Implementierung eines Typinferenzalgorithmus für Java 5.0

Arne Lüdtke[1] and Martin Plümicke[2]

[1] Eisenmann
Böblingen
arne@g-free.de
[2] University of Cooperative Education Stuttgart/Horb
Department of Information Technology
Florianstraße 15, D–72160 Horb
tel. +49-7451-521142
fax. +49-7451-521190
m.pluemicke@ba-horb.de

Zusammenfassung In diesem Artikel wird ein Java 5.0 Typ–Inferenzsystem vorgestellt. Das System ermöglicht es Programmieren Java 5.0–Programme ohne explizite Typen zu schreiben. Die Typen werden durch das System automatisch berechnet. So erlaubt es das System, analog zu funktionalen Programmiersprachen, statische Typisierung ohne explizte Typen angeben zu müssen.

1 Einleitung

Das Typsystem von Java 5.0 [GJSB05] ist gegenüber den Vorgängerversionen um parametrisierte Typen, Typvariablen, Typterme und Wildcards erweitert worden. Dies hat zur Folge, dass sehr komplexe Typausdrücke entstehen können. Beispielsweise ist

```
Vector<? extends Vector<AbstractList<Integer>>>
```

in Java 5.0 ein korrekter Typ. Programmierern fällt es oftmals schwer zu erkennen, dass Typen dieser Art für bestimmte Java 5.0 Methoden eine korrekte Typisierung wären. Desweiteren gibt es Java 5.0 Methoden, die Durchschnittstypen als allgemeinste Typisierung hätten. Durchschnittstypen sind allerdings in Java 5.0 nicht implementiert. Das heißt oftmals haben Java 5.0 Methoden nicht den allgemeinst möglichen Typ, was dem Ziel wiederverwendbaren Code zu schreiben entgegensteht.
Das hat uns veranlasst ein Typinferenz–System für Java 5.0 zu entwickeln, das den Benutzer durch automatische Typberechnungen unterstützt. Typinferenz in Java 5.0 ermöglicht es, Parameter von Methoden und lokale Variablen ungetypt zu deklarieren. Der Typinferenz–Algorithmus berechnet dann jeweils den allgemeinsten Typ.
Der Artikel ist wie folgt organisiert. Im zweiten Abschnitt wird die Theorie des Typinferenz–Algorithmus beschrieben. Im dritten Abschnitt werden Ideen der Implementierung dargestellt. Dabei wir ein Schwerpunkt auf die Typunifikation

von Typen mit Wildcards gelegt. Der Artikel wird schließlich mit einem Ausblick beendet.

2 Der Typinferenz–Algorithmus

2.1 Typsystem

Die Grundlage des Typinferenz-Systems ist eine Typordnung $\leq^*$, die wir aus der Vererbungshierarchie ableiten. Eine ganz wesentliche Eigenschaft der Typordnung ist, dass bei zwei Typtermen $\theta \leq^* \theta_2$ nur die äußersten Klassen voneinander erben dürfen. Die Argumente der Typterme müssen identisch sein oder durch sogenannte Wildcards zur möglichen Vererbung gekennzeichnet werden. Ohne diese Eigenschaft wäre das Typsystem von Java 5.0 nicht typsicher [Plü07a].
Das Java 5.0 Typsystem hat große Ähnlichkeit mit polymorphic order–sorted Typsystemen von logischen Programmiersprachen (z.B. [Smo89]).

2.2 Typ–Unifikation

Basis des Typinferenz–Algorithmus ist die Typ–Unifikation. Das Typ–Unifikations–Problem stellt sich wie folgt dar: Für zwei gegebene Typen θ_1 und θ_2 ist eine Substitution gesucht, so dass $\sigma(\theta_1) \leq^* \sigma(\theta_2)$ gilt. Es zeigt sich, dass die Typ–Unifikation für Java 5.0 Typen ohne und mit Wildcards finitär ist.
Der Typ–Unifikations–Algorithmus für Typen ohne Wildcards [Plü04] ist genauso wie der Typ–Unifikations–Algorithmus für Typen mit Wildcards [Plü07b] eine Erweiterung des Algorithmus aus [MM82]. Die wesentliche Erweiterung in beiden Fällen liegt darin, dass, wenn man im ursprünglichen Algorithmus $a = \theta$ erhält, so weiss man, dass der Variable a der Term θ zugeordnet wird. Wenn man dagegen in der Erweiterung $a \leq^* \theta$ bzw. $\theta \leq^* a$ erhält, so kann man a alle Terme zuordnen die kleiner bzw. größer als θ sind. Diese Eigenschaft ist auch die Ursache dafür, dass es in bestimmten Fällen mehrere Unifikatoren gibt.

