Von Typen und Streichen

Parametrizität in Java

Lars Hupel

So gut wie alle modernen Programmiersprachen erlauben es, Routinen über Werte und über Typen zu parametrisieren. In Java wird letzteres „Generics" genannt, in C++ hingegen „Templates". Obwohl die Konzepte die gleichen sind, unterscheidet sich die Implementierung doch stark. In diesem Artikel möchte ich diese Art der Parametrisierung in Java unter die Lupe nehmen und erklären, warum „Type Erasure" doch eine ganz gute Idee ist.

Unter Java-Programmierern hat *Type Erasure* einen schlechten Leumund. In Gesprächen bin ich oft auf Unglauben gestoßen, wenn ich versucht habe, zu erklären, dass es eine Notwendigkeit ist.

Was ist eigentlich Type Erasure?

Doch der Reihe nach: Was ist Type Erasure überhaupt? Java unterstützt seit Version 1.5 die sogenannten *Generics*. So bezeichnet man Methoden, Klassen und Interfaces, die mit einem oder mehreren Typparametern deklariert sind. Das klassische Beispiel dafür ist die *Collections*-Bibliothek:

```
// vor Java 1.5
List listOfThings = new LinkedList();
Thing thing = (Thing) listOfThings.get(0);

// seit Java 1.5
List<Thing> listOfThings = new LinkedList<Thing>();
Thing thing = listOfThings.get(0);
```

Dank Generics erspart man sich eine ganze Reihe von Casts zum erwarteten Typ. Das ist nicht nur eine Erleichterung beim Tippen, sondern auch ganz nebenbei eine Verstärkung des Typsystems. Fehler fallen dem Compiler nämlich sofort auf:

```
List<Dog> listOfDogs = new LinkedList<Dog>();
Cat cat = listOfDogs.get(0);
```

Dieser Code wird vom Compiler völlig zu Recht zurückgewiesen. Trotzdem ist alter Code weiterhin kompatibel: Lässt man die Klammern `<T>` bei der Benutzung weg, funktioniert alles wie gehabt, aber es gibt eine Warnung. Möchte man explizit machen, dass man zum Zeitpunkt der Deklaration noch nicht weiß, welche Typen man in einer Collection verwalten will, so kann man entweder `List<Object>` oder `List<?>` benutzen[1] und der Compiler lässt walten.

Als mit Version 1.5 die Generics kamen, war die Rückwärtskompatibilität eine der größten Diskussionspunkte. Quelltextkompatibilität, wie in den obigen Beispielen zu sehen, ist aber nur die eine Seite. Die andere Seite ist die Laufzeitkompatibilität: Kann ich mit Java 1.4 kompilierte Classfiles auf einer JVM 1.5 laufen lassen?

Diese Frage ist komplizierter als gedacht. Nehmen wir mal hypothetisch an, die JVM 1.5 repräsentiere `List<Dog>` im Speicher an-

1 Der Unterschied zwischen Thing<Object> und Thing<?> ist vorhanden, aber subtil, und ist nicht Thema dieses Artikels.

Photo by Nathan Dumlao on Unsplash

Lars Hupel ist Consultant bei INNOQ in München und einer der Gründer der Typelevel-Initiative, die sich der Entwicklung von typgetriebenen Scala-Bibliotheken in einer einsteigerfreundlichen Umgebung verschrieben hat. Er spricht oft auf Konferenzen und ist im Open-Source-Umfeld in Scala unterwegs. Außerdem programmiert und redet er gern über Haskell, Prolog und Rust. E-Mail: lars.hupel@innoq.com

ders als eine `List`. Ferner nehmen wir an, dass es folgende Methode gäbe, die unter Java 1.4 kompiliert worden sei:

```
public static void reverse(List things) {
  // reverse that list
}
```

Wenn nun Java 1.5-Code `reverse` mit einer `List<Thing>` aufriefe, müsste es zwingend zu einem Laufzeitkonflikt kommen. Es gibt mehrere mögliche Schlussfolgerungen:

- Man verbietet solcherlei Aufrufe, das heißt, man stellt mehr oder weniger „künstlich" eine Inkompatiblität zwischen `List` und `List<T>` her. So ähnlich macht es C#.
- Man repräsentiert `List` und `List<T>` identisch zur Laufzeit. So macht es Java.

