Implementierung eines Typunifikationsalgorithmus für Java 8

Studienarbeit T3201

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart Campus Horb

von

Florian Steurer

Juni 2015

Bearbeitungszeitraum	12 Monate
Matrikelnummer, Kurs	5525421, INF2013
Betreuer	Prof. Dr. rer. nat. Martin Plümicke, Andreas Stadelmeier
Gutachter	Prof. Dr. rer. nat. Martin Plümicke

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich, gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2015, die vorliegende T3201 Studienarbeit mit dem Titel *Implementierung eines Typunifikationsalgorithmus für Java 8* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Horb am Neckar, 25. April 2016

Abstract

Java is a programming language where it is necessary to declare types explicitly. With type inference, type declarations can be omitted, making programs shorter and easier to write. In [Plü15b], a type inference algorithm for Java is introduced. This algorithm uses a constraint-based type unication, which, for a given set of type constraints, computes all possible typings of a program [Plü07]. The type unification algorithm has been implemented as part of a compiler [PB06], which supports implicitly typed Java programs.

In [Plü15a] the Java type unification is extended by real function types. Additionally, some cases concerning wildcard types are added, that were not considered in [Plü07]. This leads to a simplification of the algorithm.

The old implementation is too inefficient to handle inputs of relevant size. A complete reimplementation aims to improve efficiency by using immutable data structures.

This paper describes the reimplementation of the type unification algorithm as well as the modification of the algorithm to handle the missing cases.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Ausgangssituation

In Java müssen bisher alle Typen explizit deklariert werden. Mit Typinferenz ist es möglich Typdeklarationen auszulassen, sodass die Typen automatisch inferiert werden. Durch inferierte Typen werden Programme kürzer und einfacher zu schreiben. Ein Typinferenzalgorithmus für Java wurde in [Plü15b] vorgestellt. Der Typinferenzalgorithmus benutzt ein Constraint-basierten Typunifikationalgorithmus [Plü07], welcher, für eine Menge an Constraints alle möglichen Typisierungen eines Java-Programms berechnet.

In [PB06] wurde dieser Algorithmus implementiert. Die Implementierung wird in einem Compiler genutzt, welcher implizit typisierte Java-Programme unterstützt, indem er Typen inferiert. Die Implementierung des Typunifikationsalgorithmus ist allerdings zu ineffizient um Eingaben von relevanter Größe unifizieren zu können. Das hat zur Folge, dass nur kleine Java-Programme in einer angemessenen Zeit kompiliert werden können.

1.2 Thema der Arbeit

Durch eine Neuimplementierung des Typunifikationsalgorithmus soll die Effizienz des Algorithmus verbessert werden, sodass Typen auch in größeren Java-Programmen inferiert werden können. Der Algorithmus soll um die in [Plü15a] beschriebenen echten Funktionstypen für Java erweitert werden. Dabei soll auch auf eine softwaretechnisch gute Implementierung geachtet werden, die erweiterbar, anpassbar und verständlich ist. Im Rahmen dieser Arbeit wird der Algorithmus außerdem um bisher nicht betrachtete Fälle erweitert, aufgrund derer bestimmte Lösungen bisher nicht gefunden wurde.

1.3 Aufbau

Kapitel 2 beschäftigt sich mit den Grundlagen auf denen diese Arbeit aufbaut. Das Kapitel beginnt mit einem Überblick zu den Themen Typtheorie und Unifikation. Anschließend wird das Java-Typsystem vorgestellt. Zum Schluss werden für die Implementierung relevante objektorientierte Entwurfsmuster vorgestellt.

In Kapitel 3 wird der Typunifikationsalgorithmus auf einer formellen und konzeptionellen Ebene behandelt. Es werden Änderungen am Algorithmus aus [Plü07] beschrieben, die den Algorithmus um bisher fehlende Fälle erweitern .

Kapitel 4 geht auf die Implementierung des Typunifkationsalgorithmus ein. Anhand von Unified Modelling Language (UML) und Codebeispielen werden der softwaretechnische Entwurf sowie konkrete Aspekte der Umsetzung erläutert.

Abschließend folgt mit Kapitel 5 ein Fazit und ein Ausblick auf künftige Entwicklungen.

Kapitel 2

Grundlagen

Dieses Kapitel beginnt mit Grundlagen zur Typen und Typtheorie. Anschließend folgen in Abschnitt 2.2 Informationen zur Unifikation.

Das Java-Typsystem bildet den Kontext indem diese Arbeit durchgeführt, daher wird es in Abschnitt 2.3 beschrieben.

Abschließend werden in Abschnitt 2.4 noch die Entwurfsmuster Visitor und Strategie beschrieben, welche in der Implementierung des Algorithmus zum Einsatz kommen.

2.1 Typtheorie

2.1.1 Typen

Ein Typ ist eine Menge an Werten, die ein Ausdruck oder eine Variable zur Laufzeit eines Programmes annehmen kann [Car96]. So kann man z.B. $\mathbb{Z} = \{0, 1, -1, 2, -2, 3, -3, \ldots\}$, also die Menge der ganzen Zahlen, auch als Typ betrachten. In vielen Programmiersprachen gibt es einen Typ, der diesem Typ $\mathbb{Z}$ entspricht wie z.B. die Klasse `BigInteger` in Java.

Ein Typ ist dabei stets nur eine Annäherung an die tatsächlichen Werte, die eine Variable oder ein Ausdruck zur Laufzeit annehmen kann [Car96]. Zum Beispiel kann eine Methode mit dem Rückgabetyp $\mathbb{Z}$ stets den Wert „2“ zurückgeben. Der Typ des Methodenaufrufs ist damit zwar eine Ganzzahl, jedoch wird zur Laufzeit nur ein konkreter Wert aus unendlich vielen Möglichkeiten angenommen.

2.1.2 Typsysteme

Ziele

Bei der Ausführung von Programmcode auf einem Rechner können verschiedene Fehler auftreten. Das Hauptziel eines Typsystems ist es, sogenannte Laufzeit-Typfehler zu vermeiden [Pie02]. Diese Art von Fehlern hat nicht zwingend den Abbruch der Programmausführung zur Folge, sondern kann auch unbemerkt bleiben und später zu undefiniertem Verhalten des Programms führen [Car96]. Welche Fehler genau vom Typsystem verhindert werden, ist für das jeweilige System individuell definiert [Pie02].

Ein gängiger Laufzeit-Typfehler der von Typsystemen vermieden wird, ist z.B. der Zugriff auf eine unpassende, aber valide Speicheradresse [Car96]. Ließe z.B. ein Typsystem den Aufruf beliebiger Methoden auf beliebigen Objekten zu, so könnte durch Aufruf einer nicht-vorhandenen Methode unerwünschter Code ausgeführt werden der an der Stelle im Speicher liegt, an der die Methode erwartet wurde.

Durch die Begrenzung der möglichen Werten eines Ausdrucks, stellen Typen Einschränkungen dar, die helfen die Korrektheit eines Programmes sicherzustellen [CW85]. Die Einführung von Typen kann außerdem zu besserem Code führen, indem die Abstraktion

gefördert wird (z.B. durch Interfaces in der Objektorientierung) und Verständlichkeit und Fehlermeldungen verbessert werden. Einem Compiler stehen durch Typen zusätzliche Informationen zur Verfügung, die dieser für die Erzeugung von effizienterem Code nutzen kann. [Pie02]

Definition 2.1 (Typkorrektheit)**.** *Die Eigenschaft eines Programmes, dass bei der Ausführung keine Laufzeit-Typfehler können [Car96].*

Definition 2.2 (Typsystem)**.** *In der Informatik ist ein Typsystem ein Satz an Regeln für Programmiersprachen, welcher die Typkorrektheit sicherstellt.*

Statische und dynamische Typisierung

Programmiersprachen können anhand der Umsetzung ihrer Typsystemen unterschieden werden. Zunächst wird zwischen statischer und dynamischer Überprüfung der Typisierung unterschieden. Bei der statischen Überprüfung ist jeder Typ zur Compilezeit bekannt. Durch einen sogenannten Typchecker wird noch vor der Ausführung des Programmes sichergestellt, dass sich das Programm typkorrekt verhält [CW85]. Bei der dynamischen Überprüfung wird typkorrektes Verhalten erst zur Laufzeit sichergestellt, indem Speicherobjekte und Operationen vor der Ausführung auf Typkompatiblität geprüft werden. Wird diese Überprüfung nicht bestanden, wird die Ausführung des Programmes abgebrochen [Car96].

Starke und schwache Typisierung

Eine weitere Unterscheidung kann zwischen stark und schwach typisierten Sprachen getroffen werden. Stark typisierte Sprachen können garantieren, dass keine Typfehler zur Laufzeit auftreten [Car96]. Die Typisierung von stark typisierten Sprachen muss daher statisch sein, um Laufzeitfehler im Voraus ausschließen zu können [Pie02]. In schwach typisierten Sprachen können während der Ausführung Fehler auftreten. Das Typsystem kann die Typfehlerfreiheit des Programms also nicht vollständig garantieren.

Implizite und explizite Typisierung

Weiterhin kann zwischen explizit und implizit typisierten Sprachen unterschieden werden. In explizit typisierten Sprachen werden die Typen in der Programmsyntax vom Programmierer angegeben. Implizite Typisierung ermöglicht das Weglassen von Typen im Programmcode. Bei impliziter Typisierung werden die Typen vom Compiler berechnet (vgl. dazu Abschnitt 2.1.6). [Car96]

Umsetzung

In der Praxis realisieren Programmiersprachen und ihre Typsysteme diese Eigenschaften in einem unterschiedlichen Grad. So ist Java z.B. im Wesentlichen eine statisch, stark und explizit typisierte Programmiersprache. Dennoch existieren auch Merkmale der dynamischen Überprüfung (Korrektheit von Array-Indizes), der schwachen Typisierung (der *Cast*-Operator) und der impliziten Typisierung (Diamant-Operator <>). Das Java-Typsystem wird in Abschnitt 2.3 noch genauer beschrieben.

2.1.3 Polymorphie

Polymorphie ist die Eigenschaft von Instanzen, Variablen oder Funktionsparametern mehr als einen Typ zu haben. Dies steht im Gegensatz zu monomorphen Typsystemen in denen alles genau einen Typ besitzt. [CW85]

Abbildung 2.1 zeigt die verschiedenen Arten von Polymorphie.

Universelle Polymorphie

Von universeller Polymorphie spricht man, wenn eine Funktion (bzw. Methode oder Prozedur) für verschiedene Typen gleich funktioniert. Die Funktion also ein einziges mal programmiert und für verschiedene Typen genutzt werden.

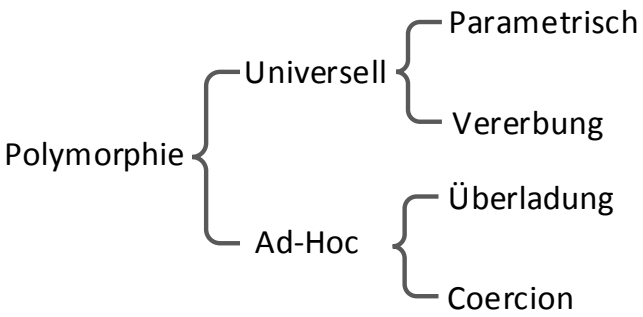

Abbildung 2.1: Arten von Polymorphie
Quelle: [CW85]

Parametrische Polymorphie

Universell Polymorphie lässt sich erreichen, indem eine Funktion einen (expliziten oder impliziten) Typparameter erhält. Dann spricht man von parametrischer Polymorphie und einer generischen Funktion. [CW85]

Vererbung

Eine weitere Möglichkeit universell polymorphe Funktionen zu programmieren, ist die Nutzung von Vererbung. Vererbung ermöglicht es, dass eine Instanz mehrere, sich überschneidende Typen haben kann [CW85]. Somit kann eine Funktion für einen Obertyp definiert und dann auch für alle Untertypen genutzt werden.

Ad-Hoc-Polymorphie

Bei Ad-Hoc-Polymorphie ist ebenfalls eine Funktion für mehrere Typen definiert. Anders als bei universeller Polymorphie hat diese aber nicht das gleiche Verhalten für alle Typen, sondern kann sich aber in ihrem Verhalten je nach Typ unterscheiden.

Überladung

Ad-Hoc-Polymorphie kann z.B. durch Überladung von Funktionen erreicht werden [CW85]. Bei der Überladung einer Funktion $f(x)$, hat diese mehrere unterschiedliche Implementie-

rungen $f_1(x : \theta_1)$ bis $f_n(x : \theta_n)$ wobei $x : \theta_i$ bedeutet, dass x vom Typ θ_i ist. Beim Aufruf wird je nach Typ θ_i des Parameters x die entsprechende Implementierung gewählt.

Coercion

Die zweite Ausprägung der Ad-Hoc-Polymorphie, das *Coercion* ist eine implizite Typumwandlung der Aufrufparameter einer Funktion und äußerlich (als Aufrufender) somit nicht von einer überladenen Funktion zu unterscheiden [CW85]. Gäbe es im obigen Beispiel noch einen Typ θ_{n+1} mit einer Typumwandlung von θ_1 nach θ_{n+1}, so wäre mit *Coercion* der Aufruf $f(x : \theta_{n+1})$ möglich. Von außen ist dies nicht von einer Überladung $f_{n+1}(x : \theta_{n+1})$ zu unterscheiden.

2.1.4 Typurteil

Ein Ausdruck ist ein Teil eines Programmes wie eine Variable, Konstante oder ein Funktionsaufruf. Ausdrücke können zu einem Wert evaluiert (ausgewertet) werden. Ob ein Ausdruck valide ist, wird durch die Syntax (Grammatik) und das Typsystem einer Programmiersprache festgelegt. Ein Ausdruck der in fast allen Programmiersprachen vorkommt ist z.B. die Addition von zwei Zahlen.

Definition 2.3 (Typannahme)**.** *Eine Typannahme $x_1 : \theta_1$ ordnet einen Ausdruck x_1 den Typ θ_1 zu. Sei z.B. $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ die Additionsfunktion für Ganzzahlen. Die Typannahme $+ : \mathbb{Z}$ sagt aus, dass ein Aufruf dieser Funktion stets ein Element vom Typ $\mathbb{Z}$ als Ergebnis hat.*

Definition 2.4 (Typurteil)**.** *Sei $\Gamma = \{x_n : \theta_n, \ldots, x_n : \theta_n\}$ eine Menge von Typannahmen. Dann ist $\Gamma \vdash M : A$ ein Typurteil. Ein Typurteil drückt aus, dass der Ausdruck M unter den Typannahmen Γ den Typ A hat. Ein Typurteil kann valide oder invalide sein.*

2.1.5 Typregeln

Typregeln sind Teil eines formalen Typsystems und erlauben die Validierung und Herleitung von Typurteilen. Eine Typregel besteht aus Prämissen oberhalb der Linie und einer Konklusion unterhalb. [Car96]

$$\text{(Allgemeine Typregel)}\ \frac{\Gamma_1 \vdash x_1 : \theta_1 \qquad . \quad . \quad . \qquad \Gamma_n \vdash x_n : \theta_n}{\Gamma \vdash x : \theta}$$

Die Typregel in 2.1.5 sagt aus, dass wenn die Ausdrücke x_1 bis x_n vom Typ θ_1 bis θ_n sind, der Ausdruck x vom Typ θ ist.

Beispiel 2.1.1 (Anwendung von Typregeln)**.** Sei ein Typsystem mit folgenden Typregeln gegeben.

$$\text{(Tautologie)}\ \frac{\Gamma \cup \{x : \theta\} \vdash x : \theta}{\Gamma \cup \{x : \theta\} \vdash x : \theta}$$

$$\text{(Addition)}\ \frac{\Gamma \vdash x_1 : \mathbb{Z} \qquad \Gamma \vdash x_2 : \mathbb{Z}}{\Gamma \vdash x_1 + x_2 : \mathbb{Z}}$$

Sei $\Gamma = \{0 : \mathbb{Z},\ 1 : \mathbb{Z},\ -1 : \mathbb{Z}, \ldots\}$ und der zu typisierende Ausdruck sei $5 + 6$. Durch Anwendung der Addition- und Tautologie-Regel lässt sich ein valider Ableitungsbaum konstruieren, der den Typ von $5 + 6$ als $\mathbb{Z}$ ausweist.

$$\dfrac{\dfrac{\text{(Tautologie)}\ \dfrac{\Gamma \cup \{5 : \mathbb{Z}\} \vdash 5 : \mathbb{Z}}{\Gamma \vdash 5 : \mathbb{Z}} \qquad \text{(Tautologie)}\ \dfrac{\Gamma \cup \{6 : \mathbb{Z}\} \vdash 6 : \mathbb{Z}}{\Gamma \vdash 6 : \mathbb{Z}}}{\text{(Addition)}}}{\Gamma \vdash 5 + 6 : \mathbb{Z}}$$

2.1.6 Typinferenz

In typisierten Programmiersprachen besitzt jeder Ausdruck (und Unterausdruck) einen Typ. Ein Programmierer muss Typinformationen jedoch nur an kritischen Stellen angeben, sodass weitere Typen automatisch hergeleitet werden können [CW85]. Typinferenz bezeichnet diese Suche nach Typen, die sich durch Anwendung der Typregeln herleiten lassen [Car96].

