Erweiterung und Neuimplementierung der Java Typunifikation

Florian Steurer und Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
florian.steurer95@gmail.com, pl@dhbw.de

Zusammenfassung. 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. We gave a type inference algorithm for Java. This algorithm uses a constraint-based type unification, which, for a given set of type constraints, computes all possible typings of a program. The type unification algorithm has been implemented as part of a compiler, which supports implicitly typed Java programs.
In this paper the Java type unification is extended by real function types. Additionally, some cases concerning wildcard types are added, that were not considered before. 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 and parallelism.

1 Einführung

In Java müssen bisher alle Typen explizit deklariert werden. Mit Typinferenz ist es möglich Typdeklarationen auszulassen, so dass 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 einen constraint-basierten Typunifikationalgorithmus [Plü07], welcher, für eine Menge an Constraints alle möglichen Typisierungen eines Java-Programms berechnet.

In [LP07] wurde dieser Algorithmus implementiert. Die Implementierung wird in einem Compiler [Sta15] 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.

Durch eine Neuimplementierung des Typunifikationsalgorithmus soll die Effizienz des Algorithmus verbessert werden, so dass 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 wurden.

2 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 2.2 ergänzend zu [Plü07] die Funktionen **smaller**, **greater**, **grArg** und **smArg** definiert und an Beispielen erläutert.

Im darauffolgenden Abschnitt 2.3 werden Informationen zu echten Funktionstypen in Java [Plü15a] zusammengefasst.

In Abschnitt 2.4 wird der Typunifikationsalgorithmus um zusätzliche Inferenzregeln ergänzt.

Abschnitt 2.5 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 2.6 behandelt.

Zuletzt folgt eine Zusammenfassung der durchgeführten Erweiterungen am Typunifikationsalgorithmus in Abschnitt 2.7.

2.1 Beschreibung

Die klassische Unifikation [Rob65,MM82] 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 $\theta^{'}$ 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 [MM82].
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.

2.2 Funktionen auf Typen

Sei $<$ die Extends–Relation eines java–Programms und $\leq^*$ die daraus resultierende Subtyp–Relation. 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 1 (smaller). *Sei $\leq^*$ die transitive und reflexive Subtyp-Relation auf Extended-Typen wie sie in [Plü07] definiert ist. Dann ist*

$$\boldsymbol{smaller}(\theta) = \{x \mid x \leq^* \theta, \nexists \overline{x} \leq^* \theta, \nexists \textit{ Subsitution } \sigma \textit{ mit } \sigma(\overline{x}) = x\}.$$

Die Funktion $\boldsymbol{smaller}(\theta)$ *berechnet die Menge aller allgemeinsten*[1] *Typen die in der Subtyp-Relation* $\leq^*$ *kleiner als* θ *sind.*

Beispiel 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 2 (greater). *Sei* $\leq^*$ *die transitive und reflexive Subtyp-Relation auf Extended-Typen wie sie in [Plü07] definiert ist. Dann ist*

$$\boldsymbol{greater}(\theta) = \{x \mid \theta \leq^* x\}$$

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

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

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

[1] vgl. Example 6 und 7 in [Plü07]

Beispiel 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 4 (grArg). *Die Funktion* $\boldsymbol{grArg}(\theta)$ *berechnet die Menge an Typen* θ' *welche, im Argument eines umschließenden Typs C dafür sorgen, dass* $C\langle\theta\rangle \leq^* C\langle\theta'\rangle$.

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

Beispiel 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 1. *Sei* $<$ *eine Extends-Relation eines Java-Programms mit der Eigenschaft, dass in der daraus resultierenden Subtyp-Relation* $\leq^*$ *keine Elemente* $\theta \leq^* C\langle\ldots, {}_{?}\sigma(\theta), \ldots\rangle$ *(F–bounded Elemente)existieren. Dann sind die Mengen* ***smaller***, ***greater***, ***smArg*** *und* ***grArg*** *endlich.*

Korollar 2. *Sei* $P_{gr}(\theta, \theta') = \theta \in \boldsymbol{greater}(\theta')$ *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. *Es gilt:*

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

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

2.3 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 1.1 zeigt wie sich die Varianz in Java auswirkt.

2.4 Typinferenzregeln

Wildcardtypen Die Inferenzregeln des Typunifikationsalgorithmus werden um weitere vier Regeln aus Abbildung 1 ergänzt um Wildcardtypen zu reduzieren.

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 5. 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

```
Tuple<Number, Number, Number> t1 = new Tuple(1, 2, 3);
Tuple<Integer, Object, Number> t2 = new Tuple(1, 2, 3);
Tuple<? extends Number, ? super Number, ? super Number>
    t3 = null;
t1 = t2; // Not Possible, will not compile.
t3 = t2; // Possible because of explicit wildcard
    variance.

Fun2<Number,Number,Number> multiply = (x) -> (y) -> x*y;
Fun2<Integer, Object, Number> foo =
        (x) -> (y) -> x.toString().length() * y;
multiply = foo; // Possible because of implicit variance
    .
```

Codebeispiel 1.1: Varianz von $FunN*$-Typen

Typen {? extends Number, ? extends Integer, Number, Integer} annehmen. Sei nun die Regel reduceWcLowRight aus Abbildung 1 uneingeschränkt anwendbar.