2.3 Typinferenz in Java 5.0

Zunächst definieren wir Typinferenz–Regeln für die abstrakte Syntax eines Kerns von Java 5.0. Dann zeigen wir die *principal type property*. Der allgemeinste Durchschnittstyp einer Methode ist der Durchschnitt aller verschiedenen allgemeinsten Typen die jeweils einzeln in Java 5.0 für die Methode ausdrückbar sind.
Aus den Typinferenz–Regeln wird dann ein Typinferenz–Algorithmus entwickelt. Der Typinferenz–Algorithmus basiert auf der Typ–Unifikation. Zunächst werden für alle zu berechnenden Typen Typvariablen als Typen angenommen. Dann werden sukzessive durch Unifikation die Typen verfeinert. Wenn die Unifikation mehrere Unifikatoren als Ergebnis liefert, wird für den zugehörigen Typ für jeden Unifikator jeweils eine Annahme gemacht. Schlägt eine Unifikation fehl, wird die zugehörige Annahme gelöscht. Alle am Ende nicht ersetzten Typvariablen werden generalisert. Besteht am Ende das Ergebnis aus verschiedenen Typannahmen, so ist ein Durchschnittstyp inferiert worden.

3 Implementierung

3.1 Typinferenz für Typen ohne Wildcards

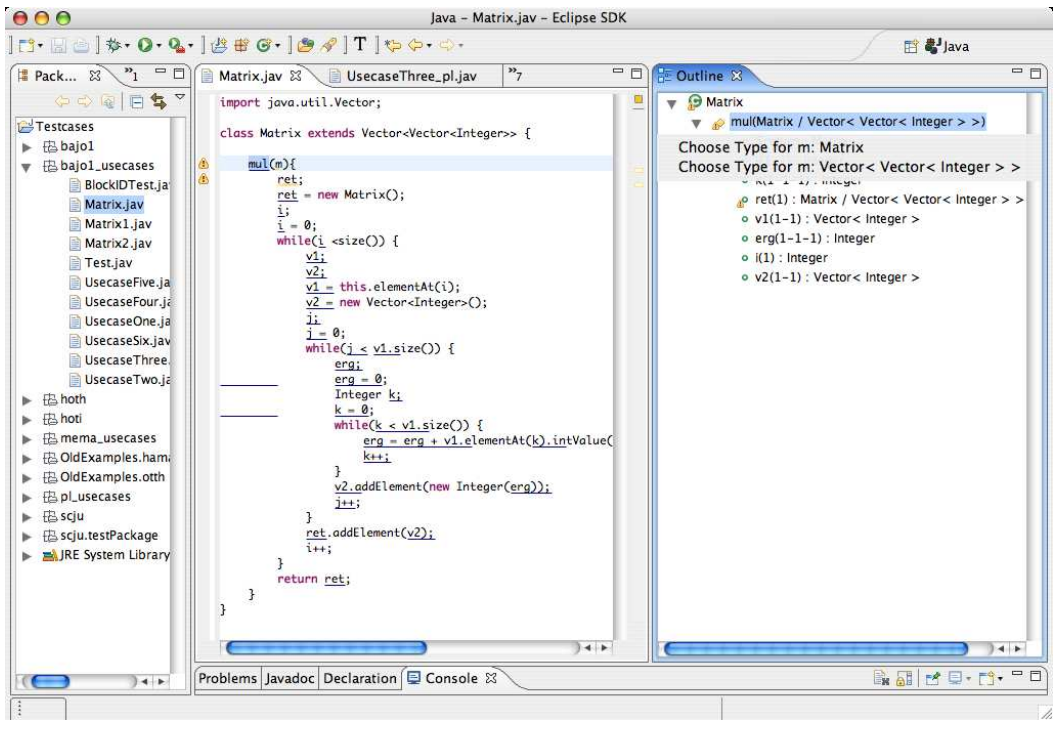

Abbildung 1. Eclipse Plugin für die Java 5.0 Typinferenz

Ein Prototyp eines Java 5.0 Compilers mit dem Typinferenz–Algorithmus ohne Wildcards ist implementiert. Die Implementierung wurde in ein Eclipse PlugIn (Abb. 1) integriert, so dass eine benutzerfreundliche Bedienbarkeit gegeben ist. Das GUI erlaubt es, sich die inferierten Typen anzeigen zu lassen und im Falle der Inferenz eines Durchschnittstyps, einen davon als gewünschten Typ auszuwählen.

3.2 Typunifikation für Typen mit Wildcards

Der Input des Algorithmus ist eine Menge von Paaren. Ein Paar ist dabei eine Collection von genau zwei Typen, welche in einem gewissen Verhältnis zueinander stehen. Dieses Verhältnis entscheidet, ob die beiden Typen nur voneinander erben, oder ob diese Identisch sein sollen. Als Ausgabe liefert der Algorithmus eine Menge von Paaren in der sogenannten *Solved Form*. Das bedeutet, die Paare bestehen aus einer Typvariable und einem Typ, und das Verhältnis ist, dass diese gleich sind.