Typinferenz leitet Typen meist nach dem Bottom-Up-Prinzip her [CW85], beginnt also mit den elementarsten Unterausdrücken und leitet aus diesen die Typen der zusammengesetzten Ausdrücke her. Implizit typisierte Sprachen wie Haskell ermöglichen es dem Programmierer weite Teile von Typinformationen wegzulassen. Diese werden dann vom Compiler inferiert.

Die Funktion $square\ x = x * x$ kann z.B. ohne Typinformationen programmiert werden. Der Typinferenz-Algorithmus nimmt dabei den Typ des Parameters x zunächst als eine Typvariable an. Die Typvariable wird zu einem späteren Zeitpunkt mithilfe der Unifikation (vgl. Abschnitt 2.2) instanziiert [CW85]. Der Typ von x wird dadurch beschränkt, dass der $*$-Operator für diesen Typ definiert sein muss.

2.2 Unifikation

Die Unifikation findet in der Typinferenz Anwendung. Sie wird z.B. im Algorithmus W der Hindley-Milner-Typinferenz verwendet um den allgemeinsten Typ einer Funktionsapplikation zu finden.

Unifikation lässt sich beschreiben als Gleichmachung zweier symbolischer Ausdrücke (Terme) durch Ersetzen bestimmter Unterausdrücke (Variablen) [RV01].

Unifikation wurde als Erstes von J.A. Robinson in [Rob65] beschrieben. Der dort vorgestellte Algorithmus zur Lösung des Unifikationsproblems hat im Worst-Case allerdings eine exponentielle Laufzeit. Algorithmen mit linearer Laufzeit wurden unter anderem von M. Paterson und M. Wegman in [PW76] und A. Martelli und U.Monatanari in [MM82] beschrieben.

Im Folgenden werden einige Begriffe zur Unifikation definiert. Eine vollständige formale Definition und Erläuterungen zur Unifikation finden sich unter anderem in [RV01].

Definition 2.5 (Substitution)**.** *Eine Substitution* $\sigma = \{(x_1 \to y_1), \ldots, (x_n \to y_n)\}$ *ist eine Funktion welche, auf einen Ausdruck angewandt werden kann und alle Variablen* x_1 *bis* x_n *durch die zugeordneten Ausdrücke* y_1 *bis* y_n *ersetzt.*

Definition 2.6 (Unifikator)**.** *Eine Substitution* σ *ist ein Unifikator für zwei Ausdrücke* t_1*,* t_2*, wen gilt* $\sigma(t_1) = \sigma(t_2)$*.*

Definition 2.7 (Allgemeinster Unifikator)**.** *Ein Unifikator* σ *ist der allgemeinste Unifikator oder auch Most-General-Unifier (MGU), wenn es für jeden weiteren Unifikator* τ *eine Subsitution* ρ *gibt, sodass* $\tau = \rho \cup \sigma$ *[RV01]. D.h. von allen Möglichkeiten die zwei Terme zu unifizieren, ersetzt der MGU am wenigsten Variablen. Der MGU ist eindeutig mit Ausnahme von Umbenennungen von Variablen [RV01].*

Definition 2.8 (Unifikationsalgorithmus)**.** *Ein Unifikationsalgorithmus ist ein Algorithmus, welcher als Eingabe zwei oder mehrere Terme erhält und für diese Terme den allgemeinsten Unifikator findet, falls dieser existiert.*

Beispiel 2.2.1 (Unifikation)**.** Seien $t_1 = f(b, g(y, a))$ und $t_2 = f(y, x)$ die zu unifizierenden Terme, wobei $\{x, y\}$ die Menge der Variablen ist, für welche andere Ausdrücke eingesetzt werden können.

Um die Terme zu unifizieren muss y mit b und x mit $g(b, a)$ ersetzt werden.

Der MGU ist also $\sigma = \{y \mapsto b, x \mapsto g(b, a)\}$. In der unifizierten Form $\sigma(t_1)$ und $\sigma(t_2)$ haben die Terme die Form $f(b, g(b, a))$

2.2.1 Martelli-Montanari-Unifikation

Der Martelli-Montanari-Algorithmus löst das Unifikationsproblem in linearer Laufzeit [MM82]. Der Algorithmus besteht aus der wiederholten Anwendung von vier Regeln auf die Elemente der Menge an Eingabetermen E. Die Menge der in E vorkommenden Variablen wird im folgenden als $Vars(E)$ bezeichnet.

$$\text{(reduce)} \quad \frac{E \cup \{f(t_1, \ldots, t_n) \doteq f(s_1, \ldots, s_n)\}}{E \cup \{t_1 \doteq s_1, \ldots, t_n \doteq s_n\}}$$

$$\text{(erase)} \quad \frac{E \cup \{t \doteq t\}}{E}$$

$$\text{(swap)} \quad \frac{E \cup \{t \doteq x\}}{E \cup \{x \doteq t\}} \quad t \notin Vars(E)$$

$$\text{(subst)} \quad \frac{E \cup \{x \doteq t\}}{E[x \mapsto t] \cup \{x \doteq t\}}$$

Haben Gleichungen eine der folgenden Formen, sind sie nicht unifizierbar.

$$f(t_1, \ldots, t_n) \doteq g(s_1, \ldots, s_n) \qquad f \neq g$$

$$f(t_1, \ldots, t_n) \doteq f(t_1, \ldots, t_m) \qquad m \neq n$$

$$x \doteq f(t_1, \ldots t_n) \qquad x \in Vars(f)$$

Enthält die Menge E eine Gleichung dieser Form, oder entsteht eine solche Gleichung während der Anwendung der Regeln, so ist die Unifikation gescheitert. Wenn ein Unifikationsalgorithmus prüft, ob eine Gleichung die dritten, nicht-unifizierbaren Form vorliegt, bezeichnet man dies als Occurs-Check.

2.3 Java

2.3.1 Typsystem

Bei Java handelt es sich um eine statisch typisierte Sprache [Gos+15]. In statisch typisierten Sprachen wird zur Compilezeit ein Typcheck durchgeführt (vgl. Abschnitt 2.1.2). Java ist laut Spezifikation auch eine stark typisierte Sprache [Gos+15]. Starke Typisierung verhindert das Auftreten von Laufzeit-Typfehlern.

Tatsächlich bietet das Java-Typsystem aber Ausnahmen, mit denen sich die starke Typisierung umgehen lässt. Eine solche Ausnahme wird in Codebeispiel 2.1 gezeigt. Java-Arrays sind kovariant, d.h. ein Array A mit Elementen vom Typ a und ein Array B mit Elementen vom Typ b stehen zueinander in derselben Subtyp-Beziehung ($\leq^*$), wie ihre Elemente a und b. Das bedeutet da $String \leq^* Object$ gilt auch $String[] \leq^* Object[]$. Das Codebeispiel nutzt diese Kovarianz um den Typcheck zur Compilezeit zu umgehen und einen Laufzeit-Fehler zu erzeugen.

```
1 String[] strings = new String[1];
2 // Kovariante Array Zuweisung
3 Object[] objects = strings;
4 // Wirft eine ArrayStoreException zur Laufzeit,
5 // da ein Integer in ein String Array gespeichert werden soll.
6 objects[0] = new Integer(1);
```

Codebeispiel 2.1: Typunsicherheit bei Java-Arrays

2.3.2 Referenztypen

In Java gibt es vier verschiedene Arten von Referenztypen: Klassen, Interfaces, Typvariablen und Arrays [Gos+15]. Instanzen von Referenztypen heißen Objekte. Objekte werden in einem Java-Programm nicht direkt, sondern über Referenzen angesprochen. Eine Referenz ist ein automatisch verwalteter Zeiger auf die Speicherstelle des Objektes und lässt keine Zeiger-Arithmetik zulässt. Bei Zuweisungen und Parameterübergaben werden also nicht die Objekte selbst, sondern Referenzen auf diese Objekte kopiert. Änderungen am Zustand eines Objektes werden durch einen Zugriff über eine Referenz durchgeführt und betreffen somit alle Teile des Programmes die ebenfalls eine Referenz auf dieses Objekt halten.

2.3.3 Generische Typen

Generische Typen oder auch parametrisierte Typen sind Typen mit einem oder mehreren Typparametern. Nur Klassen oder Interfaces können generisch sein [Gos+15]. Durch generische Typen ist es möglich, Logik unabhängig von den konkreten Typparametern zu implementieren, und so wiederverwendbaren Code zu erzeugen.

Ein bekanntes Beispiel für generische Typen ist das Interface `Collection<T>` und seine Unterklassen. Diese stellen Datenstrukturen bereit, welche für beliebige Typen `T` funktionieren.

2.3.4 Wildcardtypen

Generische Typen sind zunächst invariant, d.h. z.B. eine `List<Integer>` ist kein Untertyp einer `List<Number>` obwohl jedes Element der ersten Liste auch den Typ `Number` hat und somit von der zweiten Liste aufgenommen werden könnte.

Kovarianz und Kontravarianz kann in generischen Typen durch Wildcard-Typen erzeugt werden. In Java gibt es zwei Arten von Wildcard-Typen: `? extends T` und `? super T`.

Die Extends-Wildcard stellt Kovarianz her, d.h. umschließende generische Typen stehen zueinander in der selben Subtypbeziehung wie ihre Argumente. Somit wird eine `List<? extends Number>` zum Supertyp einer `List<? extends Integer` und auch einer `List<Integer>` und `List<Number>`.

Die Super-Wildcard stellt Kontravarianz her, bei der umschließende generische Typen sich entgegengesetzt zur Subtypbeziehung ihrer Argumente verhalten. Somit wird eine `List<? super Integer>` zum Supertyp einer `List<? super Number` und auch einer `List<Integer>` und `List<Number>`.

2.3.5 Primitive Typen

Primitive Typen beschreiben atomare Werte z.B. eine Ganzzahl. In Java gibt es acht vordefinierte primitive Typen. Diese sind `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`

und `double`. Der Programmierer kann keine neuen primitiven Typen selbst definieren [Gos+15].

Primitiven Typen teilen sich keinen Zustand [Gos+15], d.h. bei jeder Variablenzuweisung oder Parameterübergabe wird eine neue Kopie der Instanz erstellt, deren Änderungen keinen Einfluss auf die ursprüngliche Instanz haben.

Jeder primitive Typ hat einen entsprechenden Referenztyp z.B. `Integer` für den primitiven Typ `int`. Die Repräsentation eines primitiven Wertes durch einen Referenztyp ermöglicht z.B. die Verwendung in generischen Typen (vgl. Abschnitt 2.3.3) oder die Zuweisung von `null`. Um die (Rück-) Konvertierung von Primitiven zu Referenztypen zu erleichtern, bietet Java den sogenannten Boxing-Mechanismus. Boxing verpackt primitive Werte automatisch in den entsprechenden Referenztyp. Codebeispiel 2.2 demonstriert diesen Mechanismus anhand einer generischen Liste.

```
1  ArrayList<Boolean> list = new ArrayList<>();
2  list.add(true);       // Boxing boolean zu Boolean
3  boolean b = a.get(0); // Unboxing Boolean zu boolean
```

Codebeispiel 2.2: (Un-)Boxing primitiver Typen

2.3.6 Polymorphie

In Abschnitt 2.1.3 wurden die vier Arten der Polymorphie bereits beschrieben. Diese waren parametrische Polymorphie, Vererbung, Überladung und *Coercion*. Wie Java die vier Arten der Polymorphie unterstützt wird anhand von Beispielen gezeigt.

Das Codebeispiel 2.3 zeigt wie Polymorphie durch Typparameter in Java erreicht werden kann. Die Methode `fillFreeSlot` soll an der ersten Position des Arrays an der `null` steht ein Element einfügen. Durch den Typparameter `T` ist dies unabhängig vom konkreten Typ des Arrays möglich. Die Methode ist generisch. Parametrische Polymorphie tritt auch dann auf, wenn die Parameter einer Methode variante generische Typen sind.

Die Polymorphie durch Vererbung wird in Codebeispiel 2.4 gezeigt. Eine Klasse `Square` erbt von einer Klasse `Rectangle`. Da bei Quadraten alle Seitenlängen gleich sind, wird die Methode `getWidth` überschrieben, so dass sie stets dasselbe Ergebnis wie `getHeight` zurückgibt. Die Methode `area`, die den Flächeninhalt eines `Rectangles` berechnet funktio-

```java
public static <T> void fillFreeSlot(T[] array, T obj) {
    for(int i = 0; i < array.length; i++)
        if(array[i] == null)
            array[i] = obj;
}
```

Codebeispiel 2.3: Parametrische Polymorphie in Java

niert auch für `Squares` und alle weiteren Unterklassen von `Rectangle` die noch hinzugefügt werden könnten.

```java
public class Rectangle {
    public int getHeight() { /* Implementation */ }
    public int getWidth() { /* Implementation */ }
}

public class Square extends Rectangle {
    public int getWidth() { // Override method
        return getHeight();
    }
}

// Works for squares as well
public static double area(Rectangle shape) {
    return s.getWidth() * s.getHeight();
}
```

Codebeispiel 2.4: Polymorphie durch Vererbung in Java

Ein Beispiel für Ad-Hoc-Polymorphie durch Überladung ist in Codebeispiel 2.5 abgebildet. Hier sollen die Flächen von `Circles` und `Rectangles` berechnet werden. Da die Berechnung grundsätzlich unterschiedlich funktioniert, wird die Methode `area` überladen. Bei jedem Aufruf der Methode wird anhand des Typs des Parameters `shape` die passende Implementierung aufgerufen.

```java
public static void main(String[] args) {
    Circle s = new Circle();
    s.Radius = 5;
    area(s); // Correct implementation gets chosen
}

public static double area(Circle shape) {
    return s.Radius * s.Radius * Math.Pi;
}

public static double area(Rectangle shape) {
    return s.Length * s.Height;
}
```

Codebeispiel 2.5: Polymorphie durch Überladung in Java

Die vierte Art der Polymorphie, das `Coercion`, wird in Codebeispiel 2.6 gezeigt. Java

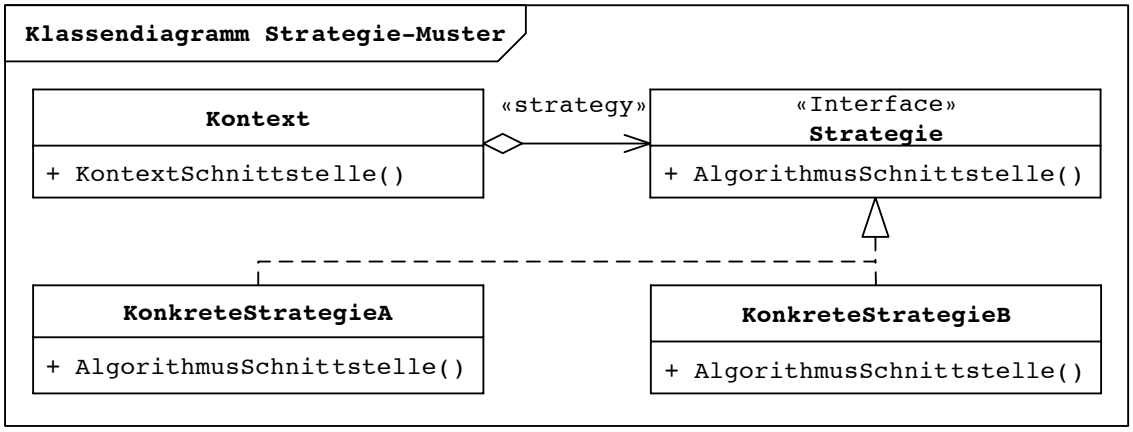

Abbildung 2.2: Klassendiagramm Strategie-Muster
Quelle: Nach [Gam+96]

ermöglicht die implizite Konvertierung von primitiven Typen, sofern diese verlustfrei ist. Außerdem ist Boxing-Konvertierung von primitiven Werten möglich. Die implizite Konvertierung von Referenztypen zu ihren Supertypen ist ebenfalls möglich, dabei handelt es sich allerdings um Polymorphie durch Vererbung.

```
1  public static void main(String[] args) {
2      int i = 4;
3      square(i) // Coercion by implicit conversion
4  }
5
6  public static double square(double d) {
7      return d*d;
8  }
```

Codebeispiel 2.6: Polymorphie durch Coercion in Java

2.4 Entwurfsmuster

2.4.1 Strategie

Das Strategie-Muster macht Algorithmen durch die Definition einer gemeinsamen Schnittstelle austauschbar [Gam+96].

Abbildung 2.2 zeigt den Aufbau des Musters als Klassendiagramm. Ein `Kontext`-Objekt wird mit einer `Strategie` parametrisiert. In der `KontextSchnittstelle()` implementierte Logik ist nun unabhängig von der konkreten Ausprägung der `Strategie`.

