Christian-Albrechts-Universität zu Kiel

Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion,
Prof. Dr. Michal Hanus

Seminar Implementierung von Programmiersprachen

Betreuer: Diplom-Informatiker Klaus Höppner

Generic<Java>

Sekib Omazic, 19.04.2004

Literatur

G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In Proceedings of *OOPSLA 98*, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, October 1998.
http://www.cis.unisa.edu.au/~pizza/gj/Documents/gj-oopsla.pdf

Gosling James: Java: An Overview - "The original Java Whitepaper" 1995:
http://java.sun.com/people/jag/OriginalJavaWhitepaper.pdf

Bracha Gilad: Adding Generics to the Java Programming Language, JavaOne Conference 2001:
http://java.sun.com/javaone/javaone2001/pdfs/2733.pdf

Paul Mingardi: Preparing for Generics, java.sun.com, November 2002
http://java.sun.com/developer/technicalArticles/releases/generics/

Bracha Gilad: Generics in the Java Programming Language, März 2004
http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

M. Torgersen, C. Hansen, E. Ernst, P. von der Ahe, G. Bracha and N. Gafter: Adding Wildcards to the Java Programming Language, In Proceedings of SAC 2004
http://www.bracha.org/wildcards.pdf

M. Odersky, P. Wadler: "Pizza into Java: Translating theory into practice", Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997,
http://pizzacompiler.sourceforge.net/doc/pizza-language-spec.pdf

1. Einführung

1.1 Was ist Java ?

Java wurde von der Firma Sun entwickelt und erstmals am 23. Mai 1995 als neue, objekt-orientierte, einfache und plattformunabhängige Programmiersprache vorgestellt. Die Sprache ist für alle Computersysteme verfügbar.
Java geht auf die Sprache Oak zurück, die 1991 entwickelt wurde, mit dem Ziel, eine einfache und plattformunabhängige Programmiersprache zu schaffen, mit der nicht nur normale Computer programmiert werden können, sondern auch die in Haushalts- oder Industriegeräten eingebauten Micro-Computer. Allgemein anerkannt wurde Java aber erst seit 1996 in Verbindung mit Web-Browsern und Internet-Anwendungen.

Die wichtigsten Eigenschaften von Java sind:

1. Plattformunabhängig
2. Objekt-orientiert
3. Syntax ähnlich wie bei C und C++
4. umfangreiche Klassenbibliothek

1.2 Erweiterung

Mit Java 1.5 werden einige neue Konzepte in Java eingeführt. Es gibt einen definierten Prozess, wie neue Eigenschaften der Sprache hinzugefügt werden, den Java community process (JPC). Hier können fundierte Verbesserungs- und Erweiterungsvorschläge gemacht werden. Wenn solche von genügend Leuten unterstützt werden, kann ein sogenannter Java specification request (JSR) aufgemacht werden. Die Erweiterung der Programmiersprache Java basiert auf mehreren JSRs. Den größten Einschnitt stellt sicherlich die Erweiterung des Typsystems auf generische Typen dar. Auch die anderen neuen Konstrukte, die verbesserte for-Schleife, Aufzählungstypen, statisches Importieren und Autoboxing, erleichtern die alltägliche Arbeit mit Java.

Die nachträgliche Erweiterung von Java um Generizität ist eine kritische Angelegenheit. Eine der wichtigsten Anforderungen war, möglichst wenig bzw. kein Einfluss auf JVM. Eine JVM ist sehr einfach auf ein PC nachzurüsten, aber in einem Handy eine neue JVM einzubauen, die vielleicht mehr Harware-Leistung und/oder Speicher benötigt würde mit Sicherheit zu einem großem Problem führen. Da Java auch wegen seiner Plattformunabhängigkeit als bevorzugte Sprache im Internet eingesetzt wird, ist dem Sicherheitsmodell besondere Beachtung zu schenken.

In den letzen Jahren gab es eine Vielzahl von Vorschlägen, wie man Java erweitern könnte, so dass die Generizität unterstützt wird. Einige von diesen Vorschlägen sind als Prototypen implementiert worden. Der wichtigste Entwurf war GJ (Generic Java), der im wesentlichen als Konzept für Java 1.5 übernommen wird. GJ basiert auf Pizza, einer Java-Erweiterung, die viele neue Features in Java einführt. In Pizza werden Funktionen höherer Ordnung und algebraische Datentypen eingeführt. Primitive Typen wie z.B. `int` sind als Typparameter erlaubt. Außerdem beeinflusst das Übersetzungsschema von Pizza die Methodennamen, so dass man die alten Java Bibliotheken nicht mit Pizza Code vermischen kann. Generic Java ist

eine vereinfachte Version von Pizza. Hier hat man sich auf Generizität und Kompatibilität konzentriert.
Im Verlauf dieser Ausarbeitung steht Java für das klassische Java und generischens Java (Generic Java oder GJ) für die neue Version mit Generizität.