Letzterer Ansatz ist, was man gewöhnlich unter Type Erasure versteht.

Warum ist Type Erasure unbeliebt?

Auch diese Frage ist sehr vielschichtig. Naheliegend scheint die Erklärung, dass Type Erasure, wie der Name schon sagt, Informationen wegwirft. Es stehen zur Laufzeit weniger Typinformationen zur Verfügung als zur Compilezeit. Warum sollte man das tun? Ist es nicht besser, mehr Informationen zur Laufzeit zu haben als weniger? Diese Problematik schiebe ich erstmal beiseite; zunächst schauen wir uns die konkrete Information an, die verloren geht.

In Java ist man es gewohnt, mittels *Reflection* Zugriff auf eine ganze Reihe von Laufzeitdaten zu bekommen:

```
Object o = db.getObjectFromSomewhere();
System.out.println(o.getClass()); // class java.lang.String
```

Mittels der `getClass()`-Methode, die auf allen `Object`s zur Verfügung steht, lässt sich zur Laufzeit abfragen, welcher Klasse ein Objekt angehört. Handelt es sich bei dem Objekt um eine Liste, dann sähe die Ausgabe möglicherweise so aus:

```
o.getClass(); // class java.util.LinkedList
```

Dabei unterscheiden sich Java 1.4 und 1.5 übrigens nicht: Egal, welche Compiler-Version diesen Code kompiliert hat, die Ausgabe ist die gleiche. Nur eine `LinkedList`, aber wir kennen den Elementtyp nicht.

Wenn man nun also den Elementtyp herausfinden will, steckt man in der Klemme. Die JVM kann diese Information nicht liefern, denn: `List` und `List<T>` sind zur Laufzeit identisch repräsentiert!

Spätestens an diesem Punkt behaupten dann die meisten, dass Type Erasure ein Fehler war und man es lieber wie C# hätte implementieren sollen. Doch dagegen gibt es zwei gewichtige Argumente: ein theoretisches und ein praktisches.

Klassen und Typen

Weiter oben schrieb ich „Mittels der `getClass()`-Methode, die auf allen `Object`s zur Verfügung steht, lässt sich zur Laufzeit abfragen, welcher Klasse ein Objekt angehört." Wohlgemerkt nicht, welchem *Typ* es angehört. Wo ist aber der Unterschied?

Versetzen wir uns kurz noch mal in die Welt von Java 1.4. Auch damals war es schon möglich, Informationsunterschiede zwischen Compiler und JVM herzustellen.

```
Object o = "Hello, I'm a string!";
o.length();
```

Dieser Code kompiliert nicht. Die JVM weiß, dass es ein String ist. Der Compiler weiß es nicht, darum wird auch der Aufruf der `length()`-Methode untersagt. Genau hierin liegt der Unterschied zwischen *Klassen* und *Typen*. Eine *Klasse* ist etwas, was ein *Objekt* (im Speicher) hat. Ein *Typ* ist etwas, was eine *Variable* (im Code) hat. Darüber hinaus hat natürlich auch ein *Ausdruck* einen Typ.

Angewandt auf das obige Beispiel hat `o` die Klasse `java.lang.String`, aber den Typ `java.lang.Object`. Der Compiler entscheidet grundsätzlich nach Typ, nicht nach Klasse, welche Aufrufe zulässig sind.

Das Standardwerk der Typen von Benjamin Pierce definiert Typsysteme wie folgt:

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. [Pie02]

Im Falle von Java heißt das: Der Compiler kann zur Compile-Zeit prüfen, dass in der JVM keine `ClassCastException` fliegt. Diese Eigenschaft wird auch als *Soundness* bezeichnet.