Das Strategie-Muster macht in der `KontextSchnittstelle()` implementierte Algorithmen

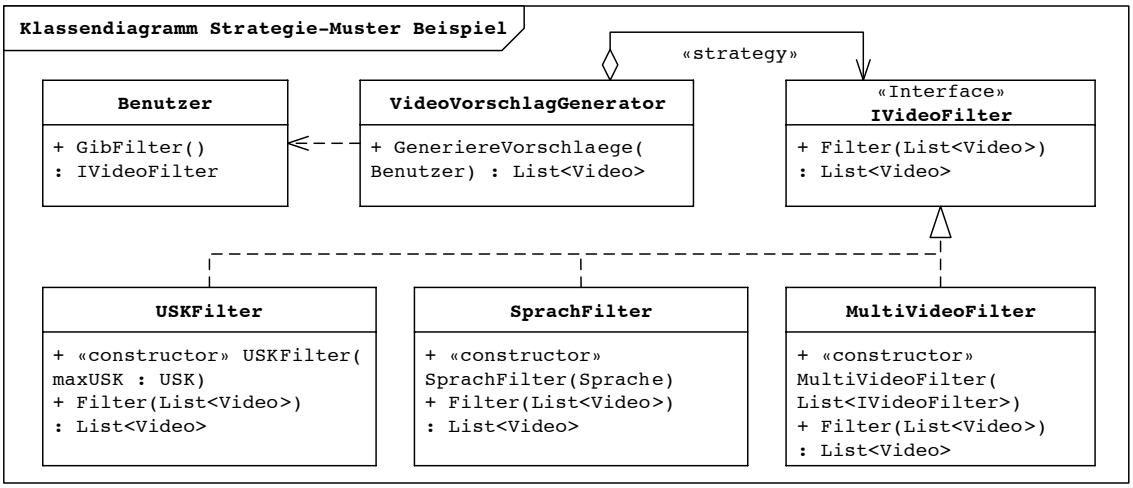

Abbildung 2.3: Klassendiagramm Anwendung des Strategie-Musters

parametrisierbar, indem es den Austausch von Unteralgorithmen (den Strategien) ermöglicht. Anhand von Kriterien wie z.B. dem Speicherplatzverbrauch kann situationsbedingt eine geeignete Strategie gewählt werden [Gam+96]. Durch Implementierung der Schnittstelle können jederzeit neue Strategien erstellt werden. Dadurch kann der Code auch im nach hinein um effizientere oder effektivere Algorithmen erweitert werden.

Eine konkrete Anwendung des Strategie-Musters wird im Klassendiagramm in Abbildung 2.3 gezeigt. Ein Video-Portal möchte basierend auf dem bisherigen Verhalten eines `Benutzers` eine Liste von Videos vorschlagen. Dazu wird ein entsprechender Algorithmus in der Klasse `VideoVorschlagGenerator` implementiert. Es sollen aber nur Videos vorgeschlagen werden, die den Daten und Einstellungen des Benutzers nach geeignet sind. Durch die Verwendung einer `IVideoFilter`-Strategie, kann der Generator eine Menge an Vorschlägen generieren und anschließend mit einer Nutzer-individuellen Strategie filtern. Die genaue Ausprägung der Strategie ist dem Generator dabei nicht bekannt. Im Klassendiagramm werden die Filterstrategien vom `Benutzer`-Objekt bereitgestellt und können Videos nach Altersfreigabe, Sprache oder mehreren Kriterien filtern. Die Verwendung des Strategie-Musters ermöglicht es, im `VideoVorschlagGenerator` allgemeinen Code zu schreiben, welcher alle `Benutzer` gleich behandelt, da die individuellen Bestandteile in die Strategien ausgelagert wurden.

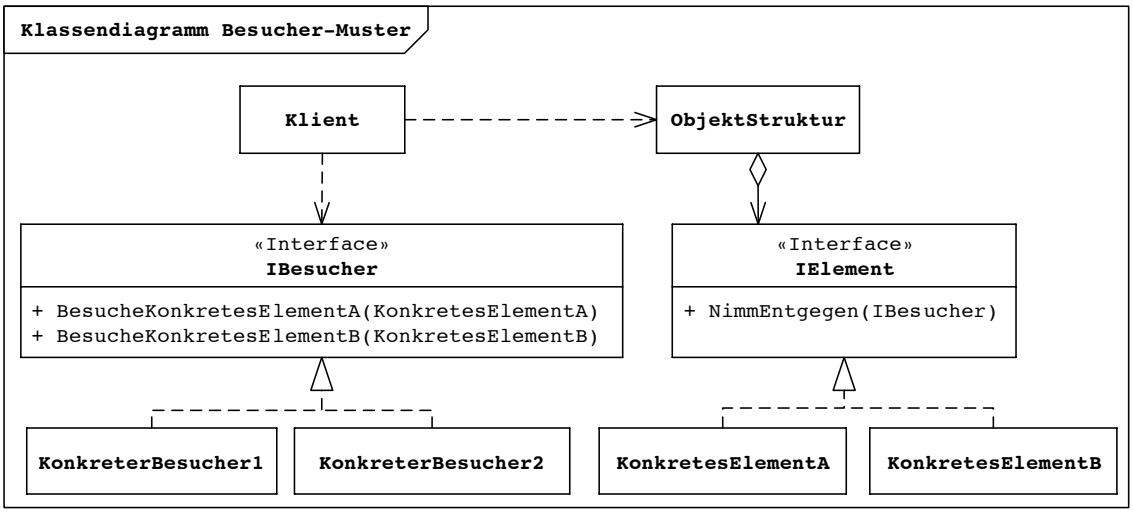

Abbildung 2.4: Klassendiagramm Besucher-Muster
Quelle: Nach [Gam+96]

2.4.2 Visitor

Das Besucher-Muster ermöglicht die Separierung von einer Datenstruktur und Operationen die auf dieser arbeiten, sodass neue Operationen ohne Änderungen an der Datenstruktur hinzugefügt werden können [Gam+96].

Abbildung 2.4 zeigt den Aufbau des Musters als Klassendiagramm. Der `IBesucher` enthält den von der Datenstruktur separierten Algorithmus. Alle Elemente einer `ObjektStruktur` implementieren ein Interface `IElement` welches in der `NimmEntgegen`-Methode einen Besucher akzeptiert. In dieser ruft das Element die entsprechende `Besuche`-Methode des Besuchers mit sich selbst als Argument auf und löst somit die entsprechende Logik im Besucher aus. Durch Implementierung der `IVisitor`-Schnittstelle ist es möglich, jederzeit neue Operationen auf der Datenstruktur hinzuzufügen ohne diese anpassen zu müssen.

In Baum- und Graphenstrukturen muss ein Knoten auch die `NimmEntgegen`-Methode seiner Kinder mit dem Besucher aufrufen. Dies kann entweder in `NimmEntgegen` des Elternknotens oder in einer zusätzlichen Methode geschehen. Durch den zweiten Ansatz lässt sich die Traversierung vom Besucher aus besser steuern.

Ein Anwendungsbeispiel des Besucher-Musters ist im Klassendiagramm in Abbildung 2.5 gezeigt. Bei der Datenstruktur handelt es sich um einen SQL-Syntax-Baum der einfache *Select*-Statements mit einer *Select*- und *From*-Klausel abbilden kann. Diese sollen nun

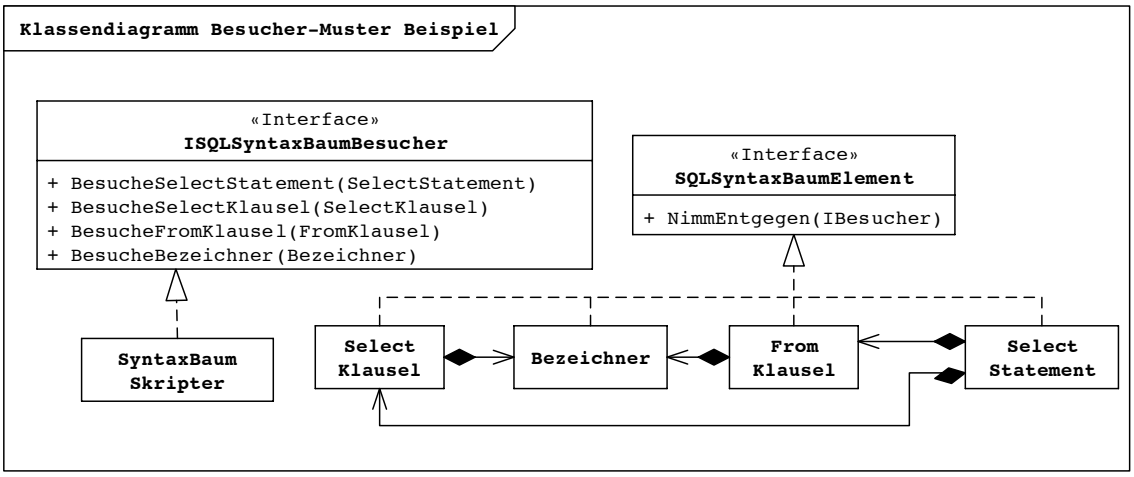

Abbildung 2.5: Klassendiagramm Anwendung des Besucher-Musters

wieder als SQL-Code ausgegeben werden. Dazu wird das Besucher-Muster und der konkrete Besucher `SyntaxBaumSkripter` implementiert. Dieser kann den Baum traversieren und in jeder der Methoden des *ISQLSyntaxBaumBesuchers* die Logik für den jeweiligen Knotentyp implementieren. In diesem Beispiel würde der Besucher ein entsprechendes Token (z.B. *Select* oder den Namen des Bezeichners) ausgeben um ein SQL-Skript zu erzeugen.

Kapitel 3

Typunifikationsalgorithmus

In diesem Kapitel wird die Typunifikation auf einer formalen Ebene beschrieben. Zu Beginn folgt eine allgemeine Beschreibung des Algorithmus und der Typunifikation.

Im Anschluss werden in Abschnitt 3.2 ergänzend zu [Plü07] die Funktionen **smaller**, **greater**, **grArg** und **smArg** definiert und an Beispielen erläutert.

Im darauffolgenden Abschnitt 3.3 werden Informationen zu echten Funktionstypen in Java [Plü15a] zusammengefasst.

In Abschnitt 3.4 wird der Typunifikationsalgorithmus um zusätzliche Inferenzregeln ergänzt.

Abschnitt 3.5 beschreibt, wie die Berechnung des kartesischen Produktes im vierten Schritt des Algorithmus angepasst werden kann, um den Lösungsraum der Unifikation um bisher nicht enthaltene Fälle zu erweitern.

Anpassungen im vierten Schritt führen dazu, dass die Abbruchbedingung der Rekursion umformuliert werden muss. Diese wird in Abschnitt 3.6 behandelt.

Zuletzt folgt eine Zusammenfassung der durchgeführten Erweiterungen am Typunifikationsalgorithmus in Abschnitt 3.7.

3.1 Beschreibung

Die klassische Unifikation (siehe Abschnitt 2.2) versucht Unifikatoren zur finden, die zwei Terme identisch machen. Die Typunifikation versucht Unifikatoren zu finden, die Constraints (Paare) der Form $(\theta \oplus \theta')$ erfüllen. Im Fall der Java-Typunifikation ist der Operator $\oplus \in \{ \lessdot , \lessdot_? , \doteq \}$. Bei der Erfüllung des Constraints ist die Semantik der Operatoren von Bedeutung:

- $(\theta \lessdot \theta')$ bedeutet, um den Constraint zu erfüllen, muss a ein Subtyp von b sein, d.h. $\theta \leq^* \theta'$.
- $(\theta \lessdot_? \theta')$ bedeutet, um den Constraint zu erfüllen, muss $\theta \in \mathbf{smArg}(\theta')$ bzw. $\theta' \in \mathbf{grArg}(\theta)$ sein.
- $(\theta \doteq \theta')$ bedeutet, um den Constraint zu erfüllen, müssen θ und θ' gleich sein. Dies ist der Fall der klassischen Unifikation.

Der Typunifikationsalgorithmus nach [Plü07] findet alle Unifikatoren, die eine Menge von Constraints Eq erfüllen. Er geht dazu in sieben Schritten vor.

1. Im ersten Schritt werden Inferenzregeln solange auf die Menge Eq angewandt, bis diese sich nicht mehr ändert.
2. Im zweiten Schritt wird eine Menge $Eq_1^{'}$ erstellt, in der alle Paare enthalten sind, deren Elemente beide Typvariablen sind.
3. Im dritten Schritt wird eine Menge $Eq_2^{'}$ erstellt, in der alle Paare enthalten sind, deren Elemente nicht beide Typvariablen sind.
4. Im vierten Schritt wird der Lösungsbaum durch Berechnung eines kartesischen Produktes aufgespannt, d.h. hier werden mögliche Lösungswege erzeugt.
5. Im fünften Schritt werden Typvariablen deren Wert gefunden wurde substituiert. Dies ähnelt der subst-Regel im Martelli-Monatanari-Unifikationsalgorithmus.
6. Im sechsten Schritt folgt der rekursive Aufruf für Mengen in den substituiert werden konnte und die Vereinigung aller Lösungen.
7. Der siebte Schritt gibt die Typunifikatoren aus.

3.2 Funktionen auf Typen

Die Funktionen **smaller**, **greater**, **smArg**, **grArg** lassen sich anhand der Finite Closure von (<) (FC(<)) (Definition siehe [Plü07]) berechnen. Ergänzend zu [Plü07] werden diese Funktionen hier definiert und beschrieben.

Definition 3.1 (smaller)**.** *Sei $\leq^*$ die transitive und reflexive Subtyp-Relation auf Extended-Typen wie sie in [Plü07] definiert ist. Dann ist* ***smaller***$(\theta) = \{x \mid x \leq^* \theta\}$.

Die Funktion ***smaller***(θ) *berechnet die Menge aller Typen die in der Subtyp-Relation $\leq^*$ kleiner als θ sind.*

Beispiel 3.2.1. Sei ein Java-Programm gegeben mit (List<T> < Collection<T>), (Integer < Number) und (Number < Object). Dann ist z.B.:

smaller(Object) = {Integer, Number, Object}

smaller(Collection<T>) = {Collection<T>, List<T>}

smaller(Collection<? extends Number>) = {Collection<? extends Number>,

Collection<? extends Integer>, Collection<Number>, Collection<Integer>,

List<? extends Number>, List<? extends Integer>, List<Number>, List<Integer>}

Definition 3.2 (greater)**.** *Sei $\leq^*$ die transitive und reflexive Subtyp-Relation auf Extended-Typen wie sie in [Plü07] definiert ist. Dann ist* ***greater***$(\theta) = \{x \mid \theta \leq^* x\}$.

Die Funktion ***greater***(θ) *berechnet die Menge aller Typen die in der Subtyp-Relation $\leq^*$ größer als θ sind.*

Beispiel 3.2.2. Sei ein Java-Programm gegeben mit (List<T> < Collection<T>), (Integer <

Number) und (Number < Object). Dann ist z.B.:

greater(Integer) = {Integer, Number, Object}

greater(List<T>) = {List<T>, Collection<T>}

greater(List<Number>) = {List<Number>, List<? extends Number>,

List<? super Number>, List<? extends Object>, List<? super Integer>,

Collection<Number>, Collection<? extends Number>, Collection<? super Number>,

Collection<? extends Object>, Collection<? super Integer>}

Definition 3.3 (smArg)**.** *Die Funktion* $\boldsymbol{smArg}(\theta)$ *berechnet die Menge an Typen* θ *welche, im Argument eines umschließenden Typs C dafür sorgen, dass* $C<\theta> \leq^* C<\theta'>$.

$$\boldsymbol{smArg}(\theta) = \begin{cases} \{^{?}\theta' \mid \bar{\theta} \leq^* \theta'\} \cup \{\theta' \mid \bar{\theta} \leq^* \theta'\} & \theta = {}^{?}\bar{\theta} \\ \{_{?}\theta' \mid \theta' \leq^* \bar{\theta}\} \cup \{\theta' \mid \theta' \leq^* \bar{\theta}\} & \theta = {}_{?}\bar{\theta} \\ \{\theta\} & sonst \end{cases}$$

Beispiel 3.2.3. Sei ein Java-Programm gegeben mit (List<T> < Collection<T>), (Integer < Number) und (Number < Object). Dann ist z.B.:

smArg(Integer) = {Integer}

smArg(? extends Number) = {Integer, Number, ? extends Number, ? extends Integer}

smArg(List<? extends Number>) = {List<? extends Number>}

smArg(? extends List<? extends Integer>) = {? extends List<? extends Integer>,

? extends List<Integer>, List<Integer>, List<? extends Integer>}

Definition 3.4 (grArg)**.** *Die Funktion* $\boldsymbol{grArg}(\theta)$ *berechnet die Menge an Typen* θ' *welche, im Argument eines umschließenden Typs C dafür sorgen, dass* $C<\theta> \leq^* C<\theta'>$.

$$\boldsymbol{grArg}(\theta) = \begin{cases} \{^{?}\theta' \mid \theta' \leq^* \bar{\theta}\} & \theta = {}^{?}\bar{\theta} \\ \{_{?}\theta' \mid \bar{\theta} \leq^* \theta'\} & \theta = {}_{?}\bar{\theta} \\ \{\theta\} \cup \{_{?}\theta' \mid \theta \leq^* \theta'\} \cup \{^{?}\theta' \mid \theta' \leq^* \theta\} & sonst \end{cases}$$

Beispiel 3.2.4. Sei ein Java-Programm gegeben mit (List<T> < Collection<T>), (Integer < Number) und (Number < Object). Dann ist z.B.:

grArg(? super Number) = {? super Number, ? super Integer}

grArg(Integer) = {Integer, ? extends Integer, ? super Integer, ? extends Number, ? extends Object}

grArg(? extends List<? extends Integer>) = {? extends List<? extends Integer>,

? extends List<? extends Number>, ? extends List<? extends Object>,

? extends Collection<? extends Integer>, ? extends Collection<? extends Number>,

? extends Collection<? extends Object>}

Korollar 3.2.1. Sei $P_{gr}(\theta, \theta') = \theta \in \mathbf{greater}(\theta')$ ein Prädikat das angibt ob ein Typ θ größer als ein Typ θ' ist. Aus der Transitivität und Reflexivität der Subtyp-Relation folgt, dass auch P_{gr} transitiv und reflexiv ist. Analoges gilt für die Funktionen **smaller**, **smArg** und **grArg**.

