzt, erweitert die abstrakte Klasse

`org.eclipse.ltk.core.refactoring.Refactoring`. Diese deklariert drei wesentliche abstrakte Methoden, die implementiert werden müssen und in denen das konkrete Refactoring durchgeführt werden muss (siehe auch [Widmer 2006]). Der Ablauf kann von einem konkreten Refactoring geringfügig verändert werden, der Standardablauf erfüllt für „Infer Type“ aber alle Voraussetzungen:

- `checkInitialConditions` wird aufgerufen, bevor der Refactoring-Wizard angezeigt wird. Hier müssen die grundsätzliche Ausführbarkeit des Refactorings überprüft werden und mögliche Daten für die Nutzerschnittstelle gesammelt werden. Fehler oder Informationen für den Nutzer werden durch die Rückgabe einer Instanz von `org.eclipse.ltk.core.refactoring.RefactoringStatus` an das Framework berichtet.
- `checkFinalConditions` wird nach der letzten Seite des Wizards (abgesehen von der Vorschau) aufgerufen. Hier kann — basierend auf den Nutzereingaben — die endgültige Durchführbarkeit des Refactorings festgestellt werden. Fehler werden wie bei `checkInitialConditions` zurückgegeben.
- `createChange` wird abschließend aufgerufen und muss eine Instanz der Klasse `org.eclipse.ltk.core.refactoring.Change` zurückgeben. Diese beinhaltet die Quelltextänderungen des Refactorings, die vom Framework in der Vorschau angezeigt oder direkt auf den Quelltext angewendet werden.

Der Wizard eines konkreten Refactorings muss `org.eclipse.ltk.ui.refactoring.RefactoringWizard` erweitern. Diese Klasse stellt die Vorschau bereit und zeigt Fehlermeldungen an, die das Refactoring in den Überprüfungsmethoden zurückgegeben hat.

Der Start des Refactorings wird durch das Refactoring-Framework nicht festgelegt. Dies geschieht mit den üblichen Object und Editor actions (siehe z. B. [Clayberg & Rubel 2006]). Diese müssen das Refactoring und den Wizard erstellen und den Wizard mittels einer `org.eclipse.ltk.ui.refactoring.RefactoringWizardOpenOperation` aufrufen.

Das Framework ist nicht an Java als Zielsprache des Refactorings gebunden, es bietet deswegen selbst auch keinerlei sprachspezifische Hilfsmittel an. Diese werden separat von den Java Development Tools bereitgestellt. Das Framework bietet noch einige weitere Möglichkeiten wie Scripting und Aggregierung von Refactorings, die von „Infer Type“ aber nicht benötigt werden.

5.1.2 Architektur der constraint-basierten Eclipse-Refactorings

Die Constraintgenerierung und -lösung der constraint-basierten Refactorings erfolgt nicht zentral, sondern wird für die verschiedenen Refactorings von separaten Klassen

durchgeführt.[22] Das Vorgehen und damit die Architektur ist jedoch für alle Refactorings sehr ähnlich und wird auch von „Infer Type“ verwendet.

Die Refactorings bestimmen zunächst, welche Kompilierungseinheiten eingelesen werden müssen. Ein spezialisierter `ASTVisitor`[23], der als Constraintgenerator bezeichnet wird, besucht alle relevanten Knoten und ruft Methoden zur Erzeugung von Constraintvariablen und Constraints im sog. Constraintmodell auf. Er bindet die erzeugten Constraintvariablen an Knoten im Abstract syntax tree, so dass sie in anderen Visitor-Methoden verwendet werden können.

Das Constraintmodell kümmert sich um die Eindeutigkeit der Variablen, die in mehreren Kompilierungseinheiten verwendet werden (das sind z. B. Variablen für Methodenparameter). Bei „Infer Type“ und „Infer Generic Type Arguments“ erstellt es zusätzliche Constraints und Variablen, die für parametrisierte Typen bzw. Raw types benötigt werden. Es bietet für jeden Typ von Constraintvariablen (also z. B. für Parameter, für Rückgabetypen oder für lokale Variablen) eigene Erzeugungsmethoden an.

Die Klassen für die Constraintvariablen spezialisieren `org.eclipse.jdt.internal.corext.refactoring.typeconstraints2.ConstraintVariable2`. Jeder Variable kann ein Typ zugeordnet werden. Dazu wird im Paket `org.eclipse.jdt.internal.corext.refactoring.typeconstraints.types` eine eigene, speichersparende[24] Typumgebung bereitgestellt. Den Constraintvariablen können zudem beliebige zusätzliche Datenobjekte zugeordnet werden. Eine wichtige Aufgabe der spezialisierten Klassen ist es, über `hashCode` und `equals` die Identität der Constraintvariable festzulegen. Constraintvariablen können zudem Informationen über den Quelltextursprung halten.

Nachdem alle Kompilierungseinheiten eingelesen wurden, wird das Constraintmodell einer Komponente zur Lösung übergeben. „Infer Type“ hat hier zwei Komponenten für die beiden Lösungsstufen. Anschließend läuft das Refactoring durch alle Constraintvariablen und modifiziert den Quelltext, wenn deren Typ geändert wurde.

5.2 Paketstruktur

Das „Infer Type“-Plugin besteht aus 63 Klassen, die sich auf 11 Pakete verteilen. Tabelle 5.1 zeigt die Pakete mit Beschreibung des Aufgabengebiets und den darin enthaltenen Klassen. In den folgenden Abschnitten wird nach Ablauf des Refactorings geordnet auf wichtige Klassen eingegangen,[25] eine Beschreibung jeder einzelnen Klasse findet sich

[22] „Extract Interface“ erweitert die Implementierung „Use Supertype Where Possible“, „Generalize Declared Type“ und „Infer Generic Type Arguments“ haben jeweils eine eigene Implementierung.
[23] Aus dem Paket `org.eclipse.jdt.core.dom`.
[24] Die Verwendung von `org.eclipse.jdt.core.dom.ITypeBinding` würde den zugehörigen Abstract syntax tree im Speicher halten.
[25] Wenn dabei auf den Aufgabenbereich einer Klasse oder einer Methode eingegangen wird, soll das nicht heißen, dass die Methode oder Klasse selbst diese Aufgabe komplett übernimmt. In der Regel werden dazu weitere Methoden aufgerufen oder Hilfsklassen benötigt.

im Javadoc auf der beiliegenden CD. Abschnitt 5.3 erläutert den Start des Refactorings, Abschnitt 5.4 die Constrainterzeugung und die Berechnung des benötigten Protokolls, Abschnitt 5.5 den Refactoring-Wizard und Abschnitt 5.6 die Bestimmung der Typänderungen und die Modifikation des Quelltextes.

Pakete	*Klassen*

`org.intoJ.inferType3`
 	
Die externen Einstiegspunkte: das Plugin, die Refactoring-Klasse und eine Fassade für andere Plugins.
	`ExternalInferTypeRunnable`
	`InferTypePlugin`
	`InferTypeRefactoring`

`org.intoJ.inferType3.internal.actions`
 	
Die Actions, mit denen „Infer Type“ gestartet wird.
	`AbstractInferTypeAction`
	`InferTypeOnMemberAction`
	`InferTypeOnSelectionAction`

`org.intoJ.inferType3.internal.constraintModel` Das Modell und die Klasse für den Gleichheitsconstraint (der Subtypconstraint befindet sich in Paketen des JDT).	`EqualityTypeConstraint` `InferTypeConstraintsModel`

`org.intoJ.inferType3.internal.constraintModel.creation`
 	
Der Constraintgenerator, die Klasse, die die Kompilationseinheiten sucht, sowie Hilfsklassen.
	`Collector`
	`CollectorUtil`
	`InferTypeConstraintsCreator`
	`MethodHandler`
	`TypeDeclElementHandler`

`org.intoJ.inferType3.internal.constraintModel.util`
 	
Hilfsklassen, die vom Model und vom Creator verwendet werden.
	`AnonWorkaround`
	`ConstraintTraversal`
	`InferTypeGmlHandler`
	`Scope`
	`SelectedElement`
	`SelectedElementVisitor`

Pakete	*Klassen*
org.intoJ.inferType3.internal. constraintModel.variables Zusätzliche Klassen für Constraintvariablen, die von „Infer Type“ benötigt werden. Die anderen Klassen für Variablen befinden sich in org.eclipse.jdt.internal. corext.refactoring. typeconstraints2.	BoundVariable DeclaringTypeVariable ElementVariable GenericMethodTypeArgumentVariable IImmutableVariable ImmutableBoundVariable ImmutableElementVariable ImmutableTypeVariableWithDescr IndependentParameterTypeVariable IndependentReturnTypeVariable
org.intoJ.inferType3.internal. manipulation Klassen, die die Veränderung des Quelltextes durchführen.	BindingUpdater CodeGenerator GenerationUtil Manipulation TypeCreator
org.intoJ.inferType3.internal. typeCalc Die Klassen für die Lösung des Constraintsystems.	AccessSet AccessSetCalculation BindingSet CovarianceType TypeCalculation TypeNameGenerator
org.intoJ.inferType3.internal. typeCalc.hierarchy Eine Typhierarchie, die das Einfügen von neuen Typen unterstützt.	BindingWrapper ExistingTCType NewTCType ParameterizedTypeUtil TCType TypeHierarchy
org.intoJ.inferType3.internal. ui Der Refactoring-Wizard, der Dialog für den Anwendungsbereich des Import-Refactorings und die Einstellungsseite von „Infer Type“.	CheckStateListener CustomInputPage ImportScopeDialog InferTypeWizard ITContentProvider ITLabelProvider ITPreferencePage WrapperLeaf

Pakete	*Klassen*

`org.intoJ.inferType3.internal.util`
 	
Allgemeine Hilfsklassen.
	`BindingUtil`
	`Cache`
	`CacheMap`
	`ConversionUtil`
	`GmlWriter`
	`InternalCalculationException`
	`LogUtil`
	`RefactoringSaveHelper`
	`VariableSet`

Tabelle 5.1: Pakete und Klassen des „Infer Type“-Plugins

5.3 Start des Refactorings

Das Refactoring wird über die Actions aus dem Paket `org.intoJ.inferType3.internal.actions` gestartet. Diese werden über die Datei plugin.xml in Eclipse registriert. Die Aktionen bestimmen die Auswahl für das Refactoring. Die Editor-Action `InferTypeOnSelectionAction` muss dazu im Abstract syntax tree nach dem ausgewählten Knoten suchen. Für eine ungültige Auswahl oder eine Auswahl, die „Infer Type“ nicht behandeln kann (wie z. B. Enums), wird eine Meldung angezeigt. Für die Auswahl einer Typimportdeklaration wird ein `ImportScopeDialog` aufgerufen, der die Anwendungsbereiche Klasse, Paket, Paket und Unterpakete und Projekt anbietet.

Steht dann eine gültige Auswahl fest, wird eine Instanz von `InferTypeRefactoring` erstellt und damit der Wizard `InferTypeWizard` aufgerufen. Das Framework stößt dann durch Aufruf der Methode `InferTypeRefactoring.checkInitialConditions` die Generierung der Constraints an und zeigt bei längerer Laufzeit dieser Methode einen Statusdialog an.