2. Generizität

2.1 Polymorphie und Generizität

Oftmals gibt es Funktionen, deren Algorithmus nicht von einem speziellen Datentyp abhängig ist, sondern mit jedem beliebigen Datentyp arbeiten kann. So ist zum Beispiel das Ermitteln der Länge einer Liste unabhängig davon, mit welchem konkreten Datentyp die Liste aufgebaut ist. Eine Abhilfe bieten hierfür polymorphe Datentypen. Der polymorphe Datentyp ist dabei ein Stellvertreter für einen beliebigen, aber noch unbestimmten, konkreten Datentyp. Er stellt somit eine Obermenge aller möglichen Typausprägungen dar. Die Polymorphie definieren wir als eine Eigenschaft von Variablen, Objekte unterschiedlichen Typs speichern zu können. Variablen können Objekte zugewiesen werden, die vom gleichen Typen wie diese Variable oder von einem abgeleiteten Typen sind.

```
List list = new LinkedList();
```

`List` ist ein statischer, in der Variablendeklaration festgelegter Typ und `LinkedList` dagegen ist ein dynamischer, beim Erzeugen des Objekts festgelegter Typ. Der statische Datentyp ist fest und kann für eigene Anwendungen nicht leicht angepasst werden. Die Standardlösungen sind:

1. Nutzung des niedrigsten gemeinsamen Typen der Klassenhierarchie, z.B. `Object`
2. Spezialisierte Implementierungen mit unterschiedlichen Typen. (Methodenüberladen)

Eine elegantere Lösung bietet die Generizität an.

Generizität (oder "parametrisierte Typen") ist ein Mechanismus, der es uns ermöglicht die Typen von Objekten zu spezifizieren, mit denen eine Klasse arbeiten kann. Diese Spezifizierung geschieht mit Hilfe von Parametern, die bei der Deklaration übergeben und während Kompilierung ausgewertet werden.

2.2 Generische Klassen

Grundsätzlich ist es auch in Java möglich Datenstrukturen zu definieren, die Daten beliebigen Typs verwalten können. Das geht bisher in Java, allerdings mit einem kleinen Nachteil. Man kann in traditionellem Java eine Klasse schreiben, in der man beliebige Objekte speichern kann. Dafür braucht man ein Feld, in dem Objekte jeden Typs gespeichert werden können. Dieses Feld muss daher den Typ `Object` erhalten, da alle Klassen Untertypen von `Object` sind:

```
class OldBox {
      Object contents;
      OldBox(Object contents){this.contents=contents;}
}
```

Wenn wir auf das Feld `contents` zugreifen, so haben wir über das darin gespeicherte Objekte keine spezifische Information mehr. Um das Objekt weiter sinnvoll nutzen zu können, ist eine dynamische Typzusicherung durchzuführen:

```
OldBox b = new OldBox("Moin");
String s = (String)b.contents;          // downcast
```

Die dynamische Typzusicherung kann aber zu einem Laufzeitfehler führen. So übersetzt das folgende Programm fehlerfrei, ergibt aber einen Laufzeitfehler:

```
OldBox b = new OldBox(new Integer(99));
String s = (String)b.contents;          // ClassCastException!
```

Es ist wünschenswert, Klassen zu implementieren, die Objekte beliebigen Typs speichern und genauso allgemein benutzbar sind wie die Klasse `OldBox` oben, aber trotzdem die statische Typsicherheit garantieren, indem sie nicht mit dem allgemeinen Typ `Object` arbeiten. Genau dieses leisten generische Klassen. Statt `Object` verwendet man eine Typvariable, die dann mit jedem beliebigen Typ instantiiert werden kann. Dem Klassenheader wird die deklaration der Typvariablen in spitzen Klammern hinzugefügt, so dass diese Typvariable im Klassenrumpf verwendet werden kann. Hierzu ersetzen wir in der obigen Klasse jedes Auftreten des Typs `Object` durch die Typvariable. Sie steht für einen beliebigen Typen. So erhält man aus der obigen Klasse `OldBox` folgende generische Klasse `Box`:

```
class Box<einTyp> {
      einTyp contents;
      Box(einTyp contents){this.contents=contents;}
}
```

Die Typvariable *`einTyp`* ist als allquantifiziert zu verstehen. Die Klasse `Box` kann mit jedem beliebigen Typen als Argument verwendet werden. Sobald man ein konkretes Objekt der Klasse `Box` erzeugen will, muss man entscheiden, für welchen Inhalt eine Box gebraucht wird. Dafür muss lediglich der entsprechende Typ in spitzen Klammern in der Klassendefinition einsetzen werden. Man verwendet dann z.B. den Typ `Box<String>`, um Strings in der Box zu speichern, oder `Box<Integer>`, um Integerobjekte darin zu speichern:

```
Box<String> b1 = new Box<String>("Alter Schwede");
Box<Integer> b2 = new Box<Integer>(new Integer(99));
```