Die Funktionsweise des Algorithmus ist im Wesentlichen, immer wieder durch die Eingabemenge zu laufen und verschiedene Regeln auf die Menge anzuwenden. Dies macht der Algorithmus solange bis keine Änderung mehr vorkommt. Die sieben Schritte des Algorithmus sind im folgenden kurz beschrieben. Eine formale Definition ist in [Plü07b] zu finden.

1. **Schritt:** Im ersten Schritt werden erweiterte Regeln der vier Regeln aus [MM82] auf die Eingabemenge angewendet, solange, bis keine Regel mehr angewendet werden kann. Die Ergebnismenge ist Eq.
2. **Schritt:** Im zweiten Schritt werden alle Paare aus der Menge Eq aussortiert, die entweder zwei Typvariablen enthalten, oder die keine Typvariablen enthalten. Diese Paare werden in die Menge Eq1 eingefügt.
3. **Schritt:** In diesem Schritt wird die Menge Eq2 erzeugt. Diese Menge ist die Differenz von Eq zu Eq1.
4. **Schritt:** In diesem Schritt wird die Menge Eq2 durchlaufen, und für die Paare der Form $(a < \theta)$ bzw. $(\theta < a)$ wird jeweils die Menge der Subtypen bzw. Supertypen von θ gebildet. Über diese Ergebnisse wird das Kreuzprodukt gebildet.
5. **Schritt:** Mit der Subst Regel [MM82] werden alle Typvariablen, in den Ergebnismengen aus Schritt 4 ersetzt.
6. **Schritt:** In diesem Schritt werden alle Ergebnissätze geprüft. Für Sätze, die in Regel 5 geändert wurden, wird neu mit dem 1. Schritt begonnen. Für die anderen wird eine Vereinigung mit einer Ergebnismenge durchgeführt.
7. **Schritt:** Aus den berechneten Mengen werden alle aussortiert, die in Solved Form sind.

Für den 2. Schritt werden zwei Funktionen *greater* und *smaller* implementiert:

Die Funktion greater: Die Funktion greater liefert die Menge aller Typen welche in der Vererbung größer sind, als der übergebene Typ. Die Funktion arbeitet in drei Unterfunktionen (greater1 – greater3), die jeweils einen Teil der Ergebnismenge erstellen. Die Vereinigung dieser drei Mengen ist das eigentliche Ergebnis. greater1 wendet die Funktion greater rekursiv auf die Parameter an. Anschließend werden alle Permutationen mit den Ergebnismengen der Parameter erzeugt. greater2 ermittelt über die *extends* Hierachie des Java–Programms alle größeren Typen. greater3 wendet greater1 auf alle Typen in greater2 an.