5.4 Generierung der Constraints und Berechnung des benötigten Protokolls

Zunächst werden wie in Abschnitt 4.1 beschrieben alle Kompilationseinheiten gesucht, aus denen relevante Constraints erzeugt werden könnten. Dies übernimmt die Klasse `Collector`. Diese werden dann als Abstract syntax tree eingelesen und einer Instanz von `InferTypeConstraintsCreator` übergeben.

Die Klasse `InferTypeConstraintsCreator` ist von `HierarchicalASTVisitor` und damit von `ASTVisitor` abgeleitet und deklariert für jeden benötigten Knotentyp des Abstract syntax tree eine eigene Methode. Dies ist jeweils entweder eine `endVisit`-Methode, wenn keine Kontrolle über die weitere Verarbeitung erforderlich ist, oder eine `visit`-Methode. Insgesamt deklariert die Klasse 41 solcher Methoden. Als Beispiel wird in Quelltext 5.1 die Methode für den einfachsten Fall der Zuweisung gezeigt:

```
1  public class InferTypeConstraintsCreator extends
2                                      HierarchicalASTVisitor {
3      /* ... */
4      public void endVisit(final Assignment node) {
5          final ConstraintVariable2 ancestor = getConstraintVariable(
6                      node.getLeftHandSide());
7          final ConstraintVariable2 descendant = getConstraintVariable(
8                      node.getRightHandSide());
9          setConstraintVariable(node, ancestor);
10         if (ancestor != null && descendant != null) {
11             model.createSubtypeConstraint(descendant, ancestor);
12         }
13     }
14 }
```

Quelltext 5.1: Die Methode `InferTypeConstraintsCreator.endVisit(Assignment node)`

Die Constraintvariablen, die von der Methode mittels `getConstraintVariable` verwendet werden, wurden bereits von anderen Visitor-Methoden erzeugt (die linke Seite z. B. im Normalfall von `public void endVisit(SimpleName node)`) und an den Knoten des Abstract syntax tree gebunden. Man sieht in diesem Beispiel auch einen Aufruf von `InferTypeConstraintModel.createSubtypeConstraint`. Quelltext 5.2 zeigt die Methode in etwas verkürzter Form. Die Methode `shouldBind` überprüft, ob es sich bei einem der beiden Operanden um eine Typvariable handelt, `constraintVariables.addOrGet` fügt die Constraintvariablen der Menge aller Variablen hinzu und verwendet, falls vorhanden, bereits existierende, identische[26] Variablen. Damit wird anschließend der Constraint erzeugt und falls es diesen Constraint noch nicht gab, den Variablen als beteiligter Constraint hinzugefügt. Die Methode `createElementEqualsConstraints` erzeugt dann zwischen den Elementvariablen von parametrisierten Typen die in Abschnitt 4.2 beschriebenen zusätzlichen Constraints. Dazu werden eine Reihe von `makeElementVariable`-Methoden benötigt, die auch die Constraints für Schranken von Typparametern erstellen.[27]

Um die Übersichtlichkeit zu erhöhen, wurden für „Infer Type“ zwei aufwendige Teile der Constrainterzeugung in Hilfsklassen ausgelagert. Die Klasse

[26] Im Sinne von `equals` und `hashCode`.
[27] Die Behandlung der Elementvariablen ist relativ aufwendig und lässt sich hier nicht komplett darstellen. Es sei deswegen auf den Quelltext des Plugins auf der beiliegenden CD verwiesen.

```
public void createSubtypeConstraint(ConstraintVariable2 descendant,
                                     ConstraintVariable2 ancestor,
                                     boolean createNewElementVars) {
    if (!shouldBind(descendant.getType()) ||
                !shouldBind(ancestor.getType())) {
        return;
    }
    descendant = constraintVariables.addOrGet(descendant);
    ancestor = constraintVariables.addOrGet(ancestor);

    final ITypeConstraint2 constraint =
                 new SubTypeConstraint2(descendant, ancestor);
    if (!allConstraints.contains(constraint))
        allConstraints.add(constraint);
        setVariableUsage(descendant, constraint);
        setVariableUsage(ancestor, constraint);

        if (isElementFollowVarType(descendant) &&
                    isElementFollowVarType(ancestor)) {
            createElementEqualsConstraints(descendant, ancestor,
                        true, createNewElementVars);
        }

    }
}
```

Quelltext 5.2: Die Methode `InferTypeConstraintsModel.createSubtypeConstraint`

`MethodHandler` enthält die Logik für die Constrainterzeugung von Methodenaufrufen und -deklarationen und wird von den entsprechenden sechs `endVisit`-Methoden in `InferTypeConstraintsCreator` aufgerufen. Die Klasse `TypeDeclElementHandler` bearbeitet alle Typdeklarationen, die eine parametrisierte Klasse als Supertyp haben (siehe Abschnitt 4.2.4).

Der Constraintgenerator erzeugt für Felder und lokale Variablen zwei Constraintvariablen: eine für den Identifier (dies ist die eigentliche Constraintvariable) und eine für den AST-Knoten, der den Typ repräsentiert, und verbindet diese mit Gleichheitsconstraints.[28] Dadurch gibt es nur einen Typ von Constraintvariablen, der den Quelltextursprung speichert (`org.eclipse.jdt.internal.corext.refactoring.typeconstraints2.TypeVariable2`). Diese Variablen werden von den beiden `makeTypeVariable`-Methoden des Modells erstellt. Quelltext 5.3 zeigt eine davon als Beispiel für die Methoden, die Constrainvariablen erzeugen (dies sind alle Methoden mit Präfix `make`).

```
public TypeVariable2 makeTypeVariable(Type type,
                                      boolean createChildrenForTypeVars) {
```

[28] Dieses Verfahren erleichtert die Implementierung im Rahmen des `ASTVisitor` und wird von „Use Supertype Where Possible“ und „Infer Generic Type Arguments“ ebenso eingesetzt.

```
3       ICompilationUnit cu =
4               RefactoringASTParser.getCompilationUnit(type);
5       TType ttype = getBoxedType(type.resolveBinding(), null);
6       if (ttype == null) {
7           return null;
8       }
9
10      CompilationUnitRange range = new CompilationUnitRange(cu, type);
11      TypeVariable2 typeVariable = new TypeVariable2(ttype, range);
12      TypeVariable2 storedCv = (TypeVariable2)
13              constraintVariables.addOrGet(typeVariable);
14
15      if (!hasElementVariables(storedCv)) {
16          makeElementVariables(storedCv, ttype,
17                               createChildrenForTypeVars);
18      }
19      return storedCv;
20  }
```

Quelltext 5.3: Die Methode `InferTypeConstraintsModel.makeTypeVariable`

Weitere Methoden dieser Art existieren für Methodenparameter und -rückgabewerte, Variablen, Typargumente generischer Methoden etc. Für Methodenparameter wird neben den Constraintvariablen für den Typ und den Identifier eine zusätzliche Methodenparametervariable erstellt, die für den Aufruf einer Methode verwendet wird.[29]

Nachdem alle Constraints — wie in Abschnitt 4.2 beschrieben — erzeugt wurden, wird von `AccessSetCalculation` das benötigte Protokoll berechnet. Einstiegspunkt ist die Methode `calculate`. Zunächst werden in der Methode `initAccessSets` die Ausgangsbelegungen festgelegt, anschließend wird in der rekursiven Methode `calculateAccessSet` — wie in Abschnitt 4.3.2 beschrieben — für alle zusammenhängenden Constraintvariablen das benötigte Protokoll berechnet. Hilfsmethoden für das Durchlaufen des Graphen befinden sich in der Klasse `ConstraintTraversal`. Die Vereinigung der Protokollobjekte für Rückwärtskanten wird in `mergeSets` durchgeführt.

Instanzen der Klasse `AccessSet` übernehmen die Aufgabe der Protokollobjekte. Eine zusätzliche Klasse `BindingSet` wird zur eindeutigen Speicherung der Methoden unter Beachtung kovarianter Rückgabetypen verwendet. Zudem referenziert `AccessSet` die beteiligten Constraintvariablen und speichert die Durchführbarkeit einer Typänderung. Ist keine Änderung durchführbar, sind alle Ursachen in textueller Form hinterlegt.

[29] Auch das ist zur Vereinfachung der Implementierung von „Use Supertype Where Possible“ übernommen worden.

5.5 Der Refactoring-Wizard

Nachdem das benötigte Protokoll berechnet wurde, kehrt die Methode `InferTypeRefactoring.checkInitialConditions` zurück. Wurde festgestellt, dass das Deklarationselement nicht änderbar ist, zeigt das Framework einen Dialog an. Ansonsten wird der Refactoring-Wizard von „Infer Type“ (Klasse `InferTypeWizard`) angezeigt. Dieser ist in Abbildung 5.1 dargestellt.

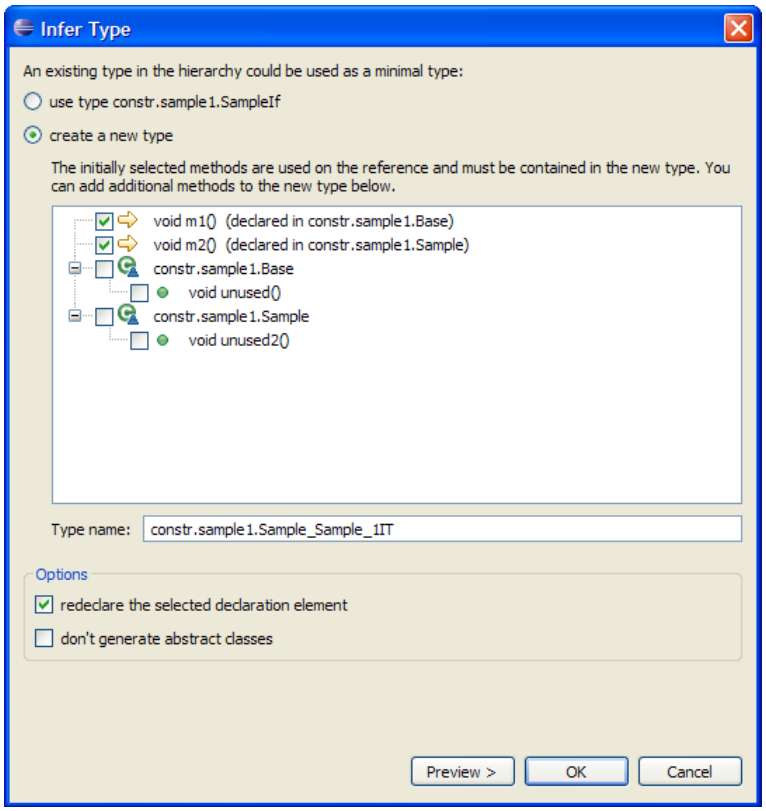

Abbildung 5.1: Der Refactoring-Wizard von „Infer Type“

Die angezeigte und einzige für „Infer Type“ spezialisierte Seite des Wizards ist in `ITInputPage` implementiert. Für das Beispiel wurde ein existierender Typ gefunden, es kann aber trotzdem ein neuer Typ erstellt werden. Das benötigte Protokoll ist am Anfang der Baumkontrolle aufgeführt und durch Pfeile gekennzeichnet — diese Methoden können auch nicht abgewählt werden. Alle weiteren in der Hierarchie deklarierten Methoden werden darunter dargestellt und können dem neuen Typ hinzugefügt werden.