Bevor wir auf den offensichtlichen Elefanten im Raum zu sprechen kommen, möchte ich noch auf die logische Richtung der Definition hinzuweisen:

- Ein Programm, welches vom Compiler akzeptiert wird, wirft keine `ClassCastException`.
- Ein Programm, welches eine `ClassCastException` wirft, kann also nicht vom Compiler akzeptiert worden sein.

Diese beiden Aussagen sind äquivalent zueinander. Folgende Aussage gilt allerdings nicht:

- Ein Programm, welches vom Compiler nicht akzeptiert wird, wirft zur Laufzeit eine `ClassCastException`.

Obiges Programm würde zur Laufzeit funktionieren, der Compiler akzeptiert es trotzdem nicht.

Nun zum Elefanten. `ClassCastException`s existieren ja nun doch. Aber warum, sollte der Compiler die nicht unterbinden? Der Schlüssel liegt im Namen: class cast exception. Wenn man explizit castet, hebelt man den Compiler aus. Dies ist das logische Äquivalent einer Meinung. Man meint, es an dieser Stelle besser zu wissen als der Compiler. Meist stimmt das auch. Manchmal aber auch nicht.

Typen seit Java 1.5

Vor Java 1.5 galt: Ein Typ entspricht genau einer Klasse (nehmen wir mal die primitiven Typen und Arrays aus). Mit der Einführung von Generics kam es geradezu zu einer kambrischen Explosion von Typen. Plötzlich kann mehr als eine Klasse bei der Bildung eines Typs beteiligt sein. Es gibt auch ganz neue Konstrukte, bei denen Lücken in Typen bleiben können, zum Beispiel in `List<? extends Animal>`. Dank Type Erasure braucht das die JVM nicht zu stören, denn

zur Laufzeit können Klassen weiterhin als einzelnes Objekt repräsentiert werden.

Salopp gesagt bedeuten aber „mehr Typen" auch „komplexere Typüberprüfung". Der Compiler muss härter arbeiten, um Ausdrücke auf ihre Korrektheit zu prüfen. Gilt zum Beispiel, dass einer Variable mit dem Typ `List<? extends Animal>` ein Ausdruck vom Typ `List<Dog>` zugewiesen werden darf? Und umgekehrt? Und was ist eigentlich mit `interface Enum<T extends Enum<T>>`?

Blicken wir noch einmal auf die `ClassCastException`s zurück. Ihre Existenz zeigt ja, dass es manche Programme gibt, bei denen man geschummelt hat. Trotzdem sind sie in gewisser Weise ein doppelter Boden. In C++, wo man mit bestimmten Mitteln beliebig casten kann, bekommt man trotzdem keinen Laufzeitfehler, wenn man es falsch macht. Das Programm zeigt schlicht und ergreifend *undefiniertes Verhalten*. Alles kann passieren, inklusive korrumpierten Speicher oder falscher Ausgabe. Die JVM hingegen bricht das Programm geregelt ab, wenn beim Casten etwas schief läuft. So werden zum Beispiel auf jeden Fall offene Ressourcen geschlossen.

Folglich muss also die JVM wissen, welche Casts zulässig sind und welche nicht. Nehmen wir jetzt noch einmal hypothetisch an, dass das gesamte Typsystem, inklusive Generics, zur Laufzeit existiere. Dann müsste logischerweise die JVM auch wissen, ob eine `List<Dog>` ein Subtyp von `List<? extends Animal>` ist. Klingt nicht weiter spannend, oder? Der Compiler weiß das doch auch?

Leider ist das nicht so einfach. Radu Grigore hat bewiesen, dass dieses Problem unentscheidbar ist:

This paper describes a reduction from the halting problem of Turing machines to subtype checking in Java. It follows that subtype checking in Java is undecidable, which answers a question posed by Kennedy and Pierce in 2007. It also follows that Java's type checker can recognize any recursive language, which improves a result of Gil and Levy from 2016. The latter point is illustrated by a parser generator for fluent interfaces. [Gri17]

Was heißt das konkret?