Die Variablen `b1` und `b2` sind jetzt nicht einfach vom Typ `Box`, sondern vom Typ `Box<String>` respektive `Box<Integer>`. Zu beachten ist, dass primitive Typen, wie z.B. `int` oder `long` als Typparameter nicht erlaubt sind.
Da wir mit generischen Typen keine Typzusicherungen mehr vorzunehmen brauchen, bekommen wir auch keine dynamischen Typfehler mehr. Der Laufzeitfehler, wie wir ihn ohne die generische Box hatten, wird jetzt bereits zur Übersetzungszeit entdeckt.

```
Box<String> b = new Box<String>(new Integer(99));  // Fehler !
```

Auch Methoden lassen sich generisch in Java definieren. Hierzu ist vor der Methodensignatur in spitzen Klammern eine Liste der für die statische Methode benutzten Typvariablen anzugeben.

Eine sehr einfache generische Methode ist die `genericRandom`-Methode, die zwei beliebige Objekte erhält und zufällig einen der beiden Parameter unverändert wieder zurückgibt. Diese Methode hat für alle Typen den gleichen Code und kann wie folgt generisch implementiert werden:

```
public <einTyp> einTyp genericRandom(einTyp m, einTyp n)
{
            return Math.random() > 0.5 ? m : n;
}
```

Wie man sieht, wird der Typparameter *einTyp* sowohl als Typ für die beiden Parameter als auch als Rückgabetyp verwendet.

2.3 Beschränkte Typvariablen (Bounds)

Bisher standen in allen Beispielen die Typvariablen einer generischen Klasse für jeden beliebigen Objekttypen. Manchmal ist es sinnvoll die Typargumente einer generischen Klasse einzuschränken, so dass nur bestimmte Typen verwendet werden können, die eine Schnittstelle implementieren oder von einer Klasse ableiten.
Nehmen wir als Beispiel unser Methode `genericRandom`. Sollte man nur Klassen verwenden, die `CharSequence` implementieren (also Zeichenfolgen wie `String` und `StringBuffer`), so wird das mit in die Definition mit hineingeschrieben:

```
public <T extends CharSequence> T genericRandom (T m, T n)
{
      return Math.random() > 0.5 ? m : n;
}
```

Somit kann die Methode `genericRandom` nur noch mit Parametern vom Typ `CharSequence` aufgerufen werden (z.B. `genericRandom("rot", "blau")`).

Durch die Einschränkung von Typvariablen wird eine minimale Schnittstele vorgegeben. Beim Aufrufen von Methoden dieser Schnittstelle ist also kein Cast notwendig und deshalb kein Laufzeitfehler mehr möglich.

Nehmen wir an, wir wollten ein typsicheres `max()` implementieren. Es soll den größeren der beiden Werte zurückgeben. Um das Maximum zu bestimmen, brauchen wir eine Vergleichsmethode. Aber nicht alle Objekte lassen sich vergleichen. Objekte die die Schnittstelle `Comparable` erfüllen stellen jedoch die Vergleichsfunktion `compareTo` zur Verfügung. Somit können nur Objekte die diese Schnitstelle erfüllen sinnvoll verglichen werden

```
public <einTyp extends Comparable> einTyp max(einTyp m, einTyp n)
{
      return m.compareTo(n) > 0 ? m : n;
}
```

Die Nutzung ist einfach:

```
System.out.println(o.max("Gehen", "Laufen")); // lefert „Laufen“
System.out.println(o.max(new Integer(12), new Integer(100)));
```

Bounds erlauben eine vollständigere Typüberprüfung zur Compilezeit und reduzieren somit die möglichen Fehlerquellen zur Laufzeit.
Zugriffe auf Methoden einer uneingeschränkten Typvariablen können zur Compilezeit nicht überprüft werden, da keine Informationen über die Typvariable vorhanden sind. Das gleiche gilt für Objekte, deren Typ nur durch eine Typvariable gegeben ist. Durch Einschränkungen wird die minimale Schnittstelle einer Typvariablen festgelegt. Somit ist für Aufrufe von Methoden aus dieser Schnittstelle kein Cast zur Laufzeit nötig.

Typvariablen dürfen mehrere Einschränkungen haben. Die Syntax ist:

$\texttt{TypeVariable implements Bound}_1 \texttt{ \& Bound}_2 \texttt{ \& ... \& Bound}_n$

2.4 Einschränkungen und Probleme

Es stellt sich die Frage, wann ein Typparameter zu einem anderen Typparameter konform ist. Man könnte denken, dass folgender Code gültig wäre:

```
Vector<Number> v = new Vector<Integer>();
```

Obwohl `Integer` eine Unterklasse von `Number` ist, ist solche Zuweisung nicht erlaubt. Das geht nur, wenn die Typparameter gleich sind. Auf der Ebene generischen Klassen gelten weiterhin die normalen Gesetze:

```
Vector<String> s = new Stack<String>();
```

`Stack` ist eine Unterklasse von `Vector` und Typparameter sind identisch.