Der Name und das Paket des neuen Typs können in der Textbox unter der Baumkontrolle angegeben werden. Werden für Fall 4 aus Abschnitt 4.2.3 zusätzliche Typen benötigt, erhalten diese automatische Namen. Diese basieren auf dem ursprünglichen Typ und dem Identifier des Deklarationselements, das mit dem zusätzlichen Typ deklariert wird. Die Abbildung zeigt auch die beiden Optionen, die nach Berechnung des

benötigten Protokolls zur Auswahl stehen. Zum einen kann auf eine Redeklarierung des ausgewählten Deklarationselements verzichtet werden. In diesem Fall wird nur der neue Typ erzeugt und als Supertyp des ausgewählten Typs hinzugefügt. Zum anderen wird hier die in Abschnitt 4.2.3 für Fall 3 erwähnte Option angeboten, keine abstrakten Klassen zu erzeugen.

Nachdem die Optionen festgelegt wurden, kann mittels „Preview“-Button eine Vorschau des Refactorings angezeigt werden. Mit „OK“ wird das Refactoring direkt ausgeführt. Für beide Fälle wird `checkFinalConditions` und `createChange` aufgerufen, die die Berechnung der Typänderungen wie im nächsten Abschnitt beschrieben durchführen.

5.6 Einführung des ausgewählten Typs

In der Methode `checkFinalConditions` wird zunächst die Berechnung der Typänderungen angestoßen. Dies übernimmt die Klasse `TypeCalculation`. In der Methode `determineTypes` wird zu Beginn der neue Typ erstellt und in die Hierarchie eingeordnet. Um dies nicht direkt mit den Quelltexten durchführen zu müssen, benötigt „Infer Type“ eine eigene Repräsentation der Typhierarchie. Diese wird mit `TypeHierarchy(IType initialType)` basierend auf dem Typ der Auswahl bereits zu Beginn des Refactorings erzeugt und enthält anfangs den dem Konstruktor übergebenen Typ und dessen Supertypen. Es gibt zwei von `TCType` abgeleitete Klassen, die existierende (`ExistingTCType`) und neue (`NewTCType`) Typen repräsentieren. Für beide Typen können neue Supertypen festgelegt oder Supertypen entfernt werden.

Danach wird unter Zuhilfenahme von Methoden aus `ConstraintTraversal` der Graph durchlaufen. Die in Abschnitt 4.3.3 beschriebene Behandlung der Subtypconstraints wird in `TypeCalculation.handleSubtypeConstraint` durchgeführt.

Anschließend werden die nötigen Quelltextänderungen bestimmt. Hierfür ist die Klasse `Manipulation` verantwortlich. Ausgehend von der Methode `calculateChanges` werden drei Aktionen durchgeführt:

1. Für alle Deklarationselemente mit neuem Typ werden deren Knoten von `rewriteDeclarationElements` im Abstract syntax tree gesucht und ersetzt. Dies muss als erstes geschehen, da sich dadurch die Signaturen von Methoden, die in einem neuen Typ verwendet werden, ändern können.

2. Für alle neuen Typen (d. h. für alle Instanzen von `NewTCType`) wird von `introduceTypes` eine neue Kompilationseinheit erstellt.

3. Für alle existierenden Typen der Hierarchie (d. h. für alle Instanzen von `ExistingTCType`) wird in `rewriteHierarchy` überprüft, ob sich die Deklaration

der Supertypen ändern muss, und diese Änderung gegebenenfalls durchgeführt.

Damit ist der Ablauf von `checkFinalConditions` beendet. Die anschließend aufgerufene Methode `createChange` gruppiert nur noch die Änderungen und übergibt sie dem Framework.

6 Anwendungsszenarien und Tests

In diesem Kapitel werden zunächst Anwendungsszenarien vorgestellt, die das „Infer Type“-Refactoring im Einsatz zeigen. Anschließend werden in Abschnitt 6.4 die automatisierten Tests beschrieben, mit denen die Korrektheit von „Infer Type“ überprüft wurde.

6.1 Entkopplung durch Bestimmung kontextspezifischer Interfaces

Im Folgenden soll eine Klasse gezeigt werden, die in verschiedenen Kontexten unterschiedliche Rollen annimmt. Maximal verallgemeinerte Interfaces können diese Rollen eines Objektes in einem bestimmten Kontext darstellen [Steimann 2001; Steimann et al. 2003] und damit von der konkreten Implementierung soweit wie möglich entkoppeln.

Als Beispiel soll im Folgenden ein Szenario dienen, das in einem ERP-System (Enterprise Resource Planning) vorkommen könnte. Dargestellt werden zwei stark vereinfachte Komponenten: eine Projektplanung und eine Buchhaltungskomponente. Eine Klasse, die in beiden Komponenten verwendet wird, ist `StaffMember`[30], die die Mitarbeiter des Unternehmens repräsentiert. Wie man in Quelltext 6.1 sieht, deklariert die Klasse eine Reihe von Methoden, die Informationen über einen Mitarbeiter zurückgeben.

```
1  public class StaffMember {
2      private String name;
3      private String department;
4      private Date dateOfBirth;
5      private int salary;
6      private Account account;
7      private EnumSet<Skill> skills;
8
9      public String getName() {
10         return name;
11     }
12     public String getDepartment() {
13         return department;
```

[30] Diese Klasse repräsentiert — wie an der Namensgebung erkennbar — bereits eine Rolle einer Person (siehe auch [Steimann 2001]). Die Rollen, die ein Mitarbeiter in einem Unternehmen spielen kann, werden aber im Folgenden noch verfeinert.

```
14      }
15      public Date getDateOfBirth() {
16          return dateOfBirth;
17      }
18      public int getSalary() {
19          return salary;
20      }
21      public Account getAccount() {
22          return account;
23      }
24      public EnumSet<Skill> getSkills() {
25          return skills;
26      }
27      /* ... */
28  }
```

Quelltext 6.1: Die Klasse `StaffMember`

Quelltext 6.2 zeigt die Verwendung der Klasse im Rahmen der Projektplanung. Ein Projekt hat einen Manager, dessen Abteilung das Projekt zugeordnet ist, und eine Reihe von Teilnehmern (zu denen der Manager selbst gehört), deren Qualifikationen für die Projektplanung benötigt werden. Um die Teilnehmer in alphabetischer Reihenfolge zurückgeben zu können, wird das Set `participants` mit der Klasse `ParticipantComparator` sortiert. Die kurze Beispielklasse zeigt dabei den Einsatz von parametrisierten Containern, parametrisierten Supertypen, einer generischen Methode und der erweiterten For-Schleife. Zudem wird die Klasse `StaffMember` in zwei verschiedenen Rollen verwendet: als Manager der Projekts und als einfacher Teilnehmer. Für dieses Beispiel ist es interessant, diese beiden Rollen auch durch verschiedene Interfaces auszudrücken. Der Manager gehört als Angestellter einer Abteilung an — diese Information wird für die Zuordnung des Projekts benötigt. Der Projektteilnehmer muss das nicht. So könnten beispielsweise freie Mitarbeiter, die nicht alle Eigenschaften der Klasse `StaffMember` haben, ebenso an Projekten teilnehmen.

```
1  public class Project {
2      private StaffMember manager;
3      private SortedSet<StaffMember> participants =
4                  new TreeSet<StaffMember>(new ParticipantComparator());
5
6      public void setManager(StaffMember manager) {
7          this.manager = manager;
8          addParticipant(manager);
9      }
10     public void addParticipant(StaffMember participant) {
11         participants.add(participant);
12     }
13     public Collection<StaffMember> getParticipants() {
```

```
14          return Collections.unmodifiableCollection(participants);
15      }
16      public EnumSet<Skill> getSkills() {
17          EnumSet<Skill> allSkills = EnumSet.noneOf(Skill.class);
18          for (StaffMember participant : participants) {
19              allSkills.addAll(participant.getSkills());
20          }
21          return allSkills;
22      }
23      public String getDepartment() {
24          return manager.getDepartment();
25      }
26      private static class ParticipantComparator
27                                  implements Comparator<StaffMember> {
28          public int compare(StaffMember o1, StaffMember o2) {
29              return o1.getName().compareTo(o2.getName());
30          }
31      }
32  }
```

Quelltext 6.2: Die Klasse `Project`

Wendet man „Infer Type“ auf die Parameter von `addParticipant` und `setManager` an, werden zwei neue Typen erzeugt, die nur die benötigten Methoden enthalten. Quelltext 6.3 zeigt das Ergebnis für die Klasse `Project`, Abbildung 6.1 zeigt die neue Typhierarchie von `StaffMember`. Dabei ist es unerheblich, in welcher Reihenfolge die beiden Parameter bearbeitet werden.

```
1  public class Project {
2      private Manager manager;
3      private SortedSet<Participant> participants =
4                  new TreeSet<Participant>(new ParticipantComparator());
5
6      public void setManager(Manager manager) {
7          this.manager = manager;
8          addParticipant(manager);
9      }
10     public void addParticipant(Participant participant) {
11         participants.add(participant);
12     }
13     public Collection<Participant> getParticipants() {
14         return Collections.unmodifiableCollection(participants);
15     }
16     public EnumSet<Skill> getSkills() {
17         EnumSet<Skill> allSkills = EnumSet.noneOf(Skill.class);
18         for (Participant participant : participants) {
19             allSkills.addAll(participant.getSkills());
20         }
21         return allSkills;
```

```
22      }
23      public String getDepartment() {
24          return manager.getDepartment();
25      }
26      private static class ParticipantComparator
27                               implements Comparator<Participant> {
28          public int compare(Participant o1, Participant o2) {
29              return o1.getName().compareTo(o2.getName());
30          }
31      }
32  }
```

Quelltext 6.3: Die Klasse `Project` — Ergebnis

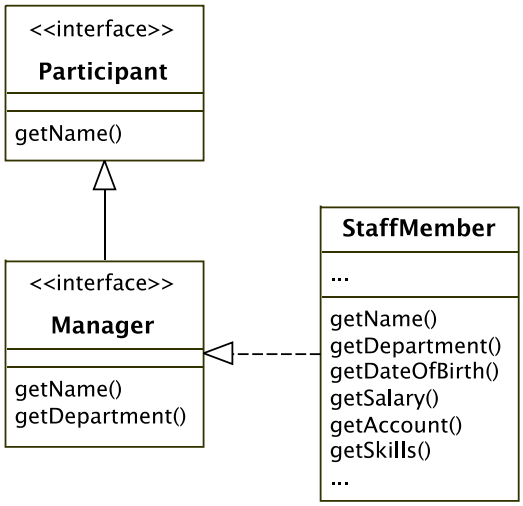

Abbildung 6.1: Die Typhierarchie von `StaffMember` nach Entkopplung von `Project`