Theorem 1. It is undecidable whether t <: t' according to a given class table. [Gri17]

Der Operator <: steht hierbei für die Subtyp-Prüfung. Unentscheidbarkeit bedeutet, dass es keinen Algorithmus gibt, der in allen Fällen in endlicher Zeit sagen kann, ob t ein Subtyp von t' ist, oder eben nicht.

Im Falle der JVM würde das also bedeuten, dass sich Programme so konstruieren lassen, dass allein das Casten zu einer Endlosschleife führen würde. Doch das ist nicht alles. Der sogenannte *Bytecode Verifier* prüft beim Laden von Klassen, ob der Bytecode wohlgeformt ist. Auch dieser muss in begrenztem Maße Subtyp-Prüfungen durchführen. Im schlimmsten Falle könnte man also eine JVM unkontrolliert abstürzen lassen, indem man speziell gefertigten Bytecode lädt.

Selbstredend ist es auch nicht toll, wenn der Compiler, der diese Checks ja auch machen muss, abstürzt, aber das ist ein weitaus kleineres Problem als eine abstürzende JVM. Zumal Java nicht die einzige Sprache mit einem unentscheidbaren Typsystem ist.

Was ist eigentlich mit Scala?

Es gibt noch einen ganz praktischen Grund, warum man die JVM nicht dazu zwingen sollte, Subtyp-Prüfungen vorzunehmen. Es geht um *Varianz*.

Ab und zu hat man den Fall, eine Collection von `T` als Collection von `U` aufzufassen, sofern `T` ein Subtyp von `U` ist. Dies nennt man *Kovarianz*. Zum Beispiel ist eine `List<Dog>` auch eine `List<Animal>`. Formal gesprochen lässt sich dies so ausdrücken: t <: t' ==> List t <: List t'. Daher kommt der Begriff Kovarianz, denn der <:-Operator geht in beiden Fällen in die gleiche Richtung.

Die Realität in Java ist allerdings komplizierter. Das liegt daran, dass eine `List<Dog>` eben im Allgemeinen nicht eine `List<Animal>` ist. Denn an eine `List<Animal>` kann man schließlich eine `Cat` anhängen, aber nicht an eine `List<Dog>`. Damit wäre das *Liskovsche Substitutionsprinzip* verletzt, das fordert, dass alle Operationen, die ein Supertyp t erlaubt, auch auf einem Subtyp t' <: t erlaubt sein müssen. Folglich gibt es in Java *Wildcards:*

```
List<Dog> goodDogs = new List<Dog>();
List<? extends Animal> goodAnimals = goodDogs;
```

Die Liste `goodAnimals` lässt sich anschließend fast wie eine `List<Animal>` benutzen, aber nur fast: Der Aufruf von Operationen, die die Liste modifizieren, wird vom Compiler untersagt.

Wenn man hingegen Operationen durchführen möchte, die die Liste modifizieren, aber nicht betrachten, so muss man `super` verwenden:

```
List<Animal> goodAnimals = new List<Animal>();
List<? super Dog> goodDogs = goodAnimals;
```

Auch dieser Code ist zulässig. Der Compiler verbietet Lese-Operationen, zum Beispiel `get`, erlaubt aber Schreib-Operationen, zum Beispiel `remove`.

Dieser Fall erscheint oberflächlich als deutlich seltener. Aber es gibt einen gewichtigen Grund, diesen Mechanismus trotzdem zu haben: `Comparator<T>`. Dabei handelt es sich um eine Klasse, die zwei Werte vom Typ `T` vergleichen kann. Praktischerweise ist es nun so, dass, wenn eine Sortierprozedur einen `Comparator<Dog>` erwartet, auch ein `Comparator<Animal>` herhalten kann. Denn eine Funktion, die Tiere vergleichen kann, kann auch Hunde vergleichen. Formal nennt man das *Kontravarianz*, denn der <:-Operator verläuft gegensätzlich: t <: t' ==> Comparator t' <: Comparator t.