Bei den Java Arrays haben wir ein anderes Verhalten. Da ist die entsprechende Zuweisung erlaubt. Diese kann jedoch zu einem Laufzeitfehler führen:

```
String [] x = {"Moin"};
Object [] b = x;
b[0]=new Integer(99);  // ArrayStoreException
```

Es ist noch zu beachten, dass parametrisierte Typen nicht direkt oder indirekt von `java.lang.Throwable` ableiten dürfen, da der Exception-Mechanismus der Virtuellen Maschine keine Generizität unterstützt.

3. Typ-Inferenz

Als Typ-Inferenz bezeichnet man das Ableiten eines Typs für ein Objekt in einem Ausdruck aus vorgegebenen Informationen über die Typen anderer Objekte. In GJ wird ein Typ-Inferenz Algorithmus implementiert, der es uns erlaubt den Typparameter beim Aufruf einer polymorphen Methode auszulassen. Das ist möglich, da die Typparameter bei der Übersetzung sowieso gelöscht werden. Die Typparameter werden für den Aufruf einer generischen Methode abgeleitet, indem man den kleinsten gemeinsamen Typ nimmt, bei dem der Methodenaufruf gültig ist. Man nennt diese Typbestimmung lokal, da sie unabhängig von dem weiteren Kontext, insbesondere einem erwarteten Rückgabetyp ist.

```
public <einTyp> LinkedList<einTyp> doublet(einTyp x, einTyp y) {
```

```
        …
}

List<Number> zs = f.doublet(new Integer(1), new Float(1.0));
```

Hier ist `Number` der kleinste gemeinsame Nenner für `Integer` und `Float`. Falls keiner eindeutiger kleinster Typ existiert, versagt die Ableitung.

```
LinkedList<Object> err = f.doublet("abc", new Integer(1));  //FEHLER
```

3.1 Der null-Typ

Wenn der Compiler keinen Typ für eine parametrisierte Methode bestimmen kann, weil sie z.B. keine Argumente hat, dann wird ein parametrisierter Typ durch den besonderen Typ `"*"` ersetzt. Er wird im Folgenden als *`null-Typ`* bezeichnet, da die Konstante null auch diesen Typ hat. Dieser Typ darf nicht im Javaquellcode vorkommen. Seine Eigenschaften sind sehr einfach: er ist ein Subtyp von allen Referenz-Typen. Darüber hinaus ist jeder Typ, der den *`null-Typ`* enthält, ein Subtyp jedes Typen, der entsteht, wenn man den *`null-Typ`* durch einen anderen Typen ersetzt.
z.B.:
`LinkedList<*>` ist ein Subtyp von `LinkedList<String>` und `Pair<Byte,*>` ist ein Subtyp von `Pair<Byte, Byte>`.

Betrachten wir nun folgende Methode:

```
1: public <einTyp> LinkedList<einTyp> empty() {
2:    return new LinkedList<einTyp>();
3: }
...
4: LinkedList<String> x = f.empty();
```

Hier gibt die Methode empty() einen parametrisierten Typ zurück. Da sie keine Parameter hat, wird für *`einTyp`* der *`null-Typ`* angenommen. Da `LinkedList<*>` ein Subtyp von `LinkedList<String>` ist, ist die Zuweisungen in Zeile 4 korrekt. Der aktuelle Argument-Typ ist immer ein Sub-Typ des formalen Parameters an der entsprechenden Position.

Es gibt jedoch eine Möglichkeit das Typsystem zu umgehen, weshalb eine weitere Einschränkung gemacht werden muss: der Methodenaufruf ist nur dann gültig, wenn der *`null-Typ`* nicht mehr als einmal im Rückgabetyp auftaucht. Dies ist eine so genannte Linearitätsbeschränkung. Auch hierzu ein kleines Beispiel:

```
1: public static <einTyp> Pair<einTyp, einTyp> duplicate(einTyp x) {
2:    return new Pair<einTyp, einTyp>(x,x);
3: }
…
4: Pair<Box<String>,Box<Byte>> p = Pair.duplicate(null); // FEHLER

5: Pair<Box<String>,Box<String>> p = Pair.duplicate((String)null));
```

Der Aufruf in der Zeile 4 ist nicht erlaubt, da die Methode duplicate einen Typ zurück gibt, der vom Compiler zu `Pair<Box<*>, Box<*>>` auswertet wird. Hier könnte ein Byte zugewiesen werden, dass dann als String ausgelesen wird. Beide Felder der Klasse Pair zeigen nämlich auf das gleiche Objekt - schon ist das Typsystem nicht mehr intakt.
Dagegen ist der zweite Aufruf erlaubt, da hier der Typ `Pair<Box<String>, Box<String>>` entspricht und keine Verletzung des Typsystems auftritt. Aber ohne den Cast in String Typ wäre * der kleinste gemeinsame Typ und der Aufruf wäre ungültig.