Das Beispiel zeigt sowohl eine Typargumentänderung von `participants` als auch von der Deklaration der Superklasse `Comparator` von `ParticipantComparator`. Ohne diese Änderungen hätten auch die Methodenparameter nicht geändert werden können. Dieser Zwang bietet gegenüber einem entsprechenden Quelltext ohne Verwendung von Generics aber Vorteile. In der Regel zeigen die Typargumente ja einen Datenfluß auf, der sich in Java 1.4 nicht in der statischen Typisierung wiederfindet. Quelltext 6.4 zeigt eine entsprechende Version der Klasse `Project` ohne Generics:

```
1  public class Project {
2      private StaffMember manager;
3      private SortedSet participants =
4                  new TreeSet(new ParticipantComparator());
5
6      public void setManager(StaffMember manager) {
7          this.manager = manager;
```

```
8           addParticipant(manager);
9       }
10      public void addParticipant(StaffMember participant) {
11          participants.add(participant);
12      }
13      public Collection getParticipants() {
14          return Collections.unmodifiableCollection(participants);
15      }
16      public Set getSkills() {
17          Set allSkills = new HashSet();
18          for (Iterator it = participants.iterator(); it.hasNext();) {
19              StaffMember participant = (StaffMember)it.next();
20              allSkills.addAll(participant.getSkills());
21          }
22          return allSkills;
23      }
24      public String getDepartment() {
25          return manager.getDepartment();
26      }
27      private static class ParticipantComparator
28                                  implements Comparator {
29          public int compare(Object o1, Object o2) {
30              return ((StaffMember)o1).getName().
31                   compareTo(((StaffMember)o2).getName());
32          }
33      }
34  }
```

Quelltext 6.4: Die Klasse `Project` ohne Verwendung von Generics

Bei Anwendung wie oben beschrieben ist Quelltext 6.5 das Ergebnis von „Infer Type“. Das Interface `Manager` enthält nur die Methode `getDepartment`, für den Parameter von `addParticipant` wurde kein Methodenaufruf festgestellt, da Downcasts nicht verfolgt werden.[31] Damit wird `Object` als Typ eingesetzt.

```
1  public class Project {
2      private Manager manager;
3      private SortedSet participants =
4                  new TreeSet(new ParticipantComparator());
5
6      public void setManager(Manager manager) {
7          this.manager = manager;
8          addParticipant(manager);
9      }
10     public void addParticipant(Object participant) {
11         participants.add(participant);
12     }
```

[31] Auch die vorherige Version von „Infer Type“ verfolgt keine Downcasts.

```
13      public Collection getParticipants() {
14          return Collections.unmodifiableCollection(participants);
15      }
16      public Set getSkills() {
17          Set allSkills = new HashSet();
18          for (Iterator it = participants.iterator(); it.hasNext();) {
19              StaffMember participant = (StaffMember)it.next();
20              allSkills.addAll(participant.getSkills());
21          }
22          return allSkills;
23      }
24      public String getDepartment() {
25          return manager.getDepartment();
26      }
27      private static class ParticipantComparator
28                                  implements Comparator {
29          public int compare(Object o1, Object o2) {
30              return ((StaffMember)o1).getName().
31                   compareTo(((StaffMember)o2).getName());
32          }
33      }
34  }
```

Quelltext 6.5: Die Klasse `Project` ohne Verwendung von Generics — Ergebnis

Die Klasse `Accounting` aus Quelltext 6.6 stellt ein weiteres Beispiel für die Verwendung der Klasse `StaffMember` dar. In diesem Fall würde die Anwendung auf die beiden lokalen Variablen `employee` eine vollständige Entkopplung liefern. Dabei würden zwei Interfaces erzeugt, eines mit den Methoden `getName`, `getAccount` und `getSalary` und eines nur mit `getSalary`. Bevorzugt man nur ein Interface, das man beispielsweise `Employee` nennen könnte, kann „Infer Type“ auf die Importdeklaration des Typs `StaffMember` angewendet werden.[32] Es wird dann das Ergebnis in Quelltext 6.7 erzeugt. Dabei gibt es nur ein neues Interface `Employee`, das alle drei aufgerufenen Methoden deklariert. Die komplette Hierarchie von `StaffMember` ist in Abbildung 6.2 dargestellt. Wird die Klasse `StaffMember` in der Klasse `Accounting` zusätzlich für andere Zwecke verwendet (z. B. um den Leiter der Buchhaltungsabteilung zu referenzieren), bietet sich alternativ die Anwendung auf das Typargument des Feldes `employees` an. Dies liefert für das gegebene Beispiel dasselbe Ergebnis.

```
1  public class Accounting {
2      private List<StaffMember> employees = new ArrayList<StaffMember>();
3
4      public void transferSalaries() {
5          for (StaffMember employee : employees) {
```

[32] Gibt es eine solche Importdeklaration nicht (z. B. weil sich die Klassen im selben Paket befinden), muss sie in der aktuellen Version von „Infer Type“ vor Anwendung des Refactorings der Kompilierungseinheit hinzugefügt werden.

```
6               wireTransfer(employee.getName(), employee.getAccount(),
7                            employee.getSalary());
8           }
9       }
10      public int getSalaries() {
11          int salaries = 0;
12          for (StaffMember employee : employees) {
13              salaries += employee.getSalary();
14          }
15          return salaries;
16      }
17      private void wireTransfer(String name, Account account, int salary) {
18          /* ... */
19      }
20  }
```

Quelltext 6.6: Die Klasse `Accounting`

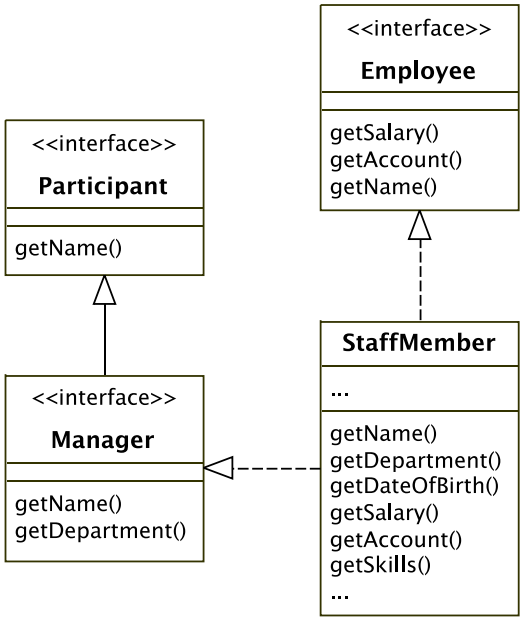

Abbildung 6.2: Die komplette Typhierarchie von `StaffMember`

```
1   public class Accounting {
2       private List<Employee> employees = new ArrayList<Employee>();
3
4       public void addEmployee(StaffMember employee) {
5           employees.add(employee);
6       }
7       public void transferSalaries() {
8           for (Employee employee : employees) {
```

```
9               wireTransfer(employee.getName(), employee.getAccount(),
10                              employee.getSalary());
11          }
12      }
13      public int getSalaries() {
14          int salaries = 0;
15          for (Employee employee : employees) {
16              salaries += employee.getSalary();
17          }
18          return salaries;
19      }
20      /* ... */
21  }
```

Quelltext 6.7: Die Klasse `Accounting` — Ergebnis

6.2 Projektweite Entkopplung mit „Infer Type“

„Infer Type“ kann nicht nur der Bestimmung einzelner kontextspezifischer Interfaces dienen, sondern auch dazu, in einem Schritt eine projektweite Entkopplung von einer bestimmten Implementierung durchzuführen. Fiedler [2007] beschreibt als einführendes Beispiel die Klasse `FTPClient`, die in einem Content-Management-System an vielen Stellen zum Datentransfer verwendet wird. Der Datentransfer soll in diesem Szenario zukünftig mit verschlüsselten Protokollen erfolgen, deswegen muss die Implementierung ausgetauscht werden. Erster Schritt ist eine vollständige Entkopplung von der konkreten Implementierung `FTPClient`. Diese Klasse deklariert eine große Anzahl von Methoden[33] — es ist deswegen wünschenswert, die vom Projekt verwendeten Methoden zu kennen, um unnötige Arbeit bei einer Ersatzimplementierung zu vermeiden.

Wie der von Fiedler [2007] übernommene Quelltext 6.8 beispielhaft zeigt, ist davon auszugehen, dass sich die für die einzelnen Deklarationselemente benötigten Protokolle unterscheiden. Die Einführung einer Vielzahl kontextspezifischer Interfaces ist aber nicht unbedingt gewünscht.

```
1  public class Example1 {
2      FTPClient client = new FTPClient();
3      boolean checkConnectionData(String host, String user,
4              String passwd) throws IOException {
5          client.connect(host);
6          client.login(user, passwd);
7          return client.isConnected();
8      }
9  }
```

[33] Als Inspiration diente die Klasse `FTPClient` des Jakarta-Commons-Net-Projekts (siehe `http://jakarta.apache.org/commons/net/`).

```
10 public class Example2 {
11     private FTPClient client;
12
13     public InputStream getInputStream(String remotePath)
14             throws IOException {
15         if (client.isConnected()) {
16             InputStream result = client.
17                         retrieveFileStream(remotePath);
18             return result;
19         } else {
20             return null;
21         }
22     }
23 }
24 public class Example3 {
25     public void makeLogDir(FTPClient connectedClient,
26             String[] messages) throws IOException {
27         connectedClient.makeDirectory("log");
28         connectedClient.changeWorkingDirectory("log");
29         OutputStream os = connectedClient.
30                               storeFileStream("messageLog");
31         writeStrings(messages, os);
32     }
33     private void writeStrings(String[] msgs, OutputStream os) {...}
34 }
```

Quelltext 6.8: Beispiele für die Verwendung von `FTPClient`

Mit „Infer Type“ kann nun relativ einfach eine projektweite Entkopplung in einem Schritt durchgeführt werden. Wendet man „Infer Type“ auf eine Typimportdeklaration von `FTPClient` an und wählt das Projekt als Anwendungsbereich, wird nur ein Interface für das insgesamt benötigte Protokoll von `FTPClient` erzeugt und für alle Deklarationselemente eingesetzt.

Voraussetzung ist dabei, dass „Infer Type“ für alle Deklarationselemente durchführbar ist. Der Quelltext der Klassen, von denen eine projektweite Entkoppelung durchgeführt werden soll, muss also änderbar sein. Zur Ausklammerung von intern benötigten Methoden ist es dabei sinnvoll, die Klassen vorab in ein eigenes Projekt zu bewegen.

Ähnlich kann „Infer Type“ durch Anwendung auf Typimportdeklarationen mit den kleineren Anwendungsbereichen „Kompilierungseinheit“, „Paket“ und „Paket und Unterpakete“ eingesetzt werden. In den kleineren Anwendungsbereichen kann damit auch eher eine spezifische Rolle erfasst werden. Hier ist kein eigenes Projekt für die zu entkoppelnden Klassen erforderlich.

„Infer Type“ kann keine vollständige Entkopplung erreichen, wenn Objekte erzeugt werden. So wird der Konstruktor in Zeile 2 von Quelltext 6.8 nicht ersetzt — es soll ja auch zunächst diese Implementierung verwendet werden. Klassischerweise wird diese

Abhängigkeit durch das Factory method pattern [Gamma et al. 1994; Steimann 2006] gelöst. Ein weiteres Entwurfsmuster für diese Problematik, das in den letzten Jahren an Popularität gewonnen hat, ist das Dependency injection pattern [Fowler 2004]. Dessen Zusammenhang mit „Infer Type“ soll im nächsten Abschnitt erläutert werden.