Warum ist das nun wichtig? Java hat sich für sogenannte *Use-Site-Varianz* entschieden, wo man explizit bei jeder Benutzung die Varianz mittels `extends` und `super` signalisieren muss. Das liegt daran, dass man manche Objekte, wie Collections, je nach Anwendungsfall als ko- oder kontravariant auffassen kann. Diese Annahme ist auch im Typsystem fest verankert.

Scala allerdings hat einen besonderen Fokus auf funktionale Programmierung und bevorzugt daher unveränderliche Collections. Objekte werden in aller Regel mit einer fixen Varianz angesprochen. Folglich entschied man sich in Scala für *Declaration-Site-Varianz*, wo ein Typparameter direkt mit entsprechender Varianz deklariert wird. Das sieht dann in etwa so aus:

```
class List[+T]() {
  // list stuff
}
val goodDogs: List[Dog] = List.empty
val goodAnimals: List[Animal] = goodDogs
```

Das `+T` steht für Kovarianz. Kontravarianz markiert man mit `-T`. Möchte man gar keine Varianz, zum Beispiel bei veränderlichen Collections, lässt man das Zeichen weg[2].

2 Scala bietet zusätzlich auch noch Use-Site-Varianz und Wildcards, dies ist aber im Regelfall nur relevant, wenn man Java-Code aufruft, welcher Wildcards benutzt.

Offensichtlich hat sich also Scala für eine andere Form des Subtypings bei Generics entschieden. Auch Kotlin folgt dem Declaration-Site-Modell. Das funktioniert nur, weil Subtyp-Prüfungen vom Compiler und nicht der JVM erledigt werden. Wüsste die JVM von Generics, schlösse das eine ganze Reihe von alternativen Implementierungen aus.

In der Praxis

Damit wäre geklärt, dass *Type Reification,* also das Bereitstellen von Typinformation zur Laufzeit, theoretisch nur schwierig zu handhaben ist. Aber Type Erasure hat auch noch handfeste praktische Vorteile. Der Schlüsselbegriff ist *Parametricity.* Bartosz Milewski schreibt hierzu:

I'm not fond of arguments based on lack of imagination. "There's no way this code may fail!" might be a sign of great confidence or the result of ignorance. The inability to come up with a counterexample doesn't prove a theorem. And yet there is one area of programming where such arguments work, and are quite useful. These are parametricity arguments: free theorems about polymorphic functions. [Mil14]

Es gibt zahlreiche Quellen, die Parametrizität genauer erläutern, darunter auch der einschlägige Artikel von Philip Wadler [Wad89]. Deswegen begnüge ich mich hier damit, eine grobe Intuition zu vermitteln. Vereinfacht: Eine Methode mit Typparameter weiß nichts und darf nichts:

```
public static <T> List<T> sorted(
   List<T> unsorted, Comparator<T> cmp) {
  // return sorted copy
}
```

`sorted` weiß nicht, was `T` ist, und darf mit `T`s aus der Eingabeliste nichts anderes machen, als sie umzusortieren und mittels des Komparators zu vergleichen. Insbesondere darf `sorted` keine `T`s neu erfinden. Es lässt sich also die folgende Aussage beweisen:

- ∀ x.sorted(unsorted,cmp).contains(x) ==> unsorted.contains(x)

Zu Deutsch: Alle Elemente x, die in der sortierten Liste enthalten sind, stammen aus der unsortierten Liste. Diese logische Aussage ist *frei*, das heißt, ihr Beweis ergibt sich allein aus der Methodensignatur und erfordert es nicht, die Implementierung der `sorted`-Methode zu betrachten. Dementsprechend sind auch keine Tests vonnöten.

Nachdem die JVM aber doch eine begrenzte Menge von Typinformationen zur Laufzeit bereithält, muss man sich darauf einigen, gewisse Konstrukte nicht zu benutzen:

- null
- fangbare Exceptions
- isinstanceof
- Casting
- equals, toString, hashCode
- getClass
- globale Nebenwirkungen[3]

Durchsetzen lässt sich das sehr einfach per statischer Analyse oder Linting. Hat man das geschafft, spart man sich eine ganze Reihe von Tests, die andernfalls notwendig würden.