Kovarianz kann leicht zu einer Verletzung des Typsystems führen. Die Argumentation beginnt damit, dass kein Typ `T<...*...>` explizit deklariert werden kann. Alles, was mit einem Wert von diesem Typ gemacht werden kann, ist eine Zuweisung an eine Variable bzw. einen Methodenparameter von einem anderen Typ. Es gibt dann drei Möglichkeiten:

1. der Variablentyp ist ein unparametrisierter Supertyp von dem Rawtype `T`. Diese Zuweisung lässt das Typsystem intakt.
2. der Variablentyp ist `T´<....U...>`, d.h. an der Position von `*` steht ein konkreter Referenztyp. `T´` ist ein Supertyp von `T`. Da nur die Konstante null den *`null-Typ`* hat, und diese auch an jeden Referenztypen zugewiesen werden kann, ist die Zuweisung ebenfalls in Ordnung.
3. die Variable ist ein Parameter p, dessen Typ eine Typvariable ist. Code der p im Methodenrumpf benutzt, ist unabhängig vom aktuellen Typ. Durch die Linearitätsbeschränkung wird der Methodenrückgabetyp maximal einmal den Typparameter enthalten, so dass dieser wieder in der Form `T<....*...>` ist.

4. Neue Features von Java 1.5

4.1 Autoboxing

Konvertierung primitiven Typen wie z.B. `int` oder `boolean` in das objektorientierte Pendant wie `Integer` oder `Boolean` erfordert zusätzlichen Code, besonders für die Methoden aus Collection API. Das Autoboxing und Auto-Unboxing von primitiven Typen erzeugt den Code, der kürzer und einfacher zu lesen ist:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 99);
int total = list.get(0);
```

Die notwendige Transformation zwischen `int` und `Integer` erledigt der Compiler.

4.2 Die neue `for`-Schleife

`for`-Schleifen laufen oft Felder oder Datenstrukturen ab. Nehmen wir als Beispiel die Argumente der Kommandozeile, die sich im Feld args[] befinden. Sie sollen auf dem Bildschirm ausgegeben werden.

```
for (int i = 0; i < args.length; i++)
    System.out.println(args[i]);
```

Auffällig ist, dass *i* eigentlich gar keine große Rolle spielt, sondern lediglich als Index seine Berechtigung hat; nur damit lässt sich das Element an einer bestimmten Stelle im Feld ansprechen. Da dieses Durchlaufen von Feldern häufig ist, haben die Entwickler bei Sun seit Java 1.5 eine Abkürzung für solche Iterationen in die Sprache eingeführt:

```
for (Typ Bezeicher : Feld)
  ...
```

Diese erweiterte Form der for-Schleife löst sich vom Index und erfragt jedes Element des Feldes. Das lässt sich als Menge vorstellen, denn der Doppelpunkt liest sich als „in“. Formulieren wir unser Beispiel mit dem erweiterten `for`.

```
for (String arg : args)
    System.out.println(arg);
```

Zu lesen ist die `for`-Zeile demnach mit „Für jedes Element arg vom Typ String in args tue...". Einen Variablennamen wie *i* für den Schleifenindex ist nicht mehr nötig, denn der Index ist nicht sichtbar. Intern setzt der Compiler diese erweiterte `for`-Schleife ganz klassisch um, sodass der Bytecode unter beiden Varianten gleich ist.

4.3 Wildcards

Wildcards sind eine Erweiterung der GJ Spezifikation. Es ist manchmal nicht sinnvoll den Typparameter genau zu spezifizieren. Stattdessen benötigen wir den Typparameter uneingeschränkt, um dem Compiler signalisieren zu können, dass der genaue Typ unwichtig ist. Wichtig ist, dass es einen Typparameter gibt. Der Name *wildcards* kommt von der Tatsache, dass wir entweder den Typparameter (z.B. `einTyp`) oder einen anderen „richtigen“ Typ mit einem uneingeschränkten Typen **?** ersetzen.

Überall wo wir den Typparameter benutzen können (Feld, Methode, Variable), können wir auch die wildcards einsetzen. Hier ist ein Beispiel mit der neuen Syntax für `for`-Schleife:

```
// Version 1:
public <einTyp extends Animal> void printAnimals(Collection <einTyp>
animals) {
    for (Animal nextAnimal: animals)
        System.out.println(nextAnimal);
}
```

In der ersten Version wird der Parameter *einTyp* deklariert aber nicht benutzt. Wir können ihn also mit einem *widlcard* ersetzen (Version 2).

```
// Version 2:
public void printAnimals(Collection <? extends Animal> animals) {
    for (Animal nextAnimal: animals)
        System.out.println(nextAnimal);
}
```

Wichtig ist, dass der Typparameter in der Methode nicht referenziert wird. Das ? sollte uns sagen, dass beliebige Typen in die Methodendeklaration passen (ähnlich wie ? bei regulären Ausdrucken). Wenn wir aber ? in der Methode benutzen, dann binden wir es implizit – und das wäre gegen die Regel.

Wildcards können in ähnlicher Art auch für Felddeklarationen verwendet werden.