6.3 „Infer Type“ als Grundlage für das Dependency injection pattern

Das Dependency injection pattern (auch als Inversion of control bezeichnet) ist ein von verschiedenen Frameworks (z. B. PicoContainer[34], Guice[35] und Spring[36]) verwendetes Entwurfsmuster, das der Vermeidung starker Kopplung zwischen Klassen dient [Fowler 2004]. Die grundsätzliche Idee ist, dass eine Klasse benötigte Objekte nicht selbst erzeugt, sondern von außen zugeführt (injected) bekommt. Damit eine Entkopplung erreicht werden kann, muss aber zunächst ein Interface vorliegen, das anstelle einer konkreten Implementierung verwendet werden kann. Dies kann mit „Infer Type“ wie in den vorherigen Abschnitten beschrieben erreicht werden.

Quelltext 6.9 zeigt, wie mit Guice die Klasse `Example1` aus Quelltext 6.8 vollständig von `FTPClient` entkoppelt werden kann. Durch die Annotation `@Inject` wird angezeigt, dass das Framework dieses Feld belegen soll. Die Klasse `TransferModule` legt die Bindungen fest, die von Guice verwendet werden. Die aufrufende Klasse `ExampleClient` erzeugt damit einen `Injector`, der in der Methode `getInstance` automatisch eine Instanz von `FTPClient` erzeugt und dem Feld `client` zuweist.

Diese Schritte muss der Entwickler selbst durchführen, nachdem er mit „Infer Type“ ein Interface erzeugt und eingesetzt hat. Hier gibt es aber bereits Überlegungen für weitere Automatisierungen basierend auf „Infer Type“ — mehr dazu in Abschnitt 8.2.

```
1  public class Example1 {
2      @Inject
3      TransferClient client;
4      boolean checkConnectionData(String host, String user,
5              String passwd) throws IOException {
6          client.connect(host);
7          client.login(user, passwd);
8          return client.isConnected();
9      }
10 }
11 public interface TransferClient {
12     void connect(String host);
13     void login(String user, String passwd);
```

[34] http://www.picocontainer.org
[35] http://code.google.com/p/google-guice/
[36] http://www.springframework.org

```
14      boolean isConnected();
15      /*...*/
16  }
17  public class FTPClient implements TransferClient {
18      /*...*/
19  }
20  class ExampleClient {
21      public void start() throws IOException {
22          Injector injector = Guice.createInjector(new TransferModule());
23          Example1 example1 = injector.getInstance(Example1.class);
24          example1.checkConnectionData("", "", "");
25      }
26  }
27  class TransferModule implements Module {
28      public void configure(Binder binder) {
29          binder.bind(TransferClient.class).to(FTPClient.class);
30      }
31  }
```

Quelltext 6.9: Entkopplung von Konstruktoren mit Guice

6.4 Tests

Um die Alltagstauglichkeit und die Korrektheit des Refactorings zu überprüfen, wurden automatisierte Tests auf allen Deklarationselementen verschiedener Projekte durchgeführt. Hierzu wurde ein eigenes Plugin der Vorgängerversion für das neue „Infer Type“-Refactoring angepasst. Dieses sucht — geordnet nach deklarierten Typen — alle Deklarationselemente eines Projekts und übergibt sie nacheinander an die Fassadeklasse `ExternalInferTypeRunnable`. Nach einer Änderung wird das Projekt neu kompiliert und im Fehlerfall abgebrochen. Solange bei einem Gesamtdurchlauf Änderungen durchgeführt werden, wird ein weiterer Gesamtdurchlauf gestartet. Das Plugin befindet sich ebenfalls auf der beiliegenden CD.

Mit diesen Tests kann zum einen festgestellt werden, ob das Refactoring für ein komplettes Projekt ausschließlich typkorrekte Änderungen durchführt. Zum anderen kann dieser Test für Projekte auf Basis von Java 1.4 zum Vergleich mit der Vorgängerversion dienen. Ein direkter Vergleich der Ergebnisse ist dabei allerdings nicht möglich: Zwar wird das Refactoring in beiden Versionen auf alle Deklarationselemente zunächst in der gleichen Reihenfolge angewendet — der grundsätzliche Unterschied der Implementierungen kann dann aber zu einer anderen Bearbeitungsreihenfolge im Zuweisungs- bzw. Constraintgraphen führen. Dadurch kommt es zu Unterschieden bei der automatischen Namensgebung zusätzlicher Typen.

Um trotzdem automatisiert die Gleichwertigkeit beider Implementierungen feststel-

len zu können, wurde folgender Ansatz gewählt: Zunächst wird das neue „Infer Type“-Refactoring wie beschrieben auf ein Projekt angewendet. Auf das Ergebnis wird anschließend die Vorgängerversion angewendet. Wieder ausgehend von der Ursprungsversion des Projekts wird der Vorgang in umgekehrter Reihenfolge wiederholt. Wurde für beide Abläufe keine Änderung vom zweiten Refactoring durchgeführt, wird von der Gleichwertigkeit der Implementierungen ausgegangen. Diese Form des Tests wurde erfolgreich mit junit 3.8.1[37], JHotDraw 6.0 Beta 1[38], DrawSWF 1.2.9[39] und der Implementierung der Vorgängerversion[40] durchgeführt.

Weiterhin wurde die neuen „Infer Type“-Implementierungen an Projekten getestet, die Generics einsetzen. Im Rahmen der Entwicklung von „Infer Type“ wurde ein Projekt mit einer Reihe von Grenzfällen der Generics-Nutzung erstellt. Dieses befindet sich auf der beiliegenden CD. Zudem wurde der Test auf die Implementierung von „Infer Type“ selbst, auf junit 4.2[41] und auf Matrex 1.0[42] durchgeführt.

Für die Implementierung von „Infer Type“ selbst und für junit 4.2 wurde zudem ein Lauf durchgeführt, in dem keine Änderung von Typargumenten gestattet war, um einen Hinweis auf die praktische Relevanz der Typargumentänderungen zu erhalten. Tabelle 6.1 enthält sowohl die Anzahl der Anwendungen als auch die Anzahl der Redeklarationen (d. h. die Zahl aller Typänderungen im Quelltext). Letzteres beinhaltet auch die Änderungen von Typargumenten selbst, ein Deklarationselement kann zudem mehrmals redeklariert worden sein. Die Zahlen zeigen jedoch durchaus, dass die Unterstützung von Typargumentänderung — zumindest für bestimmte Projekte — die Anwendbarkeit des Refactorings signifikant steigert.

	junit 4.2	Infer Type
Anwendungen ohne Typargumentänderung	100	134
Anwendungen mit Typargumentänderung	133	158
Steigerung der Anwendungen	33%	18%
Redeklarationen ohne Typargumentänderung	182	217
Redeklarationen mit Typargumentänderung	366	372
Steigerung der Redeklarationen	101%	71%

Tabelle 6.1: Steigerung der Anwendbarkeit durch Änderung von Typargumenten

Eine Schwachstelle der Implementierung, die sie mit der Vorgängerversion und den Generalisierungsrefactorings von Eclipse teilt[43], trat bei den Projekten JHotDraw, DrawSWF

[37] http://www.junit.org
[38] http://www.jhotdraw.org
[39] http://drawswf.sourceforge.net
[40] http://www.fernuni-hagen.de/ps/prjs/InferType/
[41] http://www.junit.org
[42] http://matrex.sourceforge.net
[43] Siehe Bug 53343 auf https://bugs.eclipse.org/bugs/ oder [Tip et al. 2003]

und Matrex jeweils einmal zu Tage: Durch die Typänderungen können überladene Methoden mehrdeutig werden oder doppelt auftreten. Für diese Problematik gibt es nach Wissen des Autors noch keine effiziente allgemeine Lösung. Der Benutzer muss in diesem Fall entweder den Aufruf durch Casts eindeutig machen oder eine der Methoden umbenennen.[44] Quelltext 6.10 zeigt ein Beispiel, bei dem eine Umbenennung notwendig wäre, da sonst nach Anwendung von „Infer Type“ auf `o1` oder von „Use Supertype Where Possible“ auf die Klasse `Overloading` zwei Methoden `void overloaded(Object)` in der Klasse `Overloading` definiert wären.

```
1 public class Overloading {
2     public void overloaded(Overloading o1) { }
3     public void overloaded(Object o2) { }
4
5     public void client() {
6         overloaded(new Overloading());
7     }
8 }
```

Quelltext 6.10: Beispiel für überladene Methoden

[44] Durch die vollständige Undo-Möglichkeit kann der Benutzer auch leicht auf das Refactoring verzichten, sollte dieses Problem auftreten.

7 Diskussion

In diesem Kapitel werden zunächst die Einsatzmöglichkeiten des neuen „Infer Type“-Refactorings diskutiert. Anschließend wird eine Bewertung der constraint-basierten Implementierung durchgeführt und auf Einschränkungen des Refactorings hingewiesen. Das Kapitel schließt mit einer Diskussion verwandter Arbeiten ab.

7.1 Einsatzmöglichkeiten von „Infer Type“

Das vorangegangene Kapitel hat anhand von Anwendungsszenarien gezeigt, wie mit „Infer Type“ einfach das gewünschte Ergebnis — eine Entkopplung von Klassen — erreicht werden kann. Dabei wurden kontextspezifische Interfaces für verschiedene Rollen erzeugt, die die Wiederverwendbarkeit des Programms maximieren. Dem Nutzer wird mit „Infer Type“ ein wichtiges Werkzeug zur interface-basierten Programmierung in die Hand gegeben.

Die Beispiele in Abschnitt 6.1 haben auch gezeigt, dass die für „Infer Type“ entwickelte, weitergehende Unterstützung von Java 5 entscheidend für die Anwendbarkeit des Refactorings sein kann. Die Möglichkeit, Typargumente zu ändern, ist eine Neuerung, von der auch Refactorings wie „Use Supertype Where Possible“ profitieren könnten. Wie man auch an den Ergebnissen in Tabelle 6.1 sieht, kann dies signifikant die Anzahl der Deklarationselemente steigern, für die eine Typänderung durchgeführt werden kann. Schranken von Typparametern können zwar (noch) nicht geändert werden, dies ist jedoch kein grundsätzliches Problem. Zusammen mit anderen Erweiterungen wie der Einbeziehung von Konstruktoraufrufen und der speziellen Behandlung von Downcasts könnte dies im Rahmen weiterer Arbeiten behandelt werden.[45]

In machen Situationen wird der Einsatz von „Infer Type“ allerdings nicht ganz so gradlinig wie in Kapitel 6 beschrieben verlaufen können. Eine Problematik, vor die sich Nutzer gestellt sehen, ist die Auswahl der Deklarationselemente, auf die „Infer Type“ angewendet werden soll. Um eine vollständige Entkopplung zu erreichen, müssen alle Deklarationselemente eines Typs in einer Klasse ersetzt werden. Die neue Möglichkeit, den Anwendungsbereich auf eine ganze Klasse auszudehnen, erlaubt dies zwar in einem Schritt — man verliert dabei jedoch die Möglichkeit, spezifische Interfaces für einzelne Rollen

[45] Für einen weiteren Ausblick siehe Abschnitt 8.2.

wie in Abschnitt 6.1 beschrieben zu erzeugen.