Die Funktion smaller: Die Funktion smaller wird verwendet um für einen gegebenen Typ die Menge aller Typen zu finden, welche in der Vererbungshierarchie kleiner sind, also von dem Ausgangstyp erben. Die Funktion arbeitet dabei in vier Teilschritten (smaller1 – smaller4). Diese Unterfunktionen erzeugen jeweils eine Teilmenge von smaller. Am Ende wird die Vereinigung dieser Teilmengen gebildet. Diese Vereinigung ist das Ergebnis. smaller1 ermittelt die Menge aller Typen durch rekursiven Aufruf der Funktion smaller auf den Parametern, und anschließende Permutation dieser. smaller2 bildet die Capture Conversion [GJSB05] über die Ergebnismenge von smaller1. smaller3 ermittelt über die *extends* Hierachie des Java–Programms alle Typen die kleiner sind. smaller4 wandelt die Capture Conversions, die in smaller2 erzeugt wurden, zurück.

Imlementierung der inferierten Typen Für die Implementierung der Typen mit Wildcards wurden drei neue Klassen angelegt (Abb. 2), welche die drei

Wildcard Typen darstellen. Die Klasse WildcardType steht für die allgemeine Wildcard und erbt von der Basisklasse Type. Von dieser Klasse wiederum erben die Klassen ExtendsWildcardType und SuperWildcardType. In den Klassen sind jeweils die Eigenschaften gespeichert, welche notwendig sind, um die Typen darzustellen. Zusätzlich gibt es noch drei weitere Klassen. Diese FreshWildcardType Klassen können vom Benutzer nicht deklariert werden. Im Rahmen der Unifikation werden Wildcard Types durch die Capture Conversions in FreshWildcard Types umgewandelt und wieder zurück. Dadurch wird verhindert, dass die Regeln an den falschen Stellen aufgerufen werden. Diese Klassen werden in den Funktionen smaller und greater erzeugt, werden aber aus der Ergebnismenge gelöscht, bevor diese zurückgegeben wird.