```
private Collection<? extends Animal> anm = new ArrayList<einTyp>();
```

5. Übersetzung

In Java war die Abwärtskompatibilität schon immer wichtig. Änderungen der Sprache oder Kernbibliotheken mussten abwärts kompatibel sein. Deswegen war die Implementierung von Generics schwierig. Das Format von class-Dateien konnte nicht großartig geändert werden. Der Code geschrieben mit "alten" Klassen musste auch unter GJ funktionieren. Um das zu realisieren, benutzt GJ eine Technik namens Erasure. Der Compiler arbeitet in zwei Schritten: Zuerst wird eine Typüberprüfung gemacht. Der Compiler benutzt alle Informationen, die er hat (Typparameter inklusive). Dann wird der Code transformiert, indem man alle Typparameter löscht und einige Casts einführt. Falls eine Klasse in ihrer Definition keine Typparameter nutzt, so ändert sich nichts. Typparameter werden weggelassen und durch einen Typ ersetzt, der Compiler problemlos einführen kann:
z.B. im Fall von `<T>` wird `T` zu `Object` und im Fall von `<T extends Rentable>` wird `T` auf Rentable abgebildet.

Betrachten wir folgendes Beispiel:

```
public class ShoppingCart<T extends Rentable> {
    private Vector<T> _contents = new Vector<T>();

    public void add(T rentable) {
         …
    }

    public int getTotalPurchasePrice() {
         …
         T itemInCart = iterator.next();
         …
    }

}
```

Nach dem Erasure sind alle Typparameter gelöscht und Casts dort eingeführt, wo es nötig ist. Unser Code sieht jetzt so aus:

```
public class ShoppingCart {
    private Vector _contents = new Vector(); // nur ein Vector

    public void add(Rentable rentable) {     // T ist Rentable
         …
    }

    public int getTotalPurchasePrice() {
         …
         Rentable itemInCart = (Rentable) iterator.next(); // cast
         …
    }
}
```

Der Erasure-Prozess liefert eigentlich den Code, den wir mit dem "alten" Compiler übersetzen können. Erasure ist ein interner Prozess. Den "gelöschten" Code sieht man nur mit Hilfe eines Decompilers.

Eine Nebenwirkung von Erasure ist, dass der Code manchmal nicht übersetzt werden kann:

```
class SpecialOffer<T extends Rentable, W extends Rentable> {
    public void add(T newRentable) {
        //...
    }
    public void add(W newRentable) {
        //...
    }
}
```

Beide Variablen T und W werden zum Rentable abgebildet, was dazuführen wurde, dass beide `add` Methoden identisch wären.

Das Übersetzungsschema von Generic Java nennt man auch homogene Übersetzung. Da die Homogene Übersetzung Informationen über den Typ löscht, kann ein Sicherheitsloch entstehen:

```
class SecureChannel extends Channel {
      public String read();
}
class C {
      public LinkedList<SecureChannel> cs;
...
}
```

Der Typ von `LinkedList<SecureChannel>` wird beim Kompilieren durch den Rawtype `LinkedList` ersetzt. Jetzt wäre es möglich einen unsicheren Kanal in die Liste einzufügen - z.B. durch Code der mit einem älteren Compiler kompiliert wird, oder durch Programmierung direkt im Bytecode. Die Typverletzung würde nicht bemerkt werden. Die Lösung ist eine spezielle Klasse die von der parametrisierten Klasse abgeleitet wird:

```
class SecureChannelList extends LinkedList<SecureChannel> {
       SecureChannelList(){ super(); }
}

class C {
       SecureChannelList cs;
       ...
}
```

5.1 Brückenmethoden

In GJ wird nicht nur die Erasure-Technik eingesetzt, sondern auch eine andere Art von Codetransoformation. Dieser Prozess, genant Bridging (Überbrückung) besteht aus Einfügung zusätzlichen Methoden (Brückenmethoden) in Objekte. Wie für Erasure ist auch für Bridging die Abwärtskompatibilität die Hauptmotivation.

Betrachten wir folgendes Beispiel:

```
interface Comparable<T> {
      public int compareTo(T that);
}

class Byte implements Comparable<Byte> {
      …
      public int compareTo(Byte that) {
            return value - that.value;
      }
}
```

Nach dem Übersetzen, würde unsere Klasse so aussehen:

```
interface Comparable {
      public int compareTo(Object that);
}

class Byte implements Comparable {
      …
      public int compareTo(Byte that) {
            return value - that.value;
      }