Und was ist mit Arrays?

Aleksey Shipilëv hat sich in einem Artikel [Sch16] mit der Frage beschäftigt, was die schnellere Methode ist: `list.toArray(new T[0])` oder `list.toArray(new T[size])`. Tatsächlich ist die leidige Thematik `toArray` einer der Haupt-Kritikpunkte von Type Erasure. Der scheinbar einfache Anwendungsfall, ein Array generischen Typs zu erzeugen, ist schwierig zu implementieren. In der Tat ist dies in Java ungünstig gelöst. Scala hingegen kann mit einer vernünftigen Lösung aufwarten, den sogenannten `ClassTag`s:

```
def arrayOf[T : ClassTag](t: T): Array[T] = Array(t)
arrayOf("hello")
arrayOf(List("hello", "world"))
```

Die Angabe `T : ClassTag` unterrichtet den Compiler, dass ein Aufrufer nicht nur einen Typ `T` spezifizieren muss, sondern auch noch einen *Tag* vom Typ `ClassTag[T]` bereitstellen muss. Wie man aber im obigen Code-Schnipsel sieht, braucht der Aufrufer dies nicht wirklich zu tun, denn beides wird vom Compiler inferiert. Der `Array`-Konstruktor ist entsprechend implementiert, den `ClassTag` zu analysieren und ein passendes Array zu allozieren.

Kotlin hat einen anderen Ansatz gewählt. Der Compiler verlangt zwei Merkmale von Methoden, die generische Arrays instanziieren möchten: Der Typparameter muss mit dem Schlüsselwort `reified` gekennzeichnet und die Methode als `inline` markiert werden. In etwa entspricht das dem C++-Ansatz, generische Elemente wie Klassen bei ihrer Benutzung zu kopieren. Man erhält also eine Methode für `String`, eine für `Date` usw.

Eine kleine Randbemerkung zu Javas Arrays sei mir noch gestattet. Diese sind seit Tag 1 kovariant implementiert, das heißt, der Java-Compiler lässt es zu, ein `Dog[]` als `Animal[]` zu benutzen. Dazu ist es, im Gegensatz zu anderen Collections wie Listen, nicht erforderlich (und auch nicht möglich), Wildcards zu verwenden. Was passiert aber nun, wenn man in ein Hunde-Array, welches so tut, als sei es ein Tier-Array, eine Katze schreibt? Man erhält eine `ArrayStoreException`. Leider sind Fehler dieser Art nur sehr schwer statisch zu entdecken. TypeScript hatte bis Version 2.6 ein ganz ähnliches Problem.

Fazit

Manchmal sind gängige Meinungen – wie zum Beispiel über Type Erasure – nicht fundiert. Es lohnt sich, mal bei den Akademikern vorbeizuschauen, was die so über Typen sagen. Und was andere JVM-Sprachen so anstellen. Um mit den Worten von Jens Schauder auf Twitter zu schließen:

If you encounter an Object in your code you should worry. Where did it lose its type information?

Links

[Gri17] R. Grigore, Java Generics are Turing Complete, in: ACM POPL 2017, https://arxiv.org/pdf/1605.05274.pdf

[Mil14] B. Milewski, Parametricity: Money for Nothing and Theorems for Free, 22.9.2014, https://bartoszmilewski.com/2014/09/22/parametricity-money-for-nothing-and-theorems-for-free/

[Pie02] B. C. Pierce, Types and Programming Languages, MIT Press, 2002, s. a. https://www.cis.upenn.edu/~bcpierce/tapl/

[Sch16] A. Shipilëv, Arrays of Wisdom of the Ancients, 2016, https://shipilev.net/blog/2016/arrays-wisdom-ancients/

[Wad89] Ph. Wadler, Theorem for free!, in: Int. ACM Sym. on Functional Programming Languages and Computer Architecture, 1989, https://people.mpi-sws.org/~dreyer/tor/papers/wadler.pdf

3 Dazu zählt zum Beispiel das Schreiben in Klassenvariablen.