Korollar 3.2.2. Es gilt:

$$\theta \in \mathbf{greater}(\theta') \Leftrightarrow \theta' \in \mathbf{smaller}(\theta)$$

$$\theta \in \mathbf{grArg}(\theta') \Leftrightarrow \theta' \in \mathbf{smArg}(\theta)$$

3.3 Funktionstypen

In [Plü15a] werden echte Funktionstypen ($FunN*$-Typen) für Java vorgestellt.

$FunN*$-Typen treten in der FC(<) nur als reflexives Paar auf, d.h. sie sind kein Teil des Vererbungsbaumes, erben nicht, und können nicht vererbt werden. In der Subtyp-Relation unterscheiden sich von Referenztypen, denn für $FunN*$-Typen gilt:

$$FunN*<\theta_0',\theta_1,\ldots,\theta_n> \leq^* FunN*<\theta_0,\theta_1',\ldots,\theta_n'> \text{ gdw } \forall i : \theta_i \leq^* \theta'_i$$

$FunN*$-Typen sind also kontravariant bezüglich ihres Rückgabetyps und kovariant bezüglich ihrer Argumente. Aufgrund der impliziten Varianz macht es keinen Sinn Wildcard-Typen in den Argumenten von $FunN*$-Typen zu verwenden. Wildcard-Typen sind in den

Argumenten daher verboten [Plü15a]. Das Codebeispiel 3.1 zeigt wie sich die Varianz in Java auswirkt.

```
1  Tuple<Number, Number, Number> t1 = new Tuple(1, 2, 3);
2  Tuple<Integer, Object, Number> t2 = new Tuple(1, 2, 3);
3  Tuple<? extends Number, ? super Number, ? super Number> t3 = null;
4  t1 = t2; // Not Possible, will not compile.
5  t3 = t2; // Possible because of explicit wildcard variance.
6
7  Fun2<Number, Number, Number> multiply = (x) -> (y) -> x*y;
8  Fun2<Integer, Object, Number> foo =
9          (x) -> (y) -> x.toString().length() * y;
10 multiply = foo; // Possible because of implicit variance.
```

Codebeispiel 3.1: Varianz von $FunN*$-Typen

3.4 Typinferenzregeln

3.4.1 Wildcardtypen

Die Inferenzregeln des Typunifikationsalgorithmus werden um weitere sieben Regeln aus Abbildung 3.1 ergänzt um Wildcardtypen zu reduzieren.

Die Regeln reduceWcUp, reduceWcLow und reduceWcLeft reduzieren Paare der Form $(\mathrm{x} \lessdot_? \mathrm{y})$ zu Paaren der Form $(\mathrm{x} \doteq \mathrm{y})$, da in diesen drei Fällen die Bedingung des Paares nur durch Gleichheit der beiden Typen erfüllt werden kann.

Die Regeln reduceWcLowRight und reduceWcUpRight dürfen nicht angewandt werden wenn eine Typvariable auf der linken Seite steht. Eine Anwendung würde den Lösungsraum weiter als nötig einschränken und valide Lösungen könnten vom Unifikationsalgorithmus nicht mehr gefunden werden. Dies wird im nachfolgenden Beispiel für die Regel reduceWcLowRight gezeigt.

Beispiel 3.4.1. Sei die $\mathbf{FC}(<) = \{(\text{Integer} < \text{Number})\}$ und sei das Paar (Vector<a> $\lessdot$ Vector<? extends Number>) zu unifizieren. Die Typvariable a kann somit die Typen {? extends Number, ? extends Integer, Number, Integer} annehmen. Sei nun die Regel reduceWcLowRight aus Abbildung 3.1 uneingeschränkt anwendbar.

$$\dfrac{\dfrac{(\text{Vector<a>} \lessdot \text{Vector<? extends Number>})}{(\text{a} \lessdot_? \text{? extends Number})}\ \text{(reduceWcLowRight)}}{(\text{a} \lessdot \text{Number}))}\ \text{(reduce1, siehe [Plü07])}$$

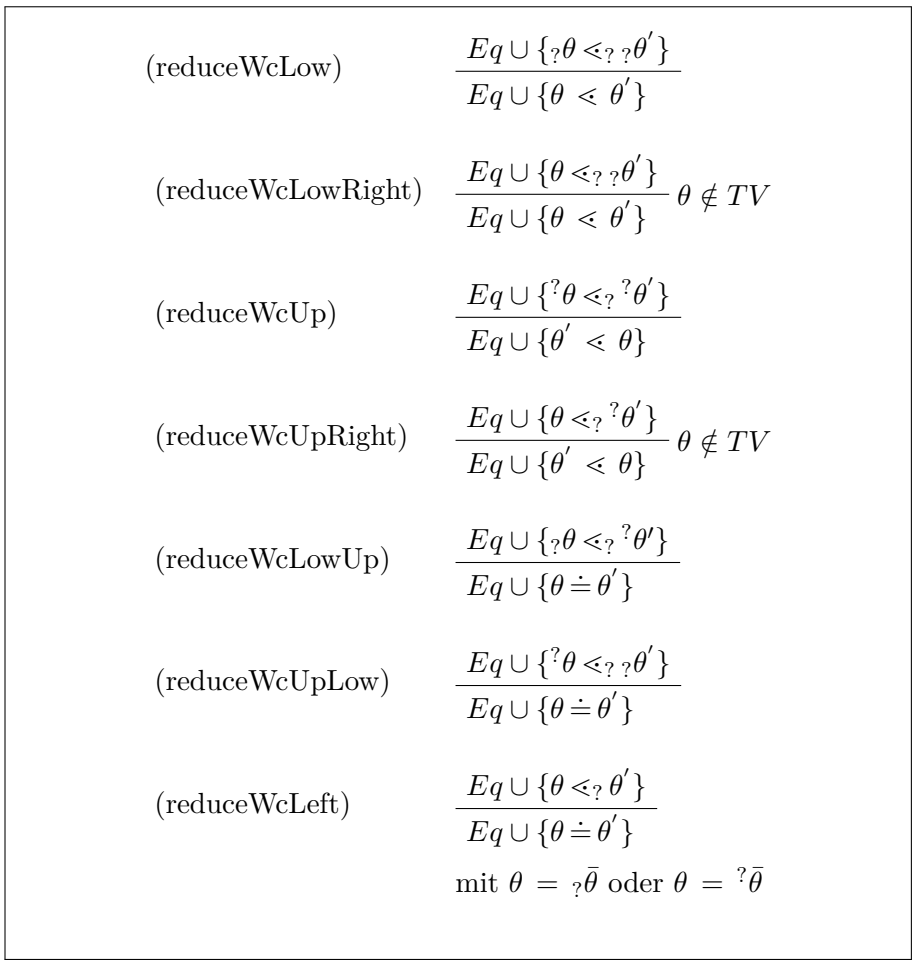

Abbildung 3.1: Typinferenzregeln für Wildcards

Das Paar (a $\lessdot$ Number)) kann nur noch von a $\in$ {Integer, Number} erfüllt werden. Somit wurde der Lösungsraum um die Typen {? extends Integer, ? extends Number} eingeschränkt und das Ergebnis ist nicht mehr vollständig.

3.4.2 Typvariablen

Zusätzlich zu den sieben Wildcard-Inferenzregeln wird der Algorithmus um drei Inferenzregeln für Typvariablen aus Abbildung 3.2 ergänzt.

Paare die durch diese Regeln reduziert werden, wurden bisher durch das kartesische Produkt im vierten Schritt des Algorithmus behandelt. Es handelt sich dabei um die Paare der Form ($\theta \lessdot a$), (${}_?\theta \lessdot_? a$), (${}^?\theta \lessdot_? a$). Die genaue Anpassung des vierten Schrittes wird in Abschnitt 3.5 beschrieben.

$$\text{(reduceTph)} \quad \frac{Eq \cup \{a \lessdot_? \theta'\}}{Eq \cup \{a \doteq \theta'\}}$$

$$\text{(reduceTphExt)} \quad \frac{Eq \cup \{_?\theta \lessdot_? a\}}{Eq \cup \{a \doteq_? b\} \cup \{\theta \lessdot b\}} \text{ (b is fresh)}$$

$$\text{(reduceTphSup)} \quad \frac{Eq \cup \{^?\theta \lessdot_? a\}}{Eq \cup \{a \doteq^? b\} \cup \{b \lessdot \theta\}} \text{ (b is fresh)}$$

Abbildung 3.2: Typinferenzregeln für Typvariablen

$$\text{(reduce}FunN*\text{)} \quad \frac{Eq \cup \{FunN*<\theta,\theta'_1,\ldots,\theta'_N> \lessdot FunN*<\theta',\theta_1,\ldots,\theta_N>\}}{Eq \cup \{\theta \lessdot \theta', \theta_1 \lessdot \theta'_1, \ldots, \theta_N \lessdot \theta'_N\}}$$

$$\text{(greater}FunN*\text{)} \quad \frac{Eq \cup \{FunN*<\theta,\theta'_1,\ldots,\theta'_N> \lessdot a\}}{Eq \cup \{a \doteq FunN*<b',b_1,\ldots,b_N>, \theta \lessdot b', b_i \lessdot \theta'_i\}}$$

where b', b_i are fresh

$$\text{(smaller}FunN*\text{)} \quad \frac{Eq \cup \{a \lessdot FunN*<\theta',\theta_1,\ldots,\theta_N>\}}{Eq \cup \{a \doteq FunN*<b,b'_1,\ldots,b'_N>, b \lessdot \theta', \theta_i \lessdot b'_i\}}$$

where b, b'_i are fresh

Abbildung 3.3: Typinferenzregeln für Funktionstypen
Quelle: [Plü15a]

3.4.3 Funktionstypen

In Abbildung 3.4 sind die Inferenzregeln für Funktionstypen aufgeführt. Die Unterstützung für Funktionstypen wurde im Rahmen dieser Arbeit erstmals implementiert, daher werden die Regeln hier aus Gründen der Vollständigkeit aufgeführt.

3.5 Kartesisches Produkt

In vierten Schritt des Typunifikationsalgorithmus werden mögliche Lösungen durch Berechnung eines kartesischen Produktes gebildet. Haben Paare eine bestimmte Form, kann es vorkommen, dass der Lösungsraum zu weit eingeschränkt wird und bestimmte Lösungen

nicht betrachtet werden.

Dies kann geschehen bei Paaren der Form $(a \lessdot_? {}_?b)$, $(a \lessdot_? {}^?b)$, $({}_?b \lessdot a)$, $({}^?b \lessdot_? a)$ sowie bei Paaren der Form $(\theta_b \lessdot_? a)$, $(\theta_b \lessdot a)$, $(a \lessdot \theta_b)$ wobei θ_b bedeutet dass b in θ vorkommt.

Dies wird im Folgenden anhand von zwei Beispielen verdeutlicht. Anschließend wird eine Anpassung am kartesischen Produkt durchgeführt um die zusätzlichen Fälle miteinzubeziehen.

Beispiel 3.5.1. Das folgende Beispiel beschäftigt sich mit Paaren der Form $(a \lessdot_? {}_?b)$ stellvertretend für alle Paare der Form $(a \lessdot_? {}_?b)$, $(a \lessdot_? {}^?b)$, $({}_?b \lessdot a)$ und $({}^?b \lessdot_? a)$.

Seien folgende Constraints zu unifizieren:

$$Eq = \{(\text{Vector<a>} \lessdot \text{Vector<? super } b\text{>})), (b \doteq \text{Integer})\} \text{ mit}$$
$$\mathbf{FC}(<) = \{(\text{Integer} < \text{Number})\}$$

Durch Anwendung der reduce1-Regel erhält man:

$$Eq = \{(a \lessdot_? \text{? super } b), (b \doteq \text{Integer})\}$$

Dies führt nach [Plü07] zu folgenden kartesischen Produkt:

$$Eq'_{set} = (\bigotimes\{[a \doteq \theta'] \mid \theta' \in \mathbf{smArg}(\text{? super } b))\}) \times \{[b \doteq \text{Integer}]\}$$

Das Problem tritt nun in der Berechnung von **smArg**(? super b) auf. Nach der Definition aus Abschnitt 3.2 enthält **smArg** alle Werte $\{{}^?\theta' | \theta' \leq^* b\}$. Die Typvariable b befindet sich aber nicht selbst in der Subtyprelation, sondern nur ihre konkreten Ausprägungen. Somit lässt sich **smArg** an dieser Stelle nicht vollständig berechnen.

Um mit dem Algorithmus fortfahren zu können, nehmen wir für **smArg**(? super b) = $\{b, \text{? super } b\}$ an, denn dies lässt sich sicher sagen. Es zeigt sich aber, dass durch diese Annahme der Lösungsraum eingeschränkt wurde. Es wird mit dem kartesischen Produkt

fortgefahren:

$$Eq'_{set} = (\{[a \doteq b]\} \times \{[a \doteq ?\text{ super } b]\}) \times \{[b \doteq \text{Integer}]\}$$
$$\{\{(a \doteq b), (b \doteq \text{Integer})\}, \{(a \doteq ?\text{ super } b), (b \doteq \text{Integer}))\}\}$$

Durch Anwendung der Substitution ergeben sich zwei Unifikatoren:

$$\sigma_1 = \{(a \mapsto \text{Integer}), (b \mapsto \text{Integer})\}$$
$$\sigma_2 = \{(a \mapsto ?\text{ super Integer}), (b \mapsto \text{Integer})\}$$

Vergleicht man das Ergebnis der Unifikation mit der Ausgangsmenge Eq unter Berücksichtigung der FC(<), sehen wir, dass folgende Unifikatoren nicht gefunden wurden:

$$\sigma_1 = \{(a \mapsto \text{Number}), (b \mapsto \text{Integer})\}$$
$$\sigma_2 = \{(a \mapsto ?\text{ super Number}), (b \mapsto \text{Integer})\}$$

Beispiel 3.5.2. Das folgende Beispiel beschäftigt sich mit Paaren der Form $(\theta_b \lessdot a)$ stellvertretend für alle Paare der Form $(\theta_b \lessdot_? a)$, $(\theta_b \lessdot a)$, $(a \lessdot \theta_b)$ wobei θ_b bedeutet dass b in θ vorkommt.

Seien folgendene Constraints zu unifizieren:

$$Eq = \{(\text{X<}b\text{>} \lessdot a), (b \doteq ?\text{ extends Number})\} \text{ mit}$$
$$\mathbf{FC}(<) = \{(\text{Integer} < \text{Number})\}$$

Dies führt nach [Plü07] zu folgenden kartesischen Produkt:

$$Eq'_{set} = (\bigotimes\{[a \doteq \theta'] \mid \theta' \in \mathbf{greater}(\text{X<}b\text{>})\}) \times \{[b \doteq ?\text{ extends Number}]\}$$

Wie schon im vorherigen Beispiel tritt nun das Problem auf, dass sich die die Funktion **greater**(X<b>) nicht vollständig berechnen lässt, da sie vom konkreten Wert der Variablen b abhängt (vergleiche dazu die Definition von **greater** in Abschnitt 3.2 und die Definition der Subtyp-Relation in [Plü15a]). Wie bereits im vorherigen Beispiel, führt dies dazu, dass

Lösungen nicht beachtet werden.