Wählt man deswegen die sukzessive Anwendung auf alle Deklarationselemente eines Typs, bestimmt „Infer Type“ in der Regel unterschiedliche neue Typen. Eine Häufung neuer Typen ist jedoch in der Regel auch nicht gewünscht. Die neue „Infer Type“-Implementierung bietet durch die Auswahl zusätzlicher Methoden die Möglichkeit an, manuell auf existierende Typen zurückzugreifen.[46] Dies hilft dem Nutzer, die Anzahl der erzeugten Interfaces zu reduzieren. Die Anzahl der insgesamt erzeugten Typen hängt aber dann immer noch von der Anwendungsreihenfolge ab. Wird zunächst ein Interface mit einer geringen Anzahl an Methoden erzeugt, muss für ein größeres benötigtes Protokoll zwingend ein zusätzliches Interface generiert werden. Eine optimale Lösung wäre die gleichzeitige Auswahl mehrerer Deklarationselemente — dies ließe sich durch ein spezielles Nutzerinterface erreichen.

Für die Entscheidungsfindung ist es grundsätzlich sehr interessant, dem Nutzer vor Anwendung des Refactorings möglichst viele Informationen über die inferierten Typen einer größeren Zahl von Deklarationselementen zu bieten. Hier gibt es auch bereits Werkzeuge: Der von Steimann & Mayer [2007] vorgestellte Type Access Analyser zeigt für einen ausgewählten Typ graphisch alle möglichen Subprotokolle in Form eines Gitters an. Zu jedem Subprotokoll werden eine Reihe weiterer Informationen präsentiert: alle Supertypen, deren Protokoll dem Subprotokoll entspricht, die Deklarationselemente, die mit diesen Typen deklariert sind, und die Deklarationselemente, deren benötigtes Protokoll dem Subprotokoll entspricht. Die Subtypbeziehung deklarierter Typen werden mit zusätzlichen Kanten im UML-Stil dargestellt. Eine Reihe von Refactorings können auf Basis dieser Informationen durchgeführt werden. Dies schließt nicht nur die Erzeugung neuer Supertypen ein, sondern auch das Zusammenfassen existierender Typen.

Fiedler [2007] kombiniert den Ansatz des Subprotokoll-Gitters auf Basis der Vorgängerversion des Type Access Analysers [47] mit Metriken, die bei der Auswahl interessanter Interfaces helfen, und bietet an, ausgewählte Interfaces zu erzeugen und einzuführen. Diese Werkzeuge könnten die Informationen über inferierte Typen vom neuen „Infer Type“-Refactoring beziehen und von der Unterstützung von Java 5 profitieren. Zudem sind eine Reihe weiterer Refactorings basierend auf dem Kern von „Infer Type“ geplant — mehr dazu in Abschnitt 8.2.

[46] Nach der Auswahl der Methoden wird nochmals überprüft, ob dadurch ein existierender Typ in Frage kommt.

[47] Die Vorgängerversion ist Teil der intoJ-Suite und kann unter `http://www.intoj.org` heruntergeladen werden.

7.2 Bewertung der constraint-basierten Implementierung

Der constraint-basierte Ansatz hat sich für die Implementierung von „Infer Type“ bewährt. Insbesondere konnten durch Constraints Abhängigkeiten beim Einsatz von Generics erfasst werden, die sich aus dem Zuweisungsgraph, wir ihn die Vorgängerimplementierung verwendet, nicht ergeben. So war es möglich, die Anwendbarkeit für Projekte, bei denen Generics zu Einsatz kommen, deutlich zu steigern.

Zudem gewährleistet der constraint-basierte Ansatz auch die Trennung zwischen der Analyse des Quelltextes, aufgrund dessen die Constraints erstellt werden, und dem eigentlichen Algorithmus. Der von Steimann [2007] beschriebene Algorithmus konnte dabei gut für die Anwendung auf Typconstraints angepasst werden. Der Algorithmus selbst benötigt dabei durch die einheitliche Repräsentation von Generics durch Subtyp- und Gleichheitsconstraints keine spezielle Kenntnis der Sprachkonstrukte von Java 5.[48] Beide Schritte des Algorithmus operieren ausschließlich auf den erstellten Constraints.

Verbesserungsmöglichkeiten für die Laufzeit der Analyse ergeben sich insbesondere durch die Eingrenzung der analysierten Kompilierungseinheiten. Bisher werden alle Kompilierungseinheiten eingelesen, in denen der ausgewählte Typ vorkommt. Während der Constraintgenerierung wird dabei keine Flußanalyse ausgehend von dem ausgewählten Deklarationselement durchgeführt — vielmehr werden alle Constraints separat erstellt. Deren Zusammenhang ergibt sich anschließend durch die in Abschnitt 4.2 beschriebene Eindeutigkeit der Constraintvariablen. Abhängig von der Verbreitung des ausgewählten Typs kann es zu einer größeren Zahl unnötigerweise erstellten Constraints kommen.

Zwar traten für die bisher getesteten Projekte keine Probleme mit der Laufzeit oder mit dem Speicherverbrauch des Refactorings auf, für häufig verwendete Typen sehr großer Projekte könnte aber eine sinnvolle Verbesserung erreicht werden. Dazu müsste bereits während der Constraintgenerierung basierend auf den bereits erstellten Constraints eine statische Flußanalyse durchgeführt werden, die aufgerufene Methoden und Zugriffe auf Felder und damit die relevanten Kompilierungseinheiten bestimmt. Palsberg & Schwartzbach [1994] zeigen eine alternative Möglichkeit mittels Trace graph, aus dem ein weiteres Constraintssystem zur Bestimmung der einzulesenden Methoden erzeugt wird.

Für Werkzeuge wie den Type Access Analyser [Steimann & Mayer 2007], die die benötigten Protokolle aller Deklarationselemente eines bestimmten Typs verwenden, würde diese Änderung jedoch keinen Vorteil darstellen. Der in dieser Arbeit beschriebene Algorithmus kann die benötigten Protokolle aller Deklarationselemente eines Typs in einem Durchlauf berechnen, alle eingelesenen Kompilierungseinheiten werden dann auch

[48] Für die Änderung des Quelltextes ist das zwar wieder erforderlich. Dies ist aber ein weiterer separater Schritt.

zwingend benötigt.

7.3 Einschränkungen von „Infer Type“

Steimann [2007] führt einige Problematiken von „Infer Type“ auf, die auch für diese Implementierung des Refactorings zutreffen. Zum einen handelt es sich um sprachbedingte Einschränkungen, die die Einführung eines neuen Typs oder die Nutzung eines existierenden, maximal verallgemeinerten Typs verhindern, zum anderen um einige subtile Probleme, die durch die Typänderungen von Deklarationselementen hervorgerufen werden.

Eine sprachbedingte Problematik ist die Beschränkung auf öffentliche Methoden. Dies ist nötig, um Interfaces einsetzen zu können. Steimann [2007] argumentiert, dass der Zugriff auf öffentliche Felder durch Getter- und Setter-Methoden gekapselt werden könne — dies entspräche auch der gängigen Praxis. Private oder geschützte Methoden und Feldern seien nicht Gegenstand einer Entkopplung — ersteres, weil dies eine Entkopplung einer Klasse von sich selbst wäre, letzteres, da über die Vererbung sowieso eine stärkere Form der Kopplung vorläge. Auch die Entkopplung innerhalb eines Pakets sei nicht sinnvoll, da dieses ja stark zusammenhängende Klassen enthalten solle („hohe Kohäsion“). Für den Zweck der Entkopplung seien diese Einschränkungen also hinnehmbar.

Eine weitere Beschränkung ist, dass „Infer Type“ in der Regel nur auf Typen angewendet werden kann, die änderbar sind. Dies ist nötig, um einen neuen maximal verallgemeinerten Typ als Supertyp deklarieren zu können.

Für beide Fälle bietet „Infer Type“ genaue Fehlermeldungen an, durch die der Nutzer etwa eine Kapselung eines Feldes oder die Einbeziehung eines Typs in das Projekt manuell durchführen und anschließend „Infer Type“ nochmals anwenden kann. Auf den Typconstraints von „Infer Type“ könnte aber für beide Fälle alternativ auch ein Algorithmus wie von „Generalize Declared Type“ angewendet werden. Damit würde man zumindest einen möglichst allgemeinen, existierenden Supertyp finden. Auf diese Art könnte „Infer Type“ auch das Refactoring „Generalize Declared Type“ vollständig ersetzen.

Werden für neu eingesetzte Typen nicht nur Interfaces, sondern auch Klassen verwendet,[49] kann es vorkommen, dass eine Subtypbeziehung nicht hergestellt werden kann. Der Grund ist, dass Klassen nur direkter Subtyp einer einzigen Klasse und Interfaces grundsätzlich nicht Subtyp einer Klasse sein können. Wird allerdings ein neues Interface erstellt und die Option gewählt, keine abstrakten Klassen zu generieren, konnte der Autor bisher keine Situation erzeugen, in der das Refactoring scheitert.[50]

[49] Dies kann sowohl für Fall 3 aus Abschnitt 4.3.3 als auch für eine existierende, maximal verallgemeinerte Superklasse auftreten.

[50] Dies ist nicht die Standardeinstellung, da dadurch in anderen Fällen eine größere Anzahl zusätzlicher neuer Typen erzeugt werden müsste.

Ziel eines Refactorings ist es, Änderungen am Quelltext durchzuführen, ohne dass sich das Verhalten des Programms ändert — dies ist zunächst für „Infer Type“ auch der Fall. Die geänderten Referenzen akzeptieren allerdings nun zusätzliche Objekte — was ja Zweck der Entkopplung ist. Dabei können jedoch Probleme entstehen, wenn implizite Kontrakte ignoriert werden. Wird beispielsweise der Typ einer Referenz von `Set` in `Iterable` geändert, geht die Anforderung verloren, dass keine Duplikate enthalten sein dürfen. „Infer Type“ kann diese Art von Kontrakten nicht analysieren, die neue Implementierung bietet dem Nutzer aber durch die Vorschau des Eclipse-Refactorings-Frameworks zumindest eine bessere Möglichkeit, die Änderungen selbst zu überprüfen.

Eine weitere Problematik, die auch im Rahmen dieser Implementierung noch nicht gelöst wurde, hängt mit überladenen Methoden zusammen (siehe Abschnitt 6.4). Durch die Änderung von Typdeklarationen können Methodenaufrufe, die bisher eindeutig waren, uneindeutig werden. Dies wird als Fehler vom Compiler festgestellt, das Refactoring kann dann rückgängig gemacht werden. Es können aber dadurch auch überladene Methoden unbeabsichtigt in überschreibende Methoden verwandelt werden. Dies stellt für den Compiler in der Regel keinen Fehler dar[51] und ist damit noch problematischer. Das Problem wird auch in [Tip et al. 2003] und [Steimann 2007] beschrieben — eventuell kann der Algorithmus aus [Dicky et al. 1996] für „Infer Type“ angepasst werden. Das Problem ließe sich zumindest umgehen, indem für jede Änderung eines Parameters nach überladenen Methoden gesucht wird, und das Refactoring bei einem möglichen Konflikt nicht durchgeführt wird. Bei den durchgeführten Tests ist der Fall der uneindeutigen Methodenaufrufe allerdings sehr selten aufgetreten.[52]

7.4 Verwandte Arbeiten

Das neue „Infer Type“-Refactoring basiert bezüglich des Refactorings selbst auf [Steimann et al. 2006] bzw. [Steimann 2007] und in Hinblick auf den constraint-basierten Ansatz auf [Tip et al. 2003] und [Fuhrer et al. 2005]. Der Inferenzalgorithmus, der in [Steimann 2007] für den Zuweisungsgraphen beschrieben wird, wurde für die Anwendung auf Typconstraints adaptiert. Die Verwandtschaft mit diesen Arbeiten ist deswegen sehr groß und wurde bereits in den vorhergehenden Kapiteln im Detail erörtert.