      // Brückenmethode
      public int compareTo(Object that) {
            return this.compareTo((Byte) that);
      }
}
```

Die zweite `compareTo` Methode ist eine Brückenmethode, die die Argumente in den bisherigen Typ castet und dann die speziellere Methode aufruft. Eine Brückenmethode ist nicht anderes, als eine Überladung der Methode mit Änderung der Argumenttypen. Die Brückenmethode hat als Argumenttypen `Object`. Dies ist notwendig, weil eine Methode, die Typparameter enthält, auch überschrieben werden darf. Konkret darf die `compareTo` Methode aus der Klasse `ByteComparator` überschrieben werden. Nun gibt die „Java Language Specifikation“ aber vor, dass eine Methode nur dann überschrieben wird, wenn die Signatur exakt gleich bleibt. Aus diesem Grund muss die Brückenmethode eingefügt werden.
Ein Problem entsteht bei Methoden einer Klasse, die gleiche Parameterliste und verschiedenen Rückgabetyp haben:

```
class Interval implements Iterator<Integer> {
      …
      public Integer next() { return new Integer(i++); }
}
```

Die Übersetzung ist, wie erwartet mit einer Brückenmethode:

```
class Interval implements Iterator {
      …
      public Integer next() { return new Integer(i++); }
      public Object next() { return next(); }   // Brückanmethode

}
```

Dies ist aber kein gültiger Javaquellcode, da die beiden `next` Methoden nicht unterschieden werden können – sie haben identische Argumente. Diese zwei Methoden lassen sich aber in der JVM an ihrer vollen Signatur (Argumenttypen und Ergebnistyp) unterscheiden. In solchen Fällen kompiliert GJ die Brückenmethoden direkt in JVM-Byte-Code.

5.2 Raw Types

Einen generischen Typen ohne seine Parameter nennt man Raw-Type. So ist z.B. `LinkedList` der Raw-Type von `LinkedList<einTyp>`. Methoden, die für Objekte eines Raw-Types aufgerufen werden, entsprechen Methoden bei denen die Typinformation gelöscht wurde.

Eine Zuweisung von einem parametrisierten Typ an den entsprechenden Raw-Type ist immer möglich. Das bedeutet also auch, dass z.B. `Collection<einTyp>` an eine Methode übergeben werden kann, die eigentlich nur `Collection` erwartet. Anders herum geht die Zuweisung auch, aber hier wird eine Unchecked-Warnung zur Compilezeit ausgegeben. Auch einige Methodenaufrufe können auf einem Raw-Type eine Unchecked-Warning auslösen, da man beim kompilieren nicht feststellen kann, ob der richtige Typ benutzt wurde:

```
LinkedList<String> xs = new LinkedList<String>();
LinkedList ys = xs;
ys.add(y);              // unchecked warning
```

Eine unchecked-Warning wird bei einem Raw Type in zwei Fällen ausgelöst:

1. Methodenaufruf, bei dem durch die Typauslöschung die Argumenttypen verändert wurden
2. Zuweisung an ein Feld, wenn die Typauslöschung den Feldtyp geändert hat

5.3 Arrays

Wie schon gesagt, wird die Typinformation vom Compiler gelöscht und steht zur Laufzeit nicht mehr zur Verfügung. Daher ist es nicht möglich, von einem als Parameter übergebenen Typen ein Objekt zu erzeugen. Lautet der Parameter z.B. `T`, so ist ein Code mit `new T();` nicht erlaubt, da daraus nach der Übersetzung "`new Object()`" werden würde.

Es ist jedoch möglich ein Array von dem Typparameter zu erzeugen: `new T[21]` ist erlaubt, und führt beim kompilieren nur zu einer "unchecked warnung". Dies muss auch so sein, damit z.B. die Collection Klassen intern mit Arrays von parametrisierten Typen arbeiten können. Es ist empfehlenswert ein Array dieser Art immer außerhalb der parametrisierten Klasse zu erzeugen (wo der Typ bekannt ist) und dann als Parameter an die Klasse zu übergeben.

Beispiel:

```
1: public <einTyp> einTyp[] singleton (einTyp x) {
2:    return new einTyp[]{ x }; // unchecked warning
3: }
…
4: String[] a = singleton("zero"); // run-time exception
```

Übersetzt sieht die Zeile 4 so aus:

```
String[] a = (String[])singleton("zero"); // run-time exception
```

Das Problem ist der Cast von Object[] in String[].

5.4 Casts und instanceof-Tests

Da zur Laufzeit keine Informationen über den Typparameter zur Verfügung stehen, werden nicht alle Casts und instanceof-Tests zugelassen. Nur wenn der Cast durch eine Kombination von Typinformationen zur Kompile- und Laufzeit geprüft werden kann, ist er zulässig.