3.5.1 Anpassungen

Die Beispiele zeigen, dass unter Umständen Lösungen nicht gefunden werden, wenn eine der Funktionen **greater**, **smaller**, **grArg** oder **smArg** auf eine Typvariable b (oder einen Typ θ_b) angewendet werden soll. Dieses Problem lässt sich lösen, indem eine neue Typvariable b' eingeführt wird, welche die alte Typvariable b ersetzt und gleichzeitig in einem zusätzlichen Constraint (d.h. Paar) von b gebunden wird. Dadurch lässt sich die Berechnung der Funktion auf einen späteren Zeitpunkt verschieben, zu dem der Wert von b bekannt ist.

$$
\begin{aligned}
&Eq'_{set}\\
&= \{Eq'_1\} \times \Big(\bigotimes_{(a \lessdot \theta') \in Eq'_2} \{([(a \doteq \mathrm{D}\langle b_1,\ldots,b_m\rangle), (b_1 \lessdot_? \theta_1), \ldots, (b_m \lessdot_? \theta_m)] \cup \sigma) \mid \\
&\qquad\qquad (\bar{\theta} \leq^* \mathrm{C}\langle\theta_1,\ldots,\theta_n\rangle) \in \mathbf{FC}(<) \\
&\qquad\qquad \bar{\theta}' \in \{\mathrm{C}\langle\theta'_1,\ldots,\theta'_n\rangle \\
&\qquad\qquad\qquad \mid \theta'_i \in \mathbf{grArg}(\theta_i), 1 \leq i \leq n\} \\
&\qquad\qquad \sigma \in Unify(\bar{\theta}', \theta') \\
&\qquad\qquad \mathrm{D}\langle\theta_1,\ldots,\theta_m\rangle \in \mathbf{smaller}(\sigma(\bar{\theta})), \\
&\qquad\qquad b_i \text{ are fresh } \}) \\
&\quad \times \Big(\bigotimes_{(a \lessdot_? {}_?\theta') \in Eq'_2} \{[(a \lessdot \theta')], [(a \doteq_? b), (b \lessdot \theta')] \mid b \text{ is fresh}\}\Big) \\
&\quad \times \Big(\bigotimes_{(a \lessdot_? {}^?\theta') \in Eq'_2} \{[(\theta' \lessdot a)], [(a \doteq {}^? b), (\theta' \lessdot b)] \mid b \text{ is fresh}\}\Big) \\
&\quad \times \Big(\bigotimes_{(\theta \lessdot a) \in Eq'_2} \{[(a \doteq \mathrm{C}\langle b_1,\ldots,b_m\rangle), (\theta_1 \lessdot_? b_1), \ldots, (\theta_m \lessdot_? b_m)] \mid \\
&\qquad\qquad\qquad \mathrm{C}\langle\theta_1,\ldots,\theta_m\rangle \in \mathbf{greater}(\theta), \\
&\qquad\qquad\qquad b_i \text{ are fresh}\}\Big) \\
&\quad \times \Big(\bigotimes_{(\theta \lessdot_? a) \in Eq'_2} \{[(a \doteq \theta)], [(a \doteq_? b), (\theta \lessdot b)], [(a \doteq_? b), (b \lessdot \theta)] \mid b \text{ is fresh}\}\Big) \\
&\quad \times \{[a \doteq \theta \mid (a \doteq \theta) \in Eq'_2]\}
\end{aligned}
$$

Es fällt auf, dass im Vergleich zu Schritt vier aus [Plü07] die Fälle $(\theta \lessdot a)$, $({}_?\theta \lessdot_? a)$ und $({}^?\theta \lessdot_? a)$ nicht mehr im kartesischen Produkt behandelt werden. Diese Fälle konnten durch die Einführung neuer Typvariablen vereinfacht und in die Inferenzregeln aus Abbildung

3.2 ausgelagert werden. Des Weiteren hat sich der Fall $(a \lessdot_? {}_? \theta')$ durch die Einführung der neuen Typvariablen wesentlich vereinfacht.

3.6 Rekursionsfall

In Abschnitt 3.5 wurde eine Anpassung und Vereinfachung des kartesischen Produktes beschrieben. Dazu werden im kartesischen Produkt Paare der Form $(\theta < \theta')$ erzeugt. Dadurch können allerdings Mengen Eq' auftreten, auf welche die subst-Regel nicht angewandt werden kann, die aber dennoch durch einen rekursiven Aufruf unifizierbar sind. Daher muss die Abbruchbedingung der Rekursion aus Schritt 6a abgeändert werden, um in diesen Fällen einen rekursiven Aufruf zuzulassen:

6. (a) Foreach $Eq' \in Eq'_{set}$ where $Eq' \neq Eq$ start again with the first step.

Das bedeutet das ein rekursiver Aufruf erfolgt, solange Änderungen an der Eingabemenge Eq durchgeführt werden konnten.

3.7 Zusammenfassung der Anpassungen

In diesem Kapitel wurde der Typunifikationsalgorithmus auf einer konzeptionellen Ebene behandelt. Dabei wurden Anpassungen durchgeführt, die den Algorithmus um bisher fehlende Fälle ergänzt haben. Hier werden die Änderungen gegenüber dem Algorithmus aus [Plü15a] noch einmal zusammengefasst.

Schritt 1 In Abschnitt 4.5 werden Regeln für Wildcards (Abbildung 3.1), Regeln für Typvariablen (Abbildung 3.2) und die in [Plü15a] eingeführten Regeln für Funktionstypen (Abbildung 3.3) ergänzt.

Schritt 4 In Abschnitt 3.5 wird das kartesische Produkt angepasst um einzelne, bisher nicht enthaltene Fälle berechnen zu können. Dafür werden neue Typvariablen eingeführt, was zur Folge hat, dass sich Teile des kartesischen Produktes wesentlich vereinfachen und in Regeln auslagern lassen.

Schritt 6 Durch die Anpassung des kartesischen Produktes, muss die Abbruchbedingung der Rekursion geändert werden. Dies wird in Abschnitt 3.6 beschrieben.

Kapitel 4

Implementierung

Dieses Kapitel beschreibt die konkrete Implementierung des Typunifikationsalgorithmus. Dafür werden in Abschnitt 4.1 zunächst einige allgemeine Aspekte erläutert.

Im Anschluss wird in Abschnitt 4.2 darauf eingegangen, wie Java-Typen im Algorithmus abgebildet werden.

Danach wird in Abschnitt 4.3 die Implementierung der FC(<) beschrieben und deren Methoden spezifiziert.

In Abschnitt 4.4 wird auf die Implementierung eines Standard-Unifikationsalgorithmus eingegangen, der an vielen Stellen als Subroutine Verwendung findet.

Die Implementierung der Typinferenzregeln wird in Abschnitt 4.5 erläutert.

Anschließend wird in Abschnitt 4.6 die Implementierung des Hauptteils des Typunifikationsalgorithmus beschrieben, wobei auch gezeigt wird, wie die zuvor beschriebenen Teile zusammenwirken.

Zuletzt wird noch auf die Verwendung von Unit-Tests zur Sicherung der Korrektheit und auf die Parallelisierung der Implementierung eingegangen.

4.1 Grundlagen

4.1.1 Unveränderlichkeit

Unveränderliche Datenstrukturen werden in der funktionalen Programmierung genutzt. Operationen auf diesen Datenstrukturen ändern nicht die Struktur selbst, sondern erzeugen eine, bis auf die erwirkte Änderung, identische Kopie. Unveränderliche Datenstrukturen haben also nur einen einzigen Zustand.

Um Klassen in Java als unveränderlich zu implementieren müssen vier Kriterien erfüllt sein [Ora16].

1. Es darf keine `Setter`-Methoden geben (oder `Setter`-Methoden erzeugen ein neues, geändertes Objekt).
2. Alle Felder müssen `final` und `private` sein, damit sie nicht von erbenden Klassen manipuliert werden können.
3. Unterklassen dürfen nicht in der Lage sein Methoden zu überschreiben. Das kann erreicht werden indem die Klasse als `final` deklariert wird.
4. Enthält die Klasse veränderliche Objekte, dürfen diese nicht verändert werden und keine Referenz darf von außen auf das Objekt gehalten werden können.

4.1.2 UML-Stereotypen

Zur Dokumentation der Implementierung des Typunifikationsalgorithmus werden unter anderem Klassen- und Sequenzdiagramme der UML genutzt. Dabei kommen UML-Stereotypen zum Einsatz. Diese werden im Folgenden definiert. Für gerichtete Assoziationen werden dabei die Begriffe Quelle und Ziel genutzt, wobei gilt: *Quelle* $\rightarrow$ *Ziel*

Definition 4.1 («`immutable`»-Stereotyp)**.** *Stereotyp für Klassen. Die Klasse erfüllt die Kriterien für Unveränderlichkeit aus Abschnitt 4.1.1.*

Definition 4.2 («`use`»-Stereotyp)**.** *Stereotyp für Assoziationen. Die Quelle benutzt Methoden des Ziels oder nutzt das Ziel als Datenstruktur.*

Definition 4.3 (`«creates»`-Stereotyp)**.** *Stereotyp für Assoziationen. Die Quelle erstellt Instanzen des Ziels.*

Definition 4.4 (`«visits»`-Stereotyp)**.** *Stereotyp für Assoziationen. Quelle und Ziel formen ein Visitor-Pattern (vgl. Abschnitt 2.4.2). Die Quelle ist der Visitor und das Ziel der Acceptor.*

Definition 4.5 (`«strategy»`-Stereotyp)**.** *Stereotyp für Assoziationen. Quelle und Ziel formen ein Strategy-Pattern (vgl. Abschnitt 2.4.1). Das Ziel ist immer ein Interface. Das Ziel enthält die Strategie und parametrisiert damit die Quelle.*

Definition 4.6 (`«applicable»`-Stereotyp)**.** *Stereotyp für Assoziationen. Die Quelle ist eine Funktion, die auf das Ziel anwendbar ist.*

Definition 4.7 (`«get»`-Stereotyp)**.** *Stereotyp für Instanzvariablen. Es existiert eine Getter-Methode um den Wert des Feldes abzufragen.*

Definition 4.8 (`«set»`-Stereotyp)**.** *Stereotyp für Instanzvariablen. Es existiert eine Setter-Methode um den Wert des Feldes zu setzen.*

Mehrfache Stereotypen werden mit «s_1, s_2,..., s_n»notiert.

Zusätzlich zu den hier definierten Stereotypen werden die Standard-Stereotypen `«constructor»`, `«interface»` und `«enum»` verwendet.

4.1.3 Optional

Die Klasse `Optional` wurde mit der Java-Stream Bibliothek eingeführt. `Optional` entspricht dem Maybe-Datentyp wie er in der funktionalen Programmierung genutzt wird.

Bei `Optional` handelt es sich um einen generischen Typ. Eine Instanz ist entweder `Optional.empty` oder enthält ein Objekt des entsprechenden Typs. Codebeispiel 4.1 zeigt ein einfaches Beispiel der Benutzung in Java.

Für die Implementierung des Typunifikationsalgorithmus werden `Optionals` als Rückgabewerte genutzt. Eine Methode die ein `Optional` zurückgibt, kann durch Rückgabe von `Optional.emtpy` mitteilen, dass sie für die Eingabe nicht anwendbar oder nicht definiert ($\perp$) ist.

```java
// Pack an Integer into an Optional
Optional<Integer> opt = Optional.of(1);

// Check if the optional contains a value
if(opt.isPresent()) {
  Integer value = opt.get(); // Unpack the value
  opt = Optional.empty();    // Assign an empty optional
}
```

Codebeispiel 4.1: Der Typ Optional in Java

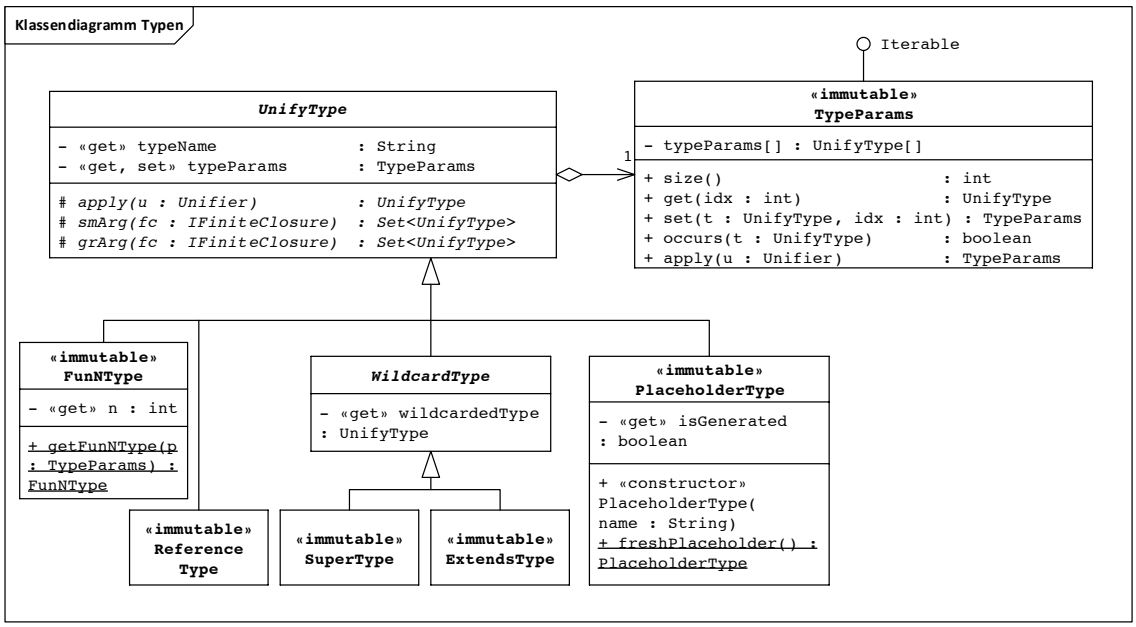

Abbildung 4.1: Klassendiagramm Typen

`Optionals` dienen zur Vermeidung von `null`-Werten. Durch Rückgabe von `Optionals` wird dem Aufrufenden einer Methode, klarer als durch Rückgabe von `null`, deutlich gemacht, dass diese Methode für bestimmte Eingaben undefiniert ist. `NullPointerExceptions` werden vermieden.

4.2 Typen

Die Typen des Java-Typsystems werden über die Klassenstruktur aus Abbildung 4.1 modelliert. Alle gezeigten, nicht abstrakten Klassen sind unveränderlich, d.h. sie haben nur einen einzigen möglichen Zustand. Verändernde Methoden erzeugen ein neues Objekt anstatt den Zustand des Alten zu verändern.

Basistyp Alle Typen erben von der gemeinsamen und abstrakten Basisklasse `UnifyType`. Diese bietet Methoden als Schnittstellen für die FC (mehr in Abschnitt 4.3) und Unifikatoren (Abschnitt 4.4). Alle Typen besitzen einen Namen (z.B. Integer oder ? extends List<T>) und Typparameter die über die Klasse `TypeParams` abbgebildet werden.

Wildcardtypen Die abstrakte Klasse `WildcardType` bildet Java-Wildcards ab und hat die konkreten Unterklassen `ExtendsType` und `SuperType`.

Typvariablen `PlaceholderTypes` sind ungebundene Typvariablen. Um gebundene Typvariablen abzubilden muss die Klassenstruktur um einen entsprechenden Typ ergänzt werden. `PlaceholderTypes` können über einen Konstruktor oder über eine statische Fabrikmethode instanziiert werden. Die Fabrik-Methode erzeugt neue Typvariablen mit einen zufallsgenerierten Namen, welche zum Zeitpunkt des Aufrufs garantiert noch nicht im Programm vorkommen.

Funktionstypen Die Klasse `FunNType` repräsentiert echte Java-Funktionstypen wie sie in [Plü15a] vorgestellt werden. Sie müssen über die Fabrikmethode `getFunNType` instanziiert werden. Die Methode stellt sicher, dass die Typparameter valide sind, d.h. keine unerlaubten [Plü15a] Wildcard-Typen enthalten.

Typparameter Alle Typen haben ein Feld welches die Typparameter (ein `TypeParams`-Objekt) enthält. Ein `TypeParams`-Objekt kann ein Occurs-Check ausführen. Außerdem kann ein Unifikator angewandt werden. Die Implementierung von Unifikatoren wird in Abschnitt 4.4 beschrieben. Bei `PlaceholderTypes` sind `TypeParams` stets leer. Bei `WildcardTypes` werden die Parameter des umschlossenen Typs als Typparameter verwendet.