Die beiden Refactorings von Eclipse mit der größten Ähnlichkeit sind „Extract Interface“ und „Generalize Declared Type“. Beide Refactorings bauen jedoch nur auf existierenden Typen auf. „Generalize Declared Type“ arbeitet zudem nur lokal, d. h. es kann Zuweisungen nicht verfolgen. Auch „Extract Interface“ kann das neue Interface nicht einsetzen, wenn eine Zuweisungen zu einem Supertyp besteht. Für eine solche Zuweisung

[51] Ausnahmen können beispielsweise bei unterschiedlichen Sichtbarkeiten auftreten.
[52] Siehe auch Abschnitt 6.4.

müsste die Typhierarchie wie in Abschnitt 4.3.3 beschrieben angepasst werden.

Grundlagen für [Tip et al. 2003] und [Fuhrer et al. 2005] sowie für diese Arbeit wurden von [Palsberg & Schwartzbach 1991] gelegt. Ebenso beschäftigen sich [Wang & Smith 2001] mit constraint-basierter Typinferenz — die Zielsetzung ist dort jedoch genau umgekehrt zu der von „Infer Type“, es soll ein möglichst spezieller Typ inferiert werden. Damit können insbesondere Downcasts überprüft werden. [Khedker et al. 2003] führen Typinferenz mit dem Ziel größerer Genauigkeit mittel Datenflußanalyse durch. Diese dynamischen Analysen sind präziser als statische Analysen, für „Infer Type“ aber nicht nützlich, da nur (statische) Typannotationen geändert werden können. Der Compiler würde präzisere Ergebnisse zurückweisen.

Ein weiterer Typinferenzalgorithmus basierend auf Begriffsanalyse (concept analysis) wird von Snelting & Tip [2000] vorgestellt. Es wird eine Klassenhierarchie erzeugt, in der jedes Objekt ausschließlich die benötigten Felder und Methoden enthält. Mit KABA wurde ein System vorgestellt, das diesen Algorithmus auf komplette Java-Programme anwendet und alle Typdeklarationen und Instanzierungen ändert [Streckenbach & Snelting 2004]. Der Fokus liegt dabei nicht auf der allgemeineren Verwendbarkeit eines Programms, sondern auf einem maximalen Zusammenhang von Modulen. Es wird sichergestellt, dass alle Felder einer Klasse immer gemeinsam verwendet werden, um die Größe der Objekte (im Sinne des Speicherverbrauchs) zu minimieren. Dafür müssen neue spezialisierte Subtypen erstellt werden. Die Modifikation erfolgt auf Basis des Bytecodes, eine Veränderung des Quelltextes ist für die Zielsetzung nicht erforderlich.

Bach [2007] stellt ein Eclipse-Plugin vor, das mögliche Typgeneralisierungen im Quelltext markiert und ein Refactoring wie „Infer Type“ aufrufen kann. Diese Arbeit verwendet die Algorithmen von „Infer Type“ und von „Generalize Declared Type“. In Abschnitt 7.1 wurde bereits der Type Access Analyser [Steimann & Mayer 2007] und die Arbeit von Fiedler [2007] erwähnt, die den Algorithmus der Vorgängerversion von „Infer Type“ verwenden und die bei der Auswahl nützlicher Interfaces eine wesentliche Hilfestellung geben können.

8 Schlussbetrachtungen

8.1 Zusammenfassung

Ziel dieser Arbeit war die Entwicklung eines Refactorings, das mittels constraintbasierter Typinferenz ein ausgewähltes Deklarationselement mit einem maximal verallgemeinerten Typ umdeklariert. Das Refactoring sollte auf Programme in Java 5 angewendet werden können und auf den Eclipse Java Development Tools basieren. Als maximal verallgemeinerter Typ wurde dabei ein Typ verstanden, der ausschließlich die für das ausgewählte Deklarationselement benötigten Methoden (d. h. das benötigte Protokoll) enthält. Ein solches Refactoring bietet dem Nutzer die Möglichkeit, automatisiert Klassen zu entkoppeln und Rollen von Typen durch Verwendung kontextspezifischer Interfaces auszudrücken. Es erleichtert damit entscheidend die interface-basierte Programmierung.

Die bestehenden Generalisierungsrefactorings von Eclipse („Use Supertype Where Possible“, „Extract Interface“ und „Generalize Declared Type“) unterstützen dazu Generics nicht weitreichend genug. Die von „Infer Type“ gewünschte Unterstützung wurde in Kapitel 3 herausgearbeitet. Es wurde gefordert, dass „Infer Type“ nicht nur auf parametrisierte Typen selbst angewendet werden soll, sondern dass auch Typargumente parametrisierter Typen — falls nötig — von „Infer Type“ geändert werden sollen. Ohne letzteres wäre die Anwendbarkeit des Refactorings für Programme, die von parametrisierten Containern Gebrauch machen, deutlich eingeschränkt.

In Kapitel 4 wurde erläutert, wie Typconstraints für Java-5-Programme erstellt werden können, so dass „Infer Type“ und auch andere Generalisierungsrefactorings diese weitreichenden Änderungen durchführen können. Die dabei beschriebenen Typconstraints sind zweistellige Relationen der Art „Typgleichheit“ und „Subtyp von oder Typgleichheit“. Mit diesen lassen sich alle Bedingungen für die Typkorrektheit eines Programms darstellen. Kernpunkt der Unterstützung von Generics ist die Erzeugung separater Constraintvariablen und Constraints für Typargumente. Alle Sprachkonstrukte von Java 5 und die dazu erzeugten Constraints wurden besprochen.

Anschließend wurde dargelegt, wie auf Basis dieser Constraints ein maximal verallgemeinerter Typ bestimmt und eingeführt werden kann. Da für „Infer Type“ neue Typen erzeugt werden, unterscheidet sich die Lösung deutlich von den Lösungen der Gene-

ralisierungsrefactorings von Eclipse. Es wurde ein zweistufiger Lösungsalgorithmus beschrieben: Zunächst wird — basierend auf einem Tiefendurchlauf des Constraintgraphen — das benötigte Protokoll bestimmt. Gibt es einen Typ in der Hierarchie mit diesem Protokoll, kann dieser verwendet werden. Ansonsten wird ein neues Interface erzeugt und dem Ausgangstyp als Supertyp hinzugefügt. Nach Einsetzen des maximal verallgemeinerten Typs für das ausgewählte Deklarationselement wird in einem zweiten Durchlauf des Graphen sichergestellt, dass alle Constraints wieder erfüllt sind. Jeder Constraint kann in einem einzigen Schritt erfüllt werden. Dabei werden in der Regel weitere Deklarationselemente geändert und unter Umständen auch zusätzliche neue Typen benötigt.

Das Refactoring wurde als Eclipse-Plugin implementiert. Es erzeugt die beschriebenen Constraints, bestimmt deren Lösung und führt Quelltextänderungen durch. Für die gewünschte Unterstützung von Java 5 musste das Constraint-Framework von Eclipse erweitert werden. Die enge Integration des Refactorings in Eclipse bietet eine Vorschau und Undo-Funktionalität, zudem werden eine Reihe neuer Optionen wie das Hinzufügen zusätzlicher Methoden angeboten. Die Eckpunkte dieser Implementierung wurden in Kapitel 5 beschrieben. In Kapitel 6 wurde anhand von Anwendungsszenarien beispielhaft der Nutzen von „Infer Type“ gezeigt. Zudem wurden die weitreichenden Tests beschrieben, mit denen die Korrektheit des Refactorings überprüft wurde.

8.2 Ausblick

In dieser Arbeit wurden bereits einige Punkte genannt, die eine Verbesserung des Kerns von „Infer Type“ darstellen würden:

- Schranken von Typparametern können bisher nicht geändert werden. Hier müssten eigene Constraints für Deklarationselemente mit Typvariablen erstellt werden. Zudem ist zu überlegen, wie eine Typvariable mit mehr als einer Schranke behandelt werden kann.
- Die Anzahl der einzulesenden Kompilierungseinheiten könnte durch einen zusätzlichen Analyseschritt während der Constraintgenerierung verringert werden. Dies wurde in Abschnitt 7.3 besprochen.
- Eine effiziente Analyse des Erhalts überladener Methoden würde nicht nur für „Infer Type“, sondern für viele Eclipse-Refactorings ein grundlegendes, wenn auch selten auftretendes Problem lösen.
- Der Zugriff auf Felder, der Aufruf nicht-öffentlicher Methoden sowie Zuweisungen zu Referenzen, deren Typ nicht im Quelltext vorliegt, verhindern bisher die Anwendung von „Infer Type“, da für diese Fälle kein neues Interface erzeugt und

eingeführt werden kann. Der Algorithmus könnte aber so erweitert werden, dass zumindest ein möglichst allgemeiner existierender Typ bestimmt wird, der für die Referenz verwendet werden kann.

- „Infer Type“ bietet momentan als Option an, ob Typen als maximal verallgemeinert betrachtet werden können, die paketsichtbare oder geschützte Methoden oder Felder deklarieren. Die Sichtbarkeit dieser Methoden und Felder hängt vom Ort eines Deklarationselements ab. Eine interessante Verbesserung wäre die automatische Bestimmung dieser relativen Sichtbarkeit von Feldern und Methoden. Dies kann allerdings nicht direkt mit den erzeugten Typconstraints erfolgen, ein zusätzlicher Analyseschritt wäre erforderlich.

- Die Anwendung auf Importdeklarationen ist bisher auf nicht-generische Typen beschränkt. Diese Möglichkeit wäre für generische Typen natürlich auch interessant. Die Erzeugung zusätzlicher Constraints kann hier jedoch wie in Abschnitt 4.3.6 beschrieben zu einem inkonsistenten Constraintsystem führen. Eventuell ist eine Lösung erforderlich, die nicht auf zusätzlichen Constraints aufbaut.

Der Einsatz der vorgestellten Constraints und des Lösungsalgorithmus muss nicht auf das „Infer Type“-Refactoring beschränkt bleiben. Eine Reihe weiterer neuer Refactorings könnten den Kern von „Infer Type“ zur Bestimmung benötigter Methoden oder zur Einführung neuer Typen nutzen. Am Lehrgebiet Programmiersysteme wird deswegen geplant, den Kern dieser Arbeit in eine eigenständige Komponente mit Namen ITcore fortzuentwickeln, die durch eine Reihe von Optionen verschiedene Anwendungsgebiete abdecken kann [Kegel & Steimann 2007]. Fragestellungen, die je nach Anwendung unterschiedlicher Behandlung bedürfen, sind unter anderem:

- Wie werden Casts behandelt?

- Werden Felder, nicht-öffentliche Methoden oder Konstruktoren in die Analyse einbezogen?

- Sollen Typen nur für ausgewählte Deklarationselemente oder für das ganze Projekt inferiert werden? Soll dabei ein gemeinsamer Typ für mehrere Deklarationselemente bestimmt werden?

- Wie wird mit leeren benötigten Protokollen umgegangen?

- Soll eine Generalisierung oder eine Spezialisierung durchgeführt werden?