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

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.

Typvariablen Zusätzlich zu den sieben Wildcard-Inferenzregeln wird der Algorithmus um drei Inferenzregeln für Typvariablen aus Abbildung 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 2.5 beschrieben.

Funktionstypen In Abbildung 2.4 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.

$$\text{(reduceWcLow)} \quad \frac{Eq \cup \{ {}_{?}\theta \lessdot_{?} {}_{?}\theta' \}}{Eq \cup \{ \theta \lessdot \theta' \}}$$

$$\text{(reduceWcLowRight)} \quad \frac{Eq \cup \{ \theta \lessdot_{?} {}_{?}\theta' \}}{Eq \cup \{ \theta \lessdot \theta' \}} \; \theta \notin TV$$

$$\text{(reduceWcUp)} \quad \frac{Eq \cup \{ {}^{?}\theta \lessdot_{?} {}^{?}\theta' \}}{Eq \cup \{ \theta' \lessdot \theta \}}$$

$$\text{(reduceWcUpRight)} \quad \frac{Eq \cup \{ \theta \lessdot_{?} {}^{?}\theta' \}}{Eq \cup \{ \theta' \lessdot \theta \}} \; \theta \notin TV$$

Abb. 1: Typinferenzregeln für Wildcards

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

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

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

Abb. 2: Typinferenzregeln für Typvariablen

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

$$(\text{greater}FunN*)\quad \frac{Eq \cup \{FunN*<\theta,\theta_1^{'},\ldots,\theta_N^{'}> \lessdot a\}}{Eq \cup \{a \doteq FunN*<b^{'},b_1,\ldots,b_N>, \theta \lessdot b^{'}, b_i \lessdot \theta_i^{'}\}}$$

where $b^{'}$, b_i are fresh

$$(\text{smaller}FunN*)\quad \frac{Eq \cup \{a \lessdot FunN*<\theta^{'},\theta_1,\ldots,\theta_N>\}}{Eq \cup \{a \doteq FunN*<b,b_1^{'},\ldots,b_N^{'}>, b \lessdot \theta^{'}, \theta_i \lessdot b_i^{'}\}}$$

where b, $b_i^{'}$ are fresh

Abb. 3: Typinferenzregeln für Funktionstypen
Quelle: [Plü15a]

2.5 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 6. 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 2.2 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:

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

Durch Anwendung der Substitution ergeben sich zwei Unifikatoren:

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

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

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

Beispiel 7. 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 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 2.2 und die Definition der Subtyp-Relation in [Plü15a]). Wie bereits im vorherigen Beispiel, führt dies dazu, dass Lösungen nicht beachtet werden.

Anpassungen

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

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.

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

2.6 Rekursionsfall

In Abschnitt 2.5 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.

2.7 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 2.4 werden Regeln für Wildcards (Abbildung 1), Regeln für Typvariablen (Abbildung 2) und die in [Plü15a] eingeführten Regeln für Funktionstypen (Abbildung 3) ergänzt.

Schritt 4 In Abschnitt 2.5 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 2.6 beschrieben.

3 Implementierung

Dieses Kapitel beschreibt zwei Aspekte der Neuimplementierung des Typunifikationsalgorithmus. Dafür werden in Abschnitt 3.1 zunächst unveränderliche Datentypen und der neue Java-Datentyp `Optional` eingeführt.

Im Abschnitt 3.2 wird schließlich auf die Parallelisierung der Implementierung eingegangen.

3.1 Grundlagen

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.

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

Bei `Optional` handelt es sich um einen generischen Typ. Eine Instanz ist entweder `Optional.empty` oder enthält ein Objekt des entsprechenden Typs.

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.

`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.

3.2 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.

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.

4 Zusammenfassung und Ausblick

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, so dass Erweiterungen und Anpassungen in der Zukunft leichter durchzuführen sind. Laufzeit und Speicherplatzverbrauch erheblich gesenkt werden, so dass 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 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.

Literatur

LP07. Arne Lüdtke and Martin Plümicke. Implementierung eines Typinferenzalgorithmus für Java 5.0. In Walter Dosch, Clemens Grelck, and Annette Stümpel, editors, *Tagungsband des 14. Kolloquiums Programmiersprachen und Grundlagen der Programmierung*, number A–07–07 in Berichte der Institute für Informatik und Mathematik, pages 141–146. Universtät zu Lübeck, 2007. (in german).

MM82. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

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

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*, pages 223–240, 2007.

Plü15a. Martin Plümicke. Java type system – proposals for java 10 or 11. In Jens Knoop, editor, *Tagungsband des 18. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS'15)*, Pörtschach, 2015. Technische Berichte der TU Wien.

Plü15b. Martin Plümicke. More type inference in Java 8. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, volume 8974 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2015.

Rob65. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

Sta15. Andreas Stadelmeier. Java type inference as an eclipse plugin. In *45. Jahrestagung der Gesellschaft für Informatik, Informatik 2015, Informatik, Energie und Umwelt, 28. September - 2. Oktober 2015 in Cottbus, Deutschland*, pages 1841–1852, 2015.