4.3 Finite Closure

Die FC(<) ist in [Plü07] definiert als der reflexive und transitive Abschluss einer bestimmten Untermenge der Subtyp-Relation. Diese Untermenge enthält dabei alle Paare die durch

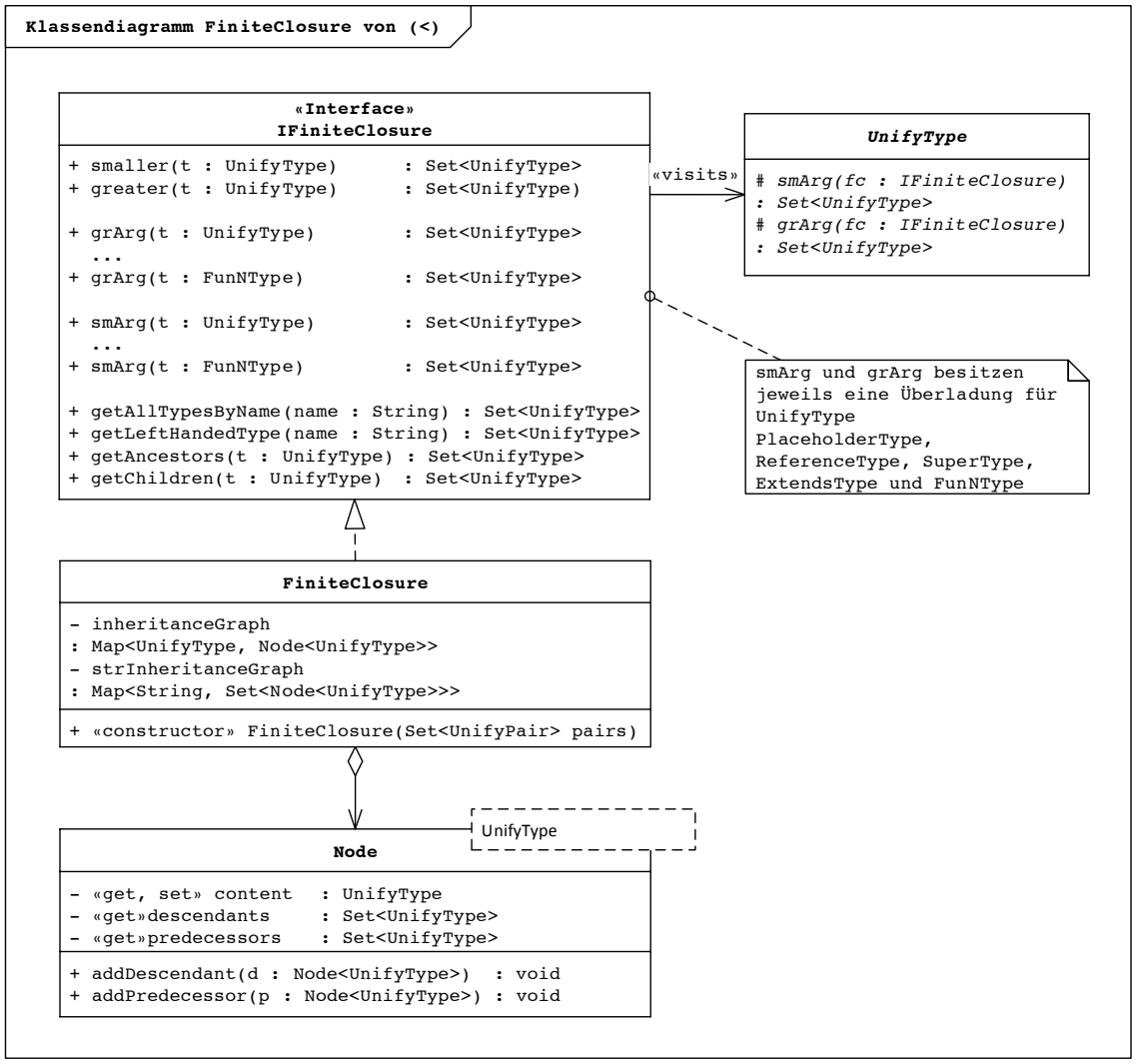

Abbildung 4.2: Klassendiagramm FiniteClosure

Klassendeklarationen entstehen. Aus der FC(<) lässt sich die Subtyp-Relation bestimmen und somit auch die Funktionen **smaller**, **greater**, **grArg** und **smArg**.

4.3.1 IFiniteClosure-Interface

Abbildung 4.2 zeigt die Implementierung der FC(<). Das Interface `IFiniteClosure` enthält die Operationen die eine Implementierung der FC(<) unterstützen muss. Die Verwendung eines Interfaces ermöglicht es, die konkrete Implementierung der FC(<) bei Bedarf auszutauschen. Dadurch ist es möglich, eine effizientere Implementierung (z.B. im Bezug auf die Laufzeit oder den Speicherplatzbedarf) ohne Anpassungen am Unifikationsalgorithmus zu verwenden. Durch alternative Implementierungen kann außerdem das Verhalten des

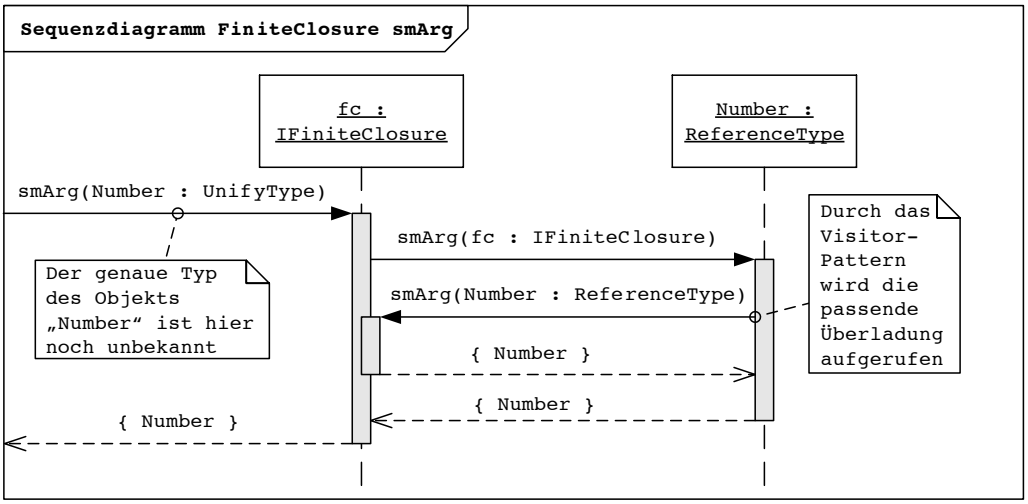

Abbildung 4.3: Sequenzdiagramm Aufruf von smArg

Unifikationsalgorithmus geändert werden. Es ließe sich z.B. eine FC(<)-Implementierung verwenden, die nicht die vollständige Subtyp-Relation berechnet, sondern nur eine Untermenge. Durch diese Einschränkung des Lösungsraumes ließe sich die Laufzeit der Unifikation (auf Kosten der Vollständigkeit) verbessern.

Das `IFiniteClosure`-Interface enthält die Methoden `smaller`, `greater`, `grArg` und `smArg`. Diese erfüllen die Spezifikation der korrespondierenden Funktionen aus Abschnitt 3.2.

Da in der Definition der Funktionen **smArg** und **grArg** eine Fallunterscheidung nach Art des Typs durchgeführt wird, bietet es sich an, diese entsprechenden Methoden durch Visitor-Pattern (vgl. Abschnitt 2.4.2) zu realisieren. Sequenzdiagramm 4.3 zeigt den Ablauf eines Aufrufs von `smArg`. Durch Verwendung des Visitor-Patterns wird stets die speziellste Überladung in der FC(<) aufgerufen, obwohl der speziellste Typ des Arguments beim Aufruf unbekannt ist. Eine Fallunterscheidung des Typs mit dem `instanceof`-Operator ist nicht notwendig.

Im Folgenden werden die weiteren Methoden des Interfaces spezifiziert.

Definition 4.9 (IFiniteClosure.getAllTypesByName)**.** *$getAllTypesByName(C)$ berechnet die Menge aller Typen die in der FC(<) vorkommen und den Namen C besitzen. Die*

Parametrisierung mit Typparametern spielt dabei keine Rolle.

$$getAllTypesByName(C) = \{C<...> \mid (C<...> < x) \in \boldsymbol{FC}(<) \vee (x < C<...>) \in \boldsymbol{FC}(<)\}$$

Definition 4.10 (IFiniteClosure.getLeftHandedType)**.** *$getLeftHandedType(C)$ durchsucht die FC(<) nach dem Typ, der den Namen C hat und im einem Paar $(C<...> < x)$ der FC(<) linksseitig steht. Das gesuchte Paar entsteht durch die Deklaration der Klasse mit dem Namen C. Da Klassen nur einmal deklariert werden können, ist der Typ $C<...>$ eindeutig bestimmbar.*

$$Sei\ T = \{C<...> \mid (C<...> < x) \in \boldsymbol{FC}(<)\}$$

$$getLeftHandedType(C) = \begin{cases} C<...> & T = \{C<...>\} \\ \bot & sonst \end{cases}$$

Definition 4.11 (IFiniteClosure.getAncestors)**.** *$getAncestors(\theta)$ bestimmt alle Typen die in der $\boldsymbol{FC}(<)$ (nicht der Subtyp-Relation) größer oder gleich θ sind.*

$$getAncestors(\theta) = \{\theta' \mid (\theta < \theta') \in \boldsymbol{FC}(<)\}$$

Definition 4.12 (IFiniteClosure.getChildren)**.** *$getChildren(\theta')$ bestimmt alle Typen die in der FC(<) (nicht der Subtyp-Relation) kleiner oder gleich θ' sind.*

$$getChildren(\theta') = \{\theta \mid (\theta < \theta') \in \boldsymbol{FC}(<)\}$$

4.3.2 Implementierung der Finite Closure

Die Implementierung des Interfaces ist im Klassendiagramm aus Abbildung 4.2 durch die Klasse `FiniteClosure` gegeben. Die `FiniteClosure` basiert im Wesentlichen auf dem transitiven Abschluss des Vererbungsbaumes der durch `Node`-Objekte abgebildet wird.

Um ein Objekt von `FiniteClosure` zu erzeugen, muss im Konstruktor eine Menge von Paaren übergeben werden. Diese Paare bilden den Vererbungsbaum des Java-Programmes ab. Abbildung 4.4 zeigt einen beispielhaften Satz an solchen Paaren. Im Konstruktor

(Integer < Number)

(Number < Object)

(List<T> < Collection<T>)

(Collection<T> < Object)

(MyList < List<Integer>)

(List<Integer> < Object)

Abbildung 4.4: Paare der FC(<)

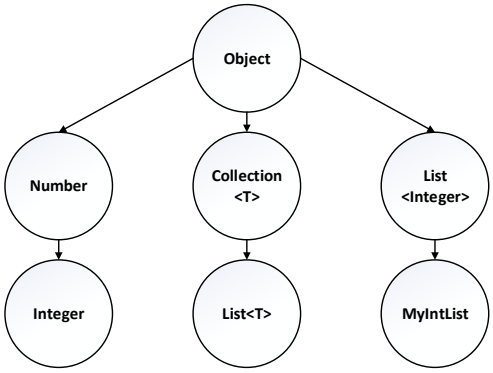

Abbildung 4.5: Vererbungsbaum

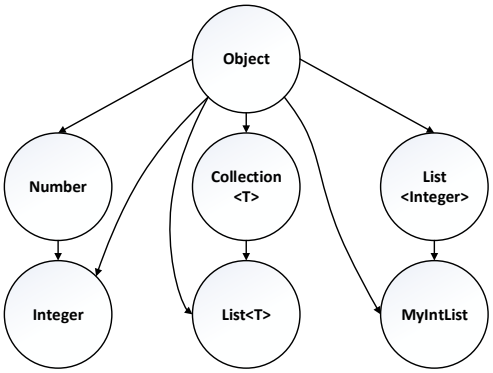

Abbildung 4.6: Transitiver Abschluss

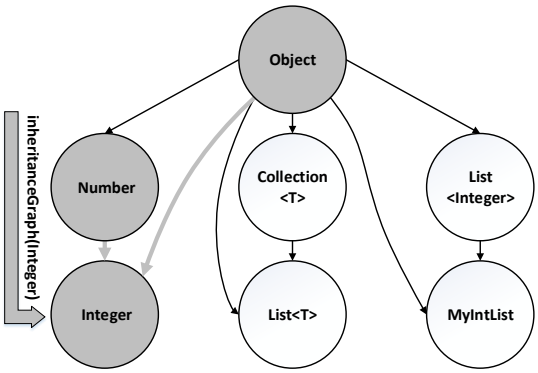

Abbildung 4.7: Aufruf getAncestors(Integer)

wird aus den Paaren der Vererbungsbaum aus `Nodes` erzeugt (Abbildung 4.5) und der transitive Abschluss gebildet (Abbildung 4.6), indem von jeder `Node` eine direkte Kante zu allen indirekten Vorgängern und Nachfolgern eingefügt wird. Der erzeugte Graph ist ein gerichteter Graph, trotzdem sind Kanten in beiden Richtungen navigierbar.

Der Zugriff auf den Graph erfolgt über zwei `Maps`. Die `Map inheritanceGraph` bildet jeden Typen auf die `Node` ab, in welcher der Typ abgelegt ist. Diese `Map` ermöglicht es Anfragen wie z.B. `getAncestors(UnifyType t)` in $O(1)$[1] zu beantworten. Dazu wird in der `Map` die `Node` gesucht, die den Typ `t` enthält. Durch die Vorberechnung des transitiven Abschlusses, stehen in dieser `Node` bereits alle gesuchten Vorgänger. Abbildung 4.7 veranschaulicht diese Vorgehen.

Die zweite `Map strInheritanceGraph` bildet den Namen eines Typs auf alle Typen aus der FC(<) ab, die diesen Namen haben. Dies ist nötig, da die FC(<) mehrere Typen gleichen Namens beinhalten kann, z.B. (MyList < **List**<Integer>) und (**List**<T> <

[1] Vorraussetzung ist, dass die `Map` Anfragen in $O(1)$ beantworten kann. Dies ist z.B. durch Verwendung einer `HashMap` möglich.

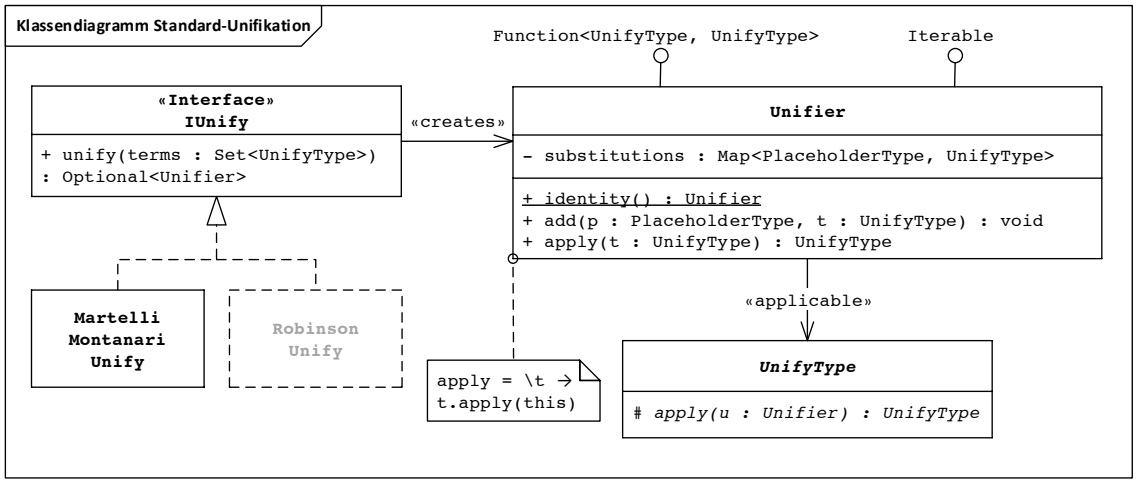

Abbildung 4.8: Klassendiagramm Standard-Unifikation

Collection<T>). Durch diese `Map` können Anfragen wie z.B. `getAllTypesByName` ebenfalls in $O(1)$ beantwortet werden.

4.4 Standard-Unifikation

Als Standard-Unifikation wird im folgenden die Unifikation bezeichnet, wie sie in Abschnitt 2.2 beschrieben wird. Als Erweiterung dazu wird in [Plü07] die Typunifikation beschreiben.

Für die Typunifikation wird ein Standard-Unifikationsalgorithmus als Subroutine benötigt. Die Standard-Unifikation wird benötigt zur Berechnung der Subtyp-Relation (für **smaller** und **greater**), in adapt und subst-Regeln sowie im vierten Schritt zur Bestimmung der Mengen bei der Berechnung des kartesischen Produktes.

Die Standard-Unifikation entspricht dem Speziallfall der Typunifikation in dem alle Typvariablen ungebunden sind und alle Paare der Eingabemenge die Form $(x \doteq y)$ haben. Die separate Implementierung eines Standard-Unifikationsalgorithmus bietet Vorteile gegenüber der Verwendung des Typunifikationsalgorithmus für Standard-Unifikationsprobleme. Diese Vorteile sind die Austauschbarkeit der Implementierung sowie einfacheres Debuggen der Typunifikation.

In Abbildung 4.8 ist ein Klassendiagramm zur Implementierung der Standard-Unifikation. Zur leichteren Austauschbarkeit wird die Implementierung vom Interface `IUnify` gekapselt.

Definition 4.13 (IUnify.unify)**.** *Die Methode* $unify(\{\theta_1, \ldots, \theta_n\})$ *berechnet den MGU* σ *einer Menge an Typtermen, sodass* $\sigma(\theta_1) = \sigma(\theta_2) = \ldots = \sigma(\theta_n)$*. Ist eine Menge* t *nicht unifizierbar ist gilt* $unify(t) = \perp$*.*

Unifikatoren werden von der Klasse `Unifier` abgebildet. `Unifier` sind eine Funktion $UnifyType \rightarrow UnifyType$.