Folgende neue Refactorings sind auf Basis von ITcore in Planung [Kegel & Steimann 2007]:

- „Replace Inheritance With Forwarding“: Erbt eine Klasse eine große Zahl von Methoden, von denen nur wenige benötigt werden, ist ihr Protokoll unnötig groß. In diesem Fall wäre es geeigneter, eine neue Klasse zu definieren, die nur die erforderlichen Methoden enthält und die Aufrufe an eine Instanz der ehemaligen Superklasse weiterleitet. Das benötigte Protokoll kann mit dem vorgestellten Algorithmus bestimmt werden. Für die Einführung des neuen Typs müsste der Algorithmus allerdings abgeändert werden, da er darauf aufbaut, dass alle Typen durch Supertypen ersetzt werden. Auch Konstruktoren müssen für dieses Refactoring geändert werden. Beides kann aber weiter auf Basis der erzeugten Typconstraints erfolgen.

- „Replace Inheritance With Delegation“: Eine Erweiterung des obigen Refactorings ist notwendig, wenn die Superklasse eigene Methoden aufruft, die von der ausgewählten Klasse überschrieben werden. In diesem Fall muss die Weiterleitung durch Delegation ersetzt werden (siehe auch [Stein 1987]). Die Berechnungen von ITcore erfolgen wie bei „Replace Inheritance With Forwarding“.

- „Inject Dependency“: Diese Anwendung wurde bereits in Abschnitt 6.3 besprochen. Die Analyse der benötigten Methoden und das Einführen des neuen Typs kann mit dem vorliegenden Algorithmus durchgeführt werden. Durch die oben genannten Erweiterungen werden auch die zugehörigen Konstruktoraufrufe bestimmt, die nötige manuelle Nacharbeit könnte für ausgewählte Frameworks dann durch ein eigenes Refactoring automatisiert werden.

- „Create Mock Object“: Für Unit-Tests werden häufig sogenannte „Mock Objects“ benötigt, die gegen Instanzen der eigentlichen Klassen ausgetauscht werden, um Tests zu isolieren und ein kontrolliertes Verhalten anderer Einheiten zu simulieren. Diese Mock-Klassen müssen dasselbe Protokoll wie die ursprünglichen Klassen bereitstellen. Ausgehend von einem Testfall kann das benötigte Protokoll berechnet werden, eine maximal verallgemeinerte Superklasse eingeführt werden und die Erzeugung des Gerüsts einer minimalen Mock-Klasse automatisiert werden. Für die beiden ersten Punkte ist der aktuelle Algorithmus direkt geeignet. Die zusätzlichen Arbeitsschritte können von einem eigenen Refactoring separat durchgeführt werden.

Neben diesen Refactorings können auch andere Werkzeuge für die interface-basierte Programmierung von ITcore Gebrauch machen. Hier sind insbesondere der Type Access Analyser [Steimann & Mayer 2007] und der darauf aufbauende Interface-Designer [Fiedler 2007] zu nennen. Beide Werkzeuge basieren momentan auf der Vorläuferversion von „Infer Type“.

8.3 Fazit

Durch diese Arbeit steht ein wichtiges Werkzeug für die interface-basierte Programmierung mit Java 5 zur Verfügung, das zudem Basis weiterer innovativer Refactorings werden kann. Die Unterstützung von Java 5 ist nicht nur im Vergleich mit der Vorläuferversion neuartig, sondern geht auch deutlich über die der Generalisierungsrefactorings von Eclipse und IntelliJ IDEA hinaus. Auch diese Refactorings könnten deshalb von dieser Arbeit profitieren.

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

Bach 2007

BACH, Markus: *Design und Implementierung eines Eclipse-Plug-Ins zur Anzeige von möglichen Typgeneralisierungen im Quelltext*, FernUniversität in Hagen, Masterarbeit, 2007

Beck & Andres 2004

BECK, Kent ; ANDRES, Cynthia: *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. – ISBN 0–321–27865–8

Bracha 2004

BRACHA, Gilad: Generics in the Java Programming Language. (2004). `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`

Brandstädt 1992

BRANDSTÄDT, Andreas: *Kurs 01685: Effiziente Graphenalgorithmen*. 01685-6-03-S1. FernUniversität in Hagen, 1992

Clayberg & Rubel 2006

CLAYBERG, Eric ; RUBEL, Dan: *Eclipse: building commercial-quality plug-ins*. Addison-Wesley, 2006. – ISBN 0–321–42672–X

Dicky et al. 1996

DICKY, Hervé ; DONY, Christophe ; HUCHARD, Marianne ; LIBOUREL, Thérèse: On Automatic Class Insertion with Overloading. In: *Proceedings of OOPSLA*, 1996, S. 251–267

Fiedler 2007

FIEDLER, Frank: *Ein Eclipse-Framework zur automatischen Bestimmung nützlicher Interfaces in Java-Programmen*, FernUniversität in Hagen, Masterarbeit, 2007

Fowler 1999

FOWLER, Martin: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. – ISBN 0–201–8567–2

Fowler 2004

FOWLER, Martin: *Inversion of Control Containers and the Dependency In-*

jection pattern. http://www.martinfowler.com/articles/injection.html. Version: 23. Januar 2004

Fuhrer et al. 2005

FUHRER, Robert ; TIP, Frank ; KIEŻUN, Adam ; DOLBY, Julian ; KELLER, Markus: Efficiently refactoring Java applications to use generic libraries. In: *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, 2005, S. 71–96

Gamma & Beck 2004

GAMMA, Erich ; BECK, Kent: *Contributing to Eclipse: principles, patterns, and plug-ins*. Addison-Wesley, 2004. – ISBN 0–321–20575–8

Gamma et al. 1994

GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series, 1994. – ISBN 0–201–63361–2

Gosling et al. 2005

GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *Java Language Specification*. 3. Auflage. Addison-Wesley Professional, 2005. – ISBN 0–321–24678–0

Jetbrains s.r.o. 2007

JETBRAINS S.R.O.: *Intellij IDEA 6.0*. http://www.jetbrains.com/idea/. Version: 16. Juni 2007

Kegel & Steimann 2007

KEGEL, Hannes ; STEIMANN, Friedrich: *ITcore: A Type Inference Package for Refactoring Tools*. Wird veröffentlicht für: 1st Workshop on Refactoring Tools, ECOOP, 2007

Khedker et al. 2003

KHEDKER, Uday P. ; DHAMDHERE, Dhananjay M. ; MYCROFT, Alan: Bidirectional data flow analysis for type inferencing. In: *Computer Languages, Systems & Structures* 29:1-2 (2003), S. 15–44

Lindholm & Yellin 1999

LINDHOLM, Tim ; YELLIN, Frank: *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, 1999. – ISBN 0–201–43294–3

Microsoft 2005

MICROSOFT: *C# Version 3.0 Specification*. http://msdn.microsoft.com. Version: Oktober 2005

Nielson et al. 2005

NIELSON, Flemming ; NIELSON, Hanne R. ; HANKIN, Chris: *Principles of Program Analysis*. Springer, 2005. – ISBN 3–540–65410–0

Palsberg & Schwartzbach 1991

PALSBERG, Jens ; SCHWARTZBACH, Michael I.: Object-oriented type inference. In: *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, ACM Press, 1991. – ISBN 0–201–55417–8, S. 146–161

Palsberg & Schwartzbach 1994

PALSBERG, Jens ; SCHWARTZBACH, Michael I.: *Object-oriented type systems*. John Wiley and Sons Ltd., 1994. – ISBN 0–471–94128–X

Snelting & Tip 2000

SNELTING, Gregor ; TIP, Frank: Understanding class hierarchies using concept analysis. In: *ACM Trans. Program. Lang. Syst.* 22 (2000), Nr. 3, S. 540–582. – ISSN 0164–0925

Steimann 2001

STEIMANN, Friedrich: Role = Interface: a merger of concepts. In: *Journal of Object-Oriented Programming* 14:4 (2001), Februar, S. 23–32

Steimann 2006

STEIMANN, Friedrich: *Kurs 01853: Moderne Programmiertechniken und -methoden*. 01853-8-01-S1. FernUniversität in Hagen, 2006

Steimann 2007

STEIMANN, Friedrich: The Infer Type Refactoring and its Use for Interface-Based Programming. In: *Journal of Object Technology* 6:2, Special Issue OOPS Track at SAC 2006 (2007), Februar, S. 67–89. `http://www.jot.fm/issues/issue_2007_02/article5`

Steimann & Mayer 2005

STEIMANN, Friedrich ; MAYER, Philip: Patterns of Interface-Based Programming. In: *Journal of Object Technology* 4:5 (2005), July–August, S. 67–89. `http://www.jot.fm/issues/issue_2007_02/article5`

Steimann & Mayer 2007

STEIMANN, Friedrich ; MAYER, Philip: *Type Access Analysis: Towards Informed Interface Design*. Wird veröffentlicht für: TOOLS EUROPE, 2007

Steimann et al. 2006

STEIMANN, Friedrich ; MAYER, Philip ; MEISSNER, Andreas: Decoupling classes with

inferred interfaces. In: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, ACM Press, 2006. – ISBN 1–59593–108–2, S. 1404–1408

Steimann et al. 2003

STEIMANN, Friedrich ; SIBERSKI, Wolf ; KÜHNE, Thomas: Towards the systematic use of interfaces in JAVA programming. In: *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, Computer Science Press, Inc., 2003. – ISBN 0–9544145–1–9, S. 13–17

Stein 1987

STEIN, Lynn A.: Delegation is inheritance. In: *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, 1987. – ISBN 0–89791–247–0, S. 138–146

Streckenbach & Snelting 2004

STRECKENBACH, Mirko ; SNELTING, Gregor: Refactoring class hierarchies with KABA. In: *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, 2004. – ISBN 1–58113–831–9, S. 315–330

Tip et al. 2003

TIP, Frank ; KIEŻUN, Adam ; BÄUMER, Dirk: Refactoring for generalization using type constraints. In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, 2003, S. 13–26

Turau 2004

TURAU, Volker: *Algorithmische Graphentheorie*. 2. Auflage. Oldenbourg, 2004. – ISBN 3–486–20038–0

Wang & Smith 2001

WANG, Tiejun ; SMITH, Scott F.: Precise Constraint-Based Type Inference for Java. In: *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK : Springer-Verlag, 2001. – ISBN 3–540–42206–4, S. 99–117

Widmer 2006

WIDMER, Tobias: Unleashing the Power of Refactoring. In: *Eclipse Magazine* Ausgabe 1 (2006). `http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html`

Inhalt der beiliegenden CD

Verzeichnis	Inhalt
dist	Das „Infer Type“-Plugin als JAR-Datei.
src	Die Quelltexte und die Eclipse-Projektdateien des „Infer Type“-Refactorings.
javadoc	Die Dokumentation der Quellen in HTML-Format.
multirefact	Das Plugin, mit dem die in Abschnitt 6.4 beschriebenen automatisierten Tests durchgeführt wurden.
tests	Alle Projekte, auf denen diese Testläufe durchgeführt wurden.
samples	Die Quellen für alle Beispiele, die in dieser Arbeit aufgeführt wurden.
eclipse	Eine Installation von Eclipse 3.2 mit „Infer Type“ für Windows.

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

München, den 29. Juni 2007

Hannes Kegel