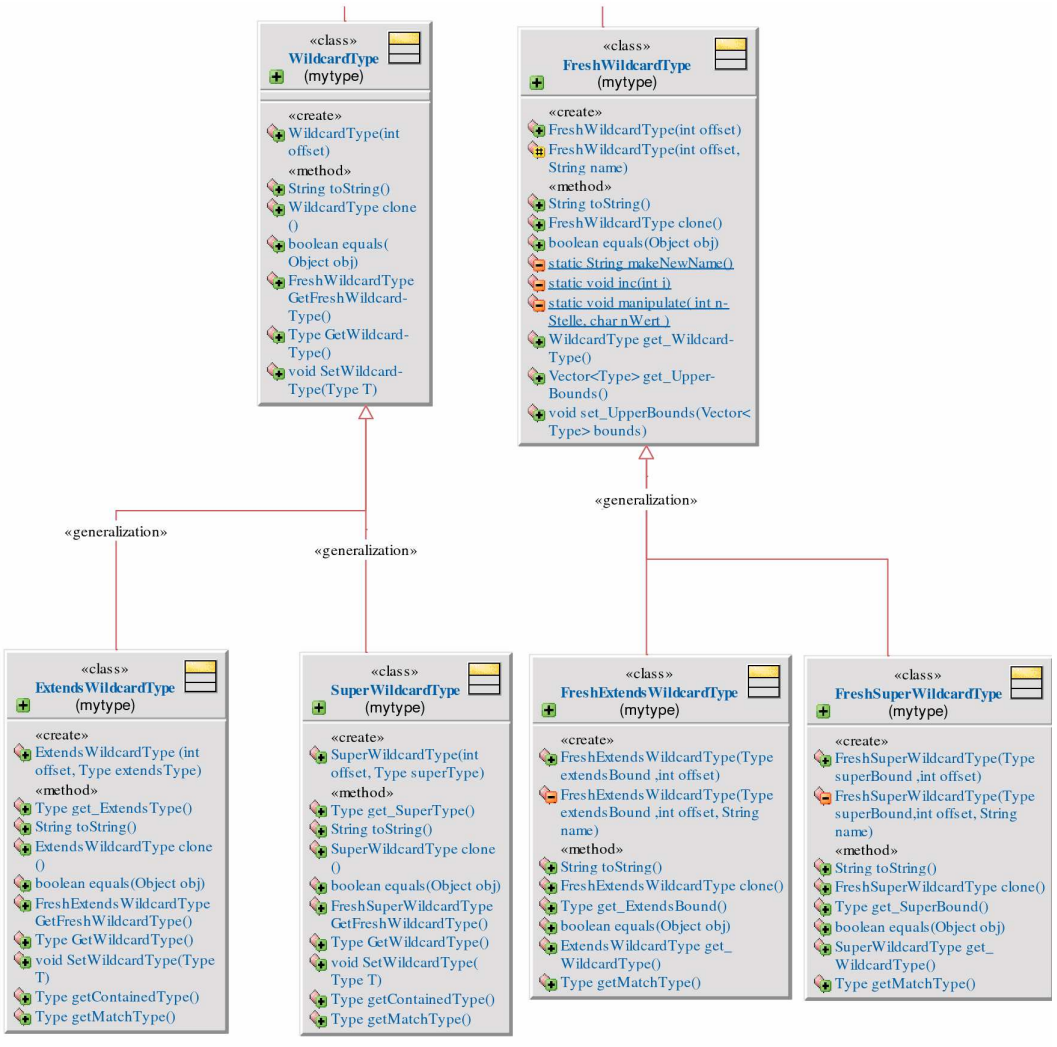

Abbildung 2. Klassendiagramm der Typimplementierung

3.3 Typinferenz für Typen mit Wildcards

Der Typinferenz–Algorithmus für Typen mit Wildcards entspricht dem für Typen ohne Wildcards. Lediglich die jeweiligen Unifikationsalgorithmen unterschei-

den sich. Die Typunifikation für Typen mit Wildcards wurde in den Typinferenz–Algorithmus und die dazugehörige GUI integriert. Für kleinere Programme funktioniert der Algorithmus problemlos. Wenn die Methoden allerdings größer sind und viele Verschachtelungen beinhalten wird das Laufzeitverhalten ungünstig. Ursache ist, dass durch die mögliche Überladung, einzelne Programmteile mehrfach durchlaufen werden. Dieses führt zu exponentiell wachsenden Laufzeiten über die Länge des Programms.
Um das System auch für größere Programme mit Wildcard–Typen sinnvoll nutzbar zu machen, sind einige Optimierungen des Typinferenz–Algorithmus notwendig. Als Ansatz sind dabei die mehrfach durchlaufenen Programmteile anzusehen. Manche Programmteile müssten nicht mehrfach durchlaufen werden, da die Berechnung keine neuen Ergebnisse bringt.

4 Ausblick

Als erstes sollen die Optimierungen des Typinferenz–Algorithmus angegangen werden, um das Laufzeitverhalten deutlich zu verbessern.
Desweiteren werden derzeit alle Methoden als Methoden betrachtet, die sich wechselseitig rekursiv aufrufen können. Da dies eher selten der Fall ist, soll ein Aufrufgraph steuern, in welcher Reihenfolge die Methoden typinferiert werden. Eine weitere Überlegung ist, die Codegenerierung so anzupassen, dass es möglich wird, Methoden mit Durchschnittstypen ohne Auswahl eines bestimmten Typs in Bytecode zu übersetzen.

Literatur

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.

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

[Plü04] Martin Plümicke. Type unification in Generic–Java. In Michael Kohlhase, editor, *Proceedings of 18th International Workshop on Unification (UNIF'04)*, July 2004.

[Plü07a] Martin Plümicke. Formalization of the Java 5.0 type system. In *24. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte*, Bad Honnef, 2. - 4. Mai 2007. Christian–Albrechts–Universität, Kiel. (to appear).

[Plü07b] Martin Plümicke. Java type unification with wildcards. In Évelyne Contejean, editor, *Proceedings of 21th International Workshop on Unification (UNIF'07)*, July 2007.

[Smo89] Gert Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany, May 1989.