Durch die Factory-Methode `identity` wird ein `Unifier` $\lambda x \rightarrow x$ erstellt.

Substitutionen $\sigma = (a \rightarrow t)$ können über die Methode `add` hinzugefügt werden. Die Substitutionen werden intern über die `Map substitutions` verwaltet.

Die Methode `apply` wendet den `Unifier` auf einen Typterm an. Die eigentliche Logik der Anwendung ist dabei in die abstrakte Methode `apply` Klasse `UnifyType` ausgelagert, da die Anwendung für unterschiedliche Arten von Typen unterschiedlich implementiert werden muss, z.B. muss bei Wildcard-Typen der Unifikator auf den inneren Typ angewandt werden.

Mit `MartelliMontanariUnify` existiert eine Implementierung des `IUnify`-Interfaces. Der Martelli-Montanari-Algorithmus ist in Abschnitt 2.2.1 beschrieben und führt die Unifikation in linearer Laufzeit durch. Weitere Implementierung, wie eine Implementierung der Robinson-Unifikation sind möglich. Die Robinson-Unifikation ist besonders für kleine Eingabegrößen effizient [HV09].

4.5 Inferenzregeln

Für die Typunifikation müssen Inferenzregeln wiederholt auf Elemente der Eingabemenge angewandt werden. Das `IRuleSet`-Interface aus Abbildung 4.9 stellt diese Regeln bereit. Bei diesem Interface handelt es sich um ein `Strategie`-Pattern, welches die Logik der Inferenzregeln für die Typunifikation enthält.

Mit Ausnahme der `subst`-Regel wird jeder Regel ein `UnifyPair` als Argument übergeben, z.B. $(x \lessdot Integer)$. Ist die Regel nicht anwendbar, weil das Paar z.B. den falschen Operator hat, so ist der Rückgabewert `Optional.empty`. Kann die Regel angewandt werden, so wird ein `Optional` mit einem Ergebniswert zurückgegeben. Dieser Wert ist entweder ein

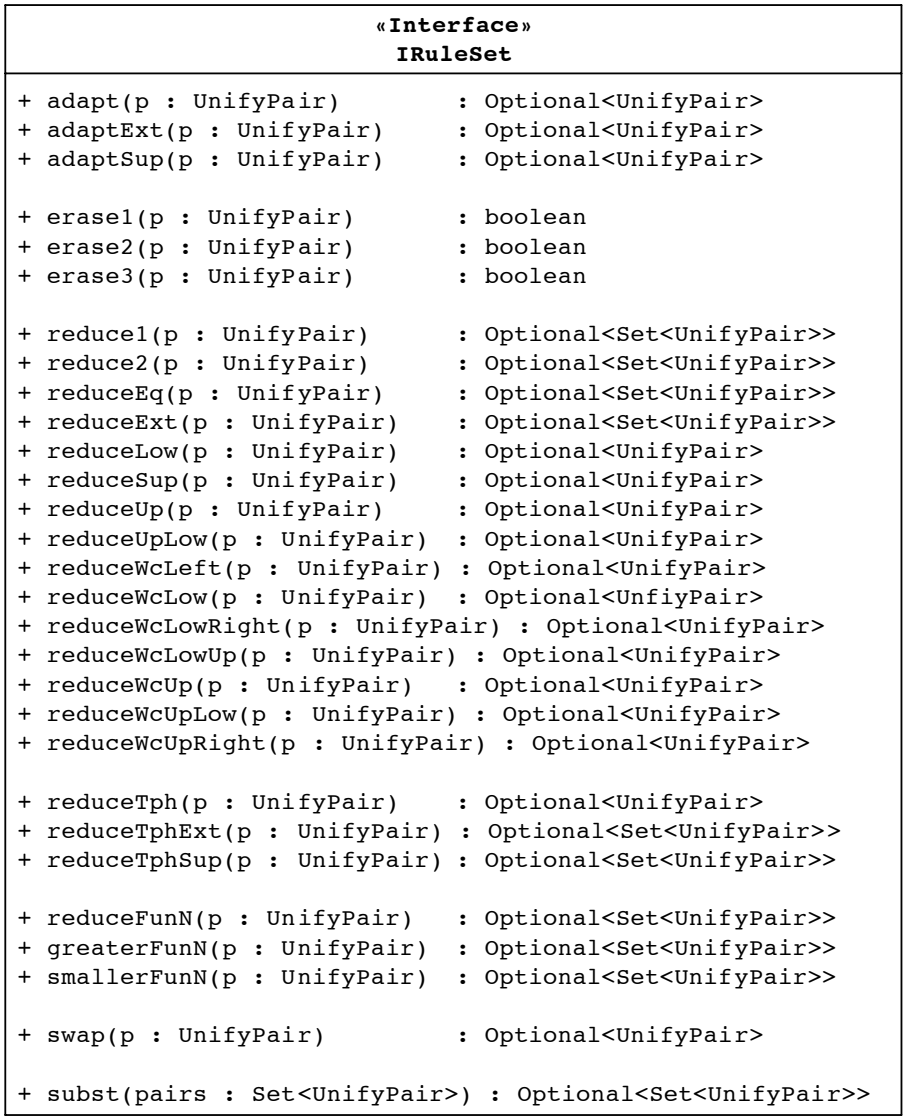

Abbildung 4.9: Klassendiagramm IRuleSet

einzelnes Paar, welches durch die Regel modifiziert wurde oder eine Menge an Paaren, wie z.B. bei der `reduce1`-Regel.

Das Ergebnis einer `erase`-Regel ist entweder `true`, falls das Paar gelöscht werden kann oder `false`, falls das Paar nicht gelöscht werden darf.

Die `subst`-Regel führt die Substitution in einer Menge `pairs` durch. Wurde mindestens ein Typ substituiert wird die entstandene Menge zurückgegeben, ansonsten ist das Ergebnis `Optional.empty`.

Neben den Regeln aus [Plü07] enthält das `IRuleSet` zusätzliche `reduce`-Regeln für Wildcards, Typvariablen und die Regeln für die Behandlung von echten Funktionstypen. Die

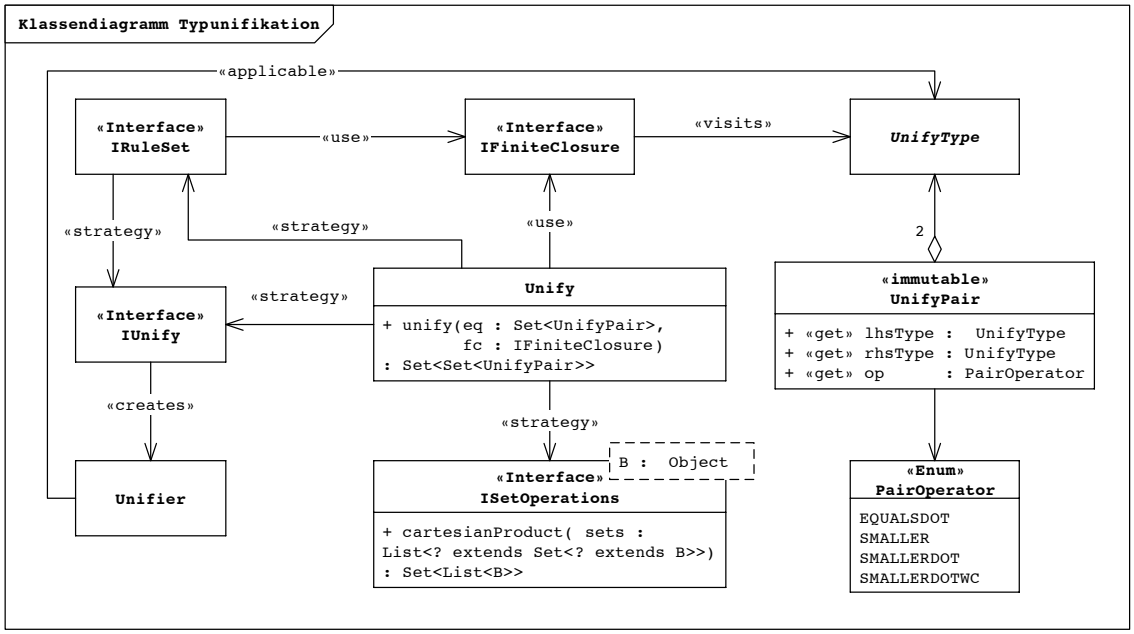

Abbildung 4.10: Klassendiagramm Typunifikation

zusätzlichen Regeln werden in Abschnitt 3.4 betrachtet.

4.6 Typunifikation

Das Klassendiagramm in Abbildung 4.10 zeigt die wichtigsten Klassen und Zusammenhänge der Typunifikation. Die Klasse `TypeUnify` bildet den Kern der gesamten Implementierung. Die Methode `unify` bestimmt alle allgemeinen Typunifikatoren der Menge `eq`. Diese Methode ist der eigentliche Typunifikationsalgorithmus.

Die Menge `eq` besteht aus `UnifyPairs`, welche die zu unifizierenden Gleichungen abbilden. Jedes `UnifyPair` besteht aus einem linken und einem rechten Typen, und einem `PairOperator`, der die Beziehung zwischen diesen Typen angibt.

Die Klasse `unify` wird mit drei verschiedenen Strategien parametrisiert. `IRuleSet`, beschrieben in Abschnitt 4.5, dient zur Anwendung der Inferenzregeln im ersten Schritt des Algorithmus. `IUnify`, beschrieben in Abschnitt 4.4, ist ein Standard-Unifikationsalgorithmus. Die Strategie `ISetOperations` bietet eine generische Methode zur Berechnung des kartesischen Produktes und findet unter anderem im vierten Schritt der Typunifikation Anwendung.

4.6.1 Beispiel

Für das folgende Beispiel sei die $\mathbf{FC}(<) = \{(\text{Int} < \text{Num})\}$.

Das Sequenzdiagramm in Abbildung 4.11 zeigt den Algorithmus bei der Unifikation der folgenden Menge:

$$Eq = \{(\text{L<}a\text{>} \lessdot \text{L<? super Int>}), (\text{Int} \lessdot \text{Num}), (a \lessdot_? b)\} \quad a, b \in TV$$

Das Beispiel ist idealisiert, es entspricht nicht exakten Ablauf, sondern wurde vereinfacht und dient dazu die grundlegende Vorgehensweise der Implementierung zu demonstrieren.

Schritt 1: Anwendung der Inferenzregeln Zunächst werden die Inferenzregeln auf die Eingabemenge angewandt. Im Sequenzdiagramm sind nur die erfolgreichen Regelanwendungen zu sehen. Andere, nicht anwendbare Regeln, können ebenfalls aufgerufen werden, geben aber ein `Optional.empty` zurück.

Durch Anwendung der `reduce1` Regel wird das Paar (L<a> $\lessdot$ L<? super Int>) zu ($a \lessdot_?$? super Int) reduziert. Die Anwendung der `erase1`-Regel auf das Paar (Int $\lessdot$ Num) gibt `true` zurück, da `Int` von `Num` erbt. Damit kann das Paar gelöscht werden. Auf das Paar ($a \lessdot_? b$) lässt sich keine Regel anwenden.

Die resultierende Menge ist:

$$Eq = \{(a \lessdot_? \text{? super Int}), (a \lessdot_? b)\}$$

Schritt 2 und 3: Aufteilen der Menge Die Menge wird aufgeteilt in eine Menge $Eq_1^{'}$ in der beide Terme des Paares Typvariablen sind und eine Menge $Eq_2^{'} = Eq \setminus Eq_1^{'}$. Dieser Schritt findet innerhalb der `unify`-Methode statt.

$$Eq_1^{'} = \{(a \lessdot_? b)\}$$
$$Eq_2^{'} = \{(a \lessdot_? \text{? super Int})\}$$

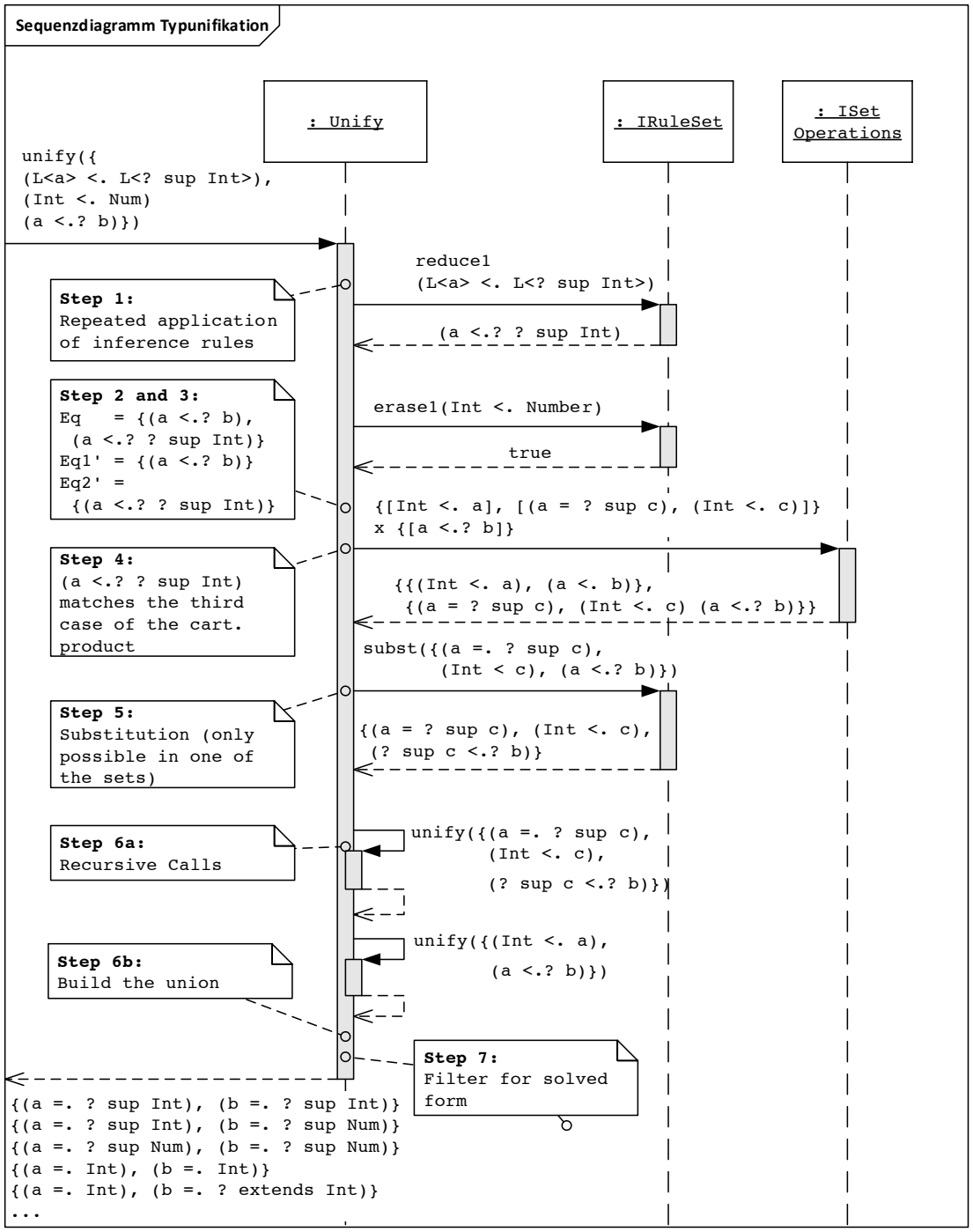

Abbildung 4.11: Sequenzdiagramm Typunifikation

Schritt 4: Kartesisches Produkt Es wird das kartesische Produkt mithilfe der `ISetOperations`-Strategie berechnet. Das Paar $(a \lessdot_? ?\text{ super Int})$ entspricht dem dritten Fall des kartesischen Produktes. Somit führt `ISetOperations` folgende Berechnung aus:

$$\begin{aligned} Eq'_{set} =& \{[\text{Int} \lessdot a], [(a \doteq ?\text{ super } c), (\text{Int} \lessdot c)]\} \times \{Eq'_1\} \\ =& \{[\text{Int} \lessdot a], [(a \doteq ?\text{ super } c), (\text{Int} \lessdot c)]\} \times \{\{(a \lessdot_? b)\}\} \\ =& \{\{(\text{Int} \lessdot a), (a \lessdot_? b)\}, \{(a \doteq ?\text{ super } c), (\text{Int} \lessdot c), (a \lessdot_? b)\}\} \end{aligned}$$

Schritt 5: Substitution Beide Untermengen des Ergebnisses aus Schritt vier werden an die `subst`-Regel des `IRuleSets` übergeben. Diese kann allerdings nur auf eine der Mengen angewandt werden, daher enthält das Sequenzdiagramm nur die erfolgreiche Anwendung.