```
public static <T> LinkedList<T> down (Collection<T> xs) {
      if (xs instanceof LinkedList<T>) return (LinkedList<T>)xs;
      else throw new ConvertException();
}
```

Zur Laufzeit wird in der `down`-Methode geprüft, ob xs vom Rawtype `LinkedList` ist. Der Compiler wird darüber hinaus sicherstellen, dass der Typparameter stimmt. Es kann nur eine `Collection` mit `T` als Inhalt übergeben werden, so dass der Cast in `LinkedList` korrekt funktioniert und zum Rückgabetyp passt.

Typparameter dürfen in Cast und instanceof Operationen nicht benutzt werden, wenn es nicht möglich ist, die Parameter zu überprüfen. Z.B. Folgendes ist ungültig:

```
class BadConvert {
      …
      public static LinkedList<String> down (Object o) {
            if (o instanceof LinkedList<String>) // compile-time error
                  return (LinkedList<String>)o;  // compile-time error
            else throw new ConvertException();
      }
}
```

Es gibt zwei Möglichkeiten, das umzugehen. Man kann den Typ spezialisieren, indem man eine neue Klasse erstellt, die `LinkedList<String>` ableitet:

```
class LinkedListString extends LinkedList<String> {...}
```

oder man schreibt eine Wrapper-Klasse, die ein Feld `LinkedList<String>` enthält:

```
class LinkedListWrapper {
      LinkedList<String> contents;
}
```

In beiden Fällen hat die resultierende Klasse keine Typparameter und kann immer in Cast-Operationen benutzt werden.

6. Retrofitting

Als Retrofitting bezeichnet man das Anpassen vorhandener Klassen an die parametrisierte Form. Um die unabhängige Kompilierung zu ermöglichen, müssen die Typinformation zur Kompile-Zeit gespeichert werden. Zum Glück unterstützt das JVM-class-file Format die Speicherung von extra Informationen. Diese Daten werden von alten Compilern und von der JVM ignoriert. Ein Compiler der über Generizität verfügt, kann diese Information aber auswerten um die korrekte Anwendung von Typen zu überprüfen. UncheckedWarnings werden dann nicht mehr ausgelöst.

GJ Code der eine typparametrisierte Bibliothek benutzt, kann auch mit einer unparametrisierten Fassung (also einer älteren Fassung der Bibliothek) arbeiten. Konkret bedeutet das, dass ein Programm unter Java 1.5 mit `LinkedList<String>` kompiliert werden kann und dieses Programm auch unter Java 1.4 mit einer alten Bibliothek wie erwartet funktionieren wird. Um vorhanden Bibliotheken die Typsignaturen hinzuzufügen hat der GJ-Compiler einen speziellen Retrofitting-Modus. Dabei wird die alte Klasse angegeben, die ohne Generizität arbeitet, und ein Sourcecodefile, dass nur die Klassendeklaration, eventuelle Attribute und die Methodensignaturen enthält.

Beispiel:

```
class LinkedList<einTyp> implements Collection<einTyp> {
      public LinkedList();
      public void add(einTyp elt);
      public Iterator<einTyp> iterator();
}
```

Dieses Quellcodefile (kein gültiger Javacode), wird mit der zu ändernden Klasse und zusätzlichen Parametern in den Compiler gesteckt. Dieser erzeugt eine neue Version der Klasse, die über die nötige Typsignatur verfügt. Alle betroffenen Bibliotheksfunktionen wurden auf diese Weise "generizitätsfähig" gemacht.

7. Fazit

In dieser Ausarbeitung wurde eine Erweiterung der Sprache Java vorgestellt: Generic Java. GJ versucht Java-Programmierung einfacher und sicherer zu machen.

GJ ist aus dem Pizza Projekt entstanden, dass Java um einige Konzepte aus funktionalen Programmiersprachen erweitert hat. Unter anderem wurden in Pizza Funktionen unterstützt die als Parameter übergeben werden können, Klassen und Methoden konnten mit Typen parametrisiert werden und die Sprache wurde um algebraische Datenstrukturen erweitert. GJ hat von diesen Konzepten nur die Typgenerizität übernommen und abgewandelt. Pizza wird inzwischen nicht mehr weiterentwickelt.

Einer der wichtigsten Vorteile von Generic Java ist die Tatsache, dass sie den Bedarf nach Laufzeit-Überprüfung verringert. Die Typüberprüfung in Java war sehr beschränkt. Diese Beschränkung wird offensichtlich, wenn die Laufzeit Fehler wegen falschen Casting ausgelöst werden. GJ Compiler verschiebt die Typüberprüfung von der Laufzeit zur Compile-Zeit. Der Code wird dadurch sicherer.
Ein anderer Vorteil für Entwickler ist die Tatsache, dass man keine zusätzlichen Klassen schreiben muss (so wie Templates bei C++), um die Generizität zu unterstützen. Es gibt nur Unterschiede in der Klassen- oder Schnittstellendeklaration.
Die Implementierung von GJ führt die erweiterte Java-Version auf die ursprüngliche Version zurücktransformiert. Aus diesem Grund lässt sich der neue Code mit dem alten problemlos vermischen.
Auch die neue Syntax für `for`-Schleife mach den Source Code übersichtlicher. Es wäre vielleicht etwas „lesbarer“ das Wort „in“ anstatt „:“ in dieser neuen Syntax zu verwenden.