Die Regel substituiert a mit ? super c wodurch diese Menge Eq' entsteht:

$$Eq' = \{(a \doteq ?\text{ super c}), (\text{Int} \lessdot c), (?\text{ super c} \lessdot_? b)\}$$

Schritt 6a: Rekursiver Aufruf Die Mengen sind nicht in gelöster Form, wodurch rekursiver Aufrufe von `unify` stattfinden. Das Ergebnis der Aufrufe ist:

$$\begin{aligned} &unify(\{(a \doteq ?\text{ super c}), (\text{Int} \lessdot c), (?\text{ super c} \lessdot_? b)\}) = \{ \\ &\quad \{(a \doteq ?\text{ super Int}), (c \doteq \text{Int}), (b \doteq ?\text{ super Int})\}, \\ &\quad \{(a \doteq ?\text{ super Int}), (c \doteq \text{Int}), (b \doteq ?\text{ super Num})\}, \\ &\quad \{(a \doteq ?\text{ super Num}), (c \doteq \text{Num}), (b \doteq ?\text{ super Num})\}\} \\ &unify((\text{Int} \lessdot a), (a \lessdot_? b)) = \{ \\ &\quad \{(a \doteq \text{Int}), (b \doteq \text{Int})\}, \{(a \doteq \text{Int}), (b \doteq ?\text{ extends Int})\} \\ &\quad \{(a \doteq \text{Int}), (b \doteq ?\text{ super Int})\}, \{(a \doteq \text{Int}), (b \doteq ?\text{ extends Num})\} \\ &\quad \{(a \doteq \text{Num}), (b \doteq \text{Num})\}, \{(a \doteq \text{Num}), (b \doteq ?\text{ extends Num})\} \\ &\quad \{(a \doteq \text{Num}), (b \doteq ?\text{ super Int})\}, \{(a \doteq \text{Num}), (b \doteq ?\text{ super Num})\}\} \end{aligned}$$

Schritt 6b: Vereinigung Die Vereinigung ist der Mengen aus 6a.

Schritt 7: Filtern nach gelöster Form Alle Untermengen befinden sich in gelöster Form, somit ist das Ergebnis der Unifikation:

$$
\begin{aligned}
Uni = &\{(a \mapsto ?\text{ super Int}), (c \mapsto \text{Int}), (b \mapsto ?\text{ super Int})\},\\
&\{(a \mapsto ?\text{ super Int}), (c \mapsto \text{Int}), (b \mapsto ?\text{ super Num})\},\\
&\{(a \mapsto ?\text{ super Num}), (c \mapsto \text{Num}), (b \mapsto ?\text{ super Num})\},\\
&\{(a \mapsto \text{Int}), (b \mapsto \text{Int})\}, \{(a \mapsto \text{Int}), (b \mapsto ?\text{ extends Int})\}\\
&\{(a \mapsto \text{Int}), (b \mapsto ?\text{ super Int})\}, \{(a \mapsto \text{Int}), (b \mapsto ?\text{ extends Num})\}\\
&\{(a \mapsto \text{Num}), (b \mapsto \text{Num})\}, \{(a \mapsto \text{Num}), (b \mapsto ?\text{ extends Num})\}\\
&\{(a \mapsto \text{Num}), (b \mapsto ?\text{ super Int})\}, \{(a \mapsto \text{Num}), (b \mapsto ?\text{ super Num})\}\}
\end{aligned}
$$

4.7 Unit-Tests

Unit-Tests dienen zur Verifizierung der Implementierung. Bei Änderungen an der Implementierung kann durch Ausführen der Unit-Tests schnell überprüft werden, ob die Implementierung der Spezifikation genügt. Dabei ist zu beachten, dass die Korrektheit der Implementierung durch Unit-Tests nicht gewährleistet werden kann. Die Tests können lediglich dazu dienen fehlerhafte Fälle zu entdecken.

Für möglichst effektive Tests, müssen die Testfälle so gewählt werden, dass eine möglichst breite Testabdeckung erreicht wird, d.h. möglichst viele verschiedene Pfade durch die Implementierung getestet werden.

Unit-Tests existieren für alle Bestandteile die wesentliche Logik enthalten, also für die FC(<), Standard-Unifikation, Inferenzregeln und Typunifikation.

Probleme Aufgrund der exponentiell wachsenden Anzahl an Lösungen der Typunifikation sind Unit-Tests nur eingeschränkt nutzbar. Insbesondere bei den Funktionen der FC(<) sowie der Typunifikation selbst, ist es aufwendig manuell alle Lösungen zu berechnen und einen entsprechenden Unit-Test zu schreiben. Daher können nur kleine Eingabegrößen vollständig verifiziert werden. Bei mittleren und großen Problemen, ist die Anzahl an

Lösungen meist so groß, dass nur das Vorhandensein einzelner Paare geprüft werden kann.

Außerdem muss bei Unit-Tests darauf geachtet werden, dass neu eingeführte Typvariablen einen zufallsgenerierten Namen besitzen. Diese müssen bei der Verifizierung einer Lösung ignoriert werden.

4.8 Parallelisierung

Für die Parallelisierung bieten sich verschiedene Ansätze an. Ein naiver Ansatz ist es, für jeden rekursiven Aufruf im sechsten Schritt einen neuen Thread zu starten. Eine weitere Möglichkeit ist die Parallelisierung mit Javas `parallelStream`. In der Praxis zeigt sich aber, dass beide Ansätze sogar langsamer als eine serielle Ausführung sind. Dies liegt

Die dritte Möglichkeit ist die Nutzung eines Fork-Join-Pools, welcher sich sehr gut zur Parallelisierung rekursiver Algorithmen eignet. Durch die begrenzte Anzahl von Threads im Pool, bleibt der Overhead zur Erstellung neuer Threads gering. Unter Nutzung des Fork-Join-Ansatzes zeigt sich eine wesentliche Verbesserung der Laufzeit des Algorithmus.

In Codebeispiel 4.2 ist die Verwendung von Fork-Join zur Parallelisierung des Unifikationsalgorithmus grob skizziert. Der Unifikationsalgorithmus erbt dazu von `RecursiveTask`, wodurch er in einem Fork-Join-Threadpool verwendet werden kann.

```java
// Wrapper class so the task can be called like a method
public class TypeUnify {
    public Set<Set<UnifyPair>> unify(Set<UnifyPair> eq, IFiniteClosure
        fc) {
      TypeUnifyTask unifyTask = new TypeUnifyTask(eq, fc); // Create
          task
      ForkJoinPool pool = new ForkJoinPool(); // Create threadpool
      pool.invoke(unifyTask); // Invoke the task
      return unifyTask.join(); // Wait for answer
    }
}

// The parallelizable task
public class TypeUnifyTask extends RecursiveTask<Set<Set<UnifyPair>>> {

    @Override
    protected Set<Set<UnifyPair>> compute() {
        return unify(eq, fc);
    }

    public Set<Set<UnifyPair>> unify(Set<UnifyPair> eq, IFiniteClosure
        fc) {
      // Do Step 1 to Step 5 of the algorithm

      // Step 6a: Fork (Recursive Call)
      Set<TypeUnifyTask> forks = new HashSet<TypeUnifyTask>();
      for (EqS from EqSset where EqS != eq) {
        TypeUnifyTask fork = new TypeUnifyTask(EqS, fc);
        forks.add(fork);
        fork.fork(); // Parallelize the task
      }

      // Step 6b: Join (Build the union)
      forks.forEach(fork -> EqSSSet.addAll(fork.join()));

      // Do Step 7
    }
}
```

Codebeispiel 4.2: Parallelisierung mit Fork-Join

Kapitel 5

Fazit

5.1 Rückblick

Der Typunifikationsalgorithmus wurde um bisher nicht beachtete Fälle erweitert, aufgrund derer bestimmte Lösungen nicht gefunden wurden. Dafür wurden neue Inferenzregeln für Wildcards und Typvariablen eingefügt. Außerdem wurde die Berechnung des kartesischen Produktes angepasst. Durch die Änderungen konnte der Algorithmus, insbesondere die Berechnung des kartesischen Produktes, vereinfacht werden.

Die Neuimplementierung unterstützt nun auch echte Java-Funktionstypen. Die Implementierung wurde nach softwaretechnischen Prinzipien entwickelt, sodass Erweiterungen und Anpassungen in der Zukunft leichter durchzuführen sind. Laufzeit und Speicherplatzverbrauch erheblich gesenkt werden, sodass die Typunifikation nun effizienter ist. Dies wurde vor allem durch unveränderliche Datenstrukturen erreicht. Obwohl die Effizienz erheblich gesteigert wurde, kann die Unifikation bei mittleren und großen Java-Programmen immer noch viel Zeit in Anspruch nehmen.

Ein objektiver Vergleich der Implementierungen in einem Benchmark fällt allerdings schwierig aus. Das hat einerseits den Grund, dass die neue Implementierung exponentiell mehr Lösungen findet, und somit für das gleiche Problem mehr leistet, andererseits ist es sehr aufwändig eine angemessen große Testbasis von Hand zu erstellen.

5.2 Ausblick

Ein Ziel ist es, die Laufzeit und Effizienz der Implementierung weiter zu verbessern. Dazu könnten z.B. durch Profiling-Tools besonders rechenintensive Operationen ermittelt und gezielt verbessert werden. Außerdem kann die Laufzeit möglicherweise durch weitere Parallelisierung, z.B. des kartesischen Produktes verbessert werden.

Der Typunifikationsalgorithmus selbst kann in Zukunft noch um gebundene Typvariablen erweitert werden. Diese entstehen z.B. durch Klassendeklarationen der Form

```
1 class X<T extends B1 & B2 & ... >
```

Codebeispielverzeichnis

Abbildungsverzeichnis

Symbolverzeichnis

$\perp$	Zeichen für undefiniert.
$\vdash$	Typurteil, welches einem Wert einen Typ zuordnet.
$\mapsto$	Zuordnung eines Wertes des Definitions- zu einem Wert des Wertebereichs.
σ	$\sigma = (x \mapsto y)$ ist eine Funktion die in einem Term alle Variablen x mit dem Term y ersetzt.
MGU	Der allgemeinste Unifikator zweier Terme.
$\leq^*$	Die Subtyp-Relation.
$\lessdot$	Zeigt an, dass ein Paar $(\theta \lessdot_? \theta')$ so unifiziert werden soll, dass $\theta \leq^* \theta'$.
$\lessdot_?$	Zeigt an, dass ein Paar $(\theta \lessdot \theta')$ so unifiziert werden soll, dass $\theta \in \mathbf{smArg}(\theta')$.
$\doteq$	Zeigt an, dass ein Paar $(\theta \doteq \theta')$ so unifiziert werden soll, dass $\theta = \theta'$.
$<$	$(\theta < \theta')$ gibt an, dass θ ein Subtyp von θ' ist.
$\mathbf{FC}(<)$	Der transitive und reflexive Abschluss der $(<)$-Relation.

θ	Ein Typterm.
${}_?\theta$	Extends-Wildcard von θ, ? extends θ.
${}^?\theta$	Super-Wildcard von θ, ? super θ.
Γ	Eine Umgebung in welcher Typurteile deduziert werden.
TV	Menge der Typvariablen.
smaller	Funktion die alle Subtypen berechnet (Definition siehe Abschnitt 3.2).
greater	Funktion die alle Supertypen berechnet (Definition siehe Abschnitt 3.2).
smArg	Funktion die kleinere Typen im Kontext eines Typparameters berechnet (Definition siehe Abschnitt 3.2).
grArg	Funktion die größere Typen im Kontext eines Typparameters berechnet (Definition siehe Abschnitt 3.2).

Glossar

Allgemeinster Unifikator

↑Unifikator der am wenigsten ↑Substitutionen enthält.

Extendstyp

Typ der Form ? extends θ, der dazu dient ↑Kovarianz herzustellen.

Funktionstyp

Implizit ↑varianter Typ von Funktionen.

Invarianz

Eigenschaft von Typen keine ↑Varianz zu besitzen.

Kontravarianz

Kontravariante Typen stehen in der entgegengesetzten Subtyp-Beziehung wie ihre Parameter.

Kovarianz

Kovariante Typen stehen zueinander in der selben Subtyp-Beziehung wie ihre Parameter.

Substitution

Eine Substitution ersetzt eine ↑Typvariable durch einen ↑Typterm.

Supertyp

Typ der Form ? super θ, der dazu dient ↑Kontravarianz herzustellen.

Typ

Eine Einschränkung der möglichen Werte einer Variablen.

Typinferenz

Das Herleiten von Typen nach den Regeln des ↑Typsystems.

Typkorrektheit

Die Eigenschaft eines Programmes frei von Laufzeit-Typfehler zu sein.

Typsystem

Ein Satz an Regeln deren Einhaltung die ↑Typkorrektheit sicherstellt..

Typterm

↑Typ oder ↑Typvariable.

Typunifikation

Verallgemeinerung der Unifikation bei der nicht nur nach Gleichheit unifiziert wird, sondern Ungleichungen gültig gemacht werden.

Typvariable

Eine Variable aus der Menge der Typen.

Unifikator

Menge an ↑Substitutionen die zwei Typterme identisch macht..

Unit-Test

Ein Unit-Test ist Code, welcher anderen Code ausführt und dabei die Korrektheit vorher getroffener Annahmen überprüft. Treffen diese zu, wurde der Test bestanden [Osh09, S.4].

Varianz

↑Kovarianz oder ↑Kontravarianz.

Wildcardtyp

↑Extendstyp oder ↑Supertyp.

Literaturverzeichnis

[Car96] Luca Cardelli. “Type Systems”. In: *ACM Comput. Surv.* 28.1 (März 1996), S. 263–264. ISSN: 0360-0300. DOI: 10.1145/234313.234418. URL: http://doi.acm.org/10.1145/234313.234418.

[CW85] Luca Cardelli und Peter Wegner. “On Understanding Types, Data Abstraction, and Polymorphism”. In: *ACM Comput. Surv.* 17.4 (Dez. 1985), S. 471–523. ISSN: 0360-0300. DOI: 10.1145/6041.6042. URL: http://doi.acm.org/10.1145/6041.6042.

[Gam+96] Erich Gamma et al. *Entwurfsmuster*. Addison Wesley, 1996. ISBN: 3827318629.

[Gos+15] James Gosling et al. *The Java language specification*. Bd. 8. 2015.

[HV09] Kryštof Hoder und Andrei Voronkov. “KI 2009: Advances in Artificial Intelligence: 32nd Annual German Conference on AI, Paderborn, Germany, September 15-18, 2009. Proceedings”. In: Hrsg. von Bärbel Mertsching, Marcus Hund und Zaheer Aziz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. Kap. Comparing Unification Algorithms in First-Order Theorem Proving, S. 435–443. ISBN: 978-3-642-04617-9. DOI: 10.1007/978-3-642-04617-9_55. URL: http://dx.doi.org/10.1007/978-3-642-04617-9_55.

[MM82] Alberto Martelli und Ugo Montanari. “An Efficient Unification Algorithm”. In: *ACM Trans. Program. Lang. Syst.* 4.2 (Apr. 1982), S. 258–282. ISSN: 0164-0925. DOI: 10.1145/357162.357169. URL: http://doi.acm.org/10.1145/357162.357169.

[Ora16] Oracle. *A Strategy for Defining Immutable Objects*. https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html. Abruf am 17.04.2016. 2016.

[Osh09] Roy Osherove. *The art of unit testing*. Manning Publications Co., 2009. ISBN: 9781617290893.

[PW76] M. S. Paterson und M. N. Wegman. "Linear Unification". In: *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*. STOC '76. Hershey, Pennsylvania, USA: ACM, 1976, S. 181–186. DOI: 10.1145/800113.803646. URL: http://doi.acm.org/10.1145/800113.803646.

[Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002. ISBN: 9780262162098.

[Plü07] Martin Plümicke. "Java Type Unification with Wildcards". In: *Applications of Declarative Programming and Knowledge Management, 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*. 2007, S. 223–240. DOI: 10.1007/978-3-642-00675-3_15. URL: http://dx.doi.org/10.1007/978-3-642-00675-3_15.

[Plü15a] Martin Plümicke. "Java Type System – Proposals for Java 10 or 11". In: *Tagungsband des 18. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS'15)*. Hrsg. von Jens Knoop. Technische Berichte der TU Wien. Pörtschach, 2015.

[Plü15b] Martin Plümicke. "More Type Inference in Java 8". In: *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*. Hrsg. von Andrei Voronkov und Irina Virbitskaite. Bd. 8974. Lecture Notes in Computer Science. Springer, 2015, S. 248–256.

[PB06] Martin Plümicke und Jörg Bäuerle. "Typeless programming in Java 5.0". In: *4th International Conference on Principles and Practices of Programming in Java*. Hrsg. von Ralf Gitzel et al. Bd. 178. ACM International Conference Proceeding Series. Mannheim University Press, Aug. 2006, S. 175–181.

[RV01] Alan JA Robinson und Andrei Voronkov. *Handbook of automated reasoning*. Bd. 1. Elsevier, 2001. ISBN: 0444508139.

[Rob65] J. A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *J. ACM* 12.1 (Jan. 1965), S. 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253. URL: http://doi.acm.org/10.1145/321250.321253.