

Objekt-Orientierte Programmiersprachen

Martin Gasbichler, Holger Gast

9. / 14. Februar 2006

Generics: Polymorphie für OOPS

- Generics sind ein relativ neues Sprachfeature in Mainstream OO-Sprachen
 - Java 2 (JDK 1.5)
 - C#
- Sie bauen jedoch auf früheren Konzepten auf
 - Parametrische Polymorphie in ML
 - Bounded Polymorphism bei Subtyping
- Grund für späte Aufnahme in Sprachen: Interaktion zwischen
 - Parametrischer Polymorphie (Generics) und
 - Subklassen-Polymorphie

Ziel hier: Ein Überblick zu Java Generics

- Anwendungen von Generics
- Rückführung auf ML Polymorphie
- Interaktion: Parameterische & Subtyp-Polymorphie
- Implementierung, Einschränkungen, Fallstricke von Java Generics

Dokumentation zu Java Generics

- Gilad Bracha: Generics in the Java Programming Language, 2004
<http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>
- Grundlage:
 - Bracha, Odersky, Stoutamire, Wadler: “Making the Future Safe for the Past: Adding Genericity to the Java Programming Language”, OOPSLA '98
 - Agesen, Freund, Mitchell: “Adding Type Parameterization to the Java Language”, OOPSLA '97
- Die Spezifikation: “Adding Generic Types to the Java Programming Language” (JSR 14)
- Mehr: <http://java.sun.com/j2se/1.5.0/docs/guide/language>

Motivation: Ein Stapel für beliebige Objekte

```
class Stack {
    private class Node {
        private Object value;
        private Node next;
    }
    private Node top;
    public boolean empty() { return top == null; }
    public void push(Object x) {
        Node n = new Node();
        n.next = top;
        n.value = x;
        top = n;
    }
    public Object pop() {
        Object r = top.value;
        top = top.next;
        return r;
    }
}
```

Stack speichert beliebige Objekte in verketteter Liste

Anwendung Stapel

```
Stack s = new Stack();
for (int i=0; i<17; ++i)
    s.push(new Integer(i));
int sum = 0;
while (!s.empty()) {
    Integer i = (Integer)s.pop();
    sum += i.intValue();
}
System.out.println("Summe ist "+sum);
```

- Typische Situation: Wir kennen den Typ der Elemente von s
- ⇒ Invariante: “s enthält nur Objekte, deren **dynamischer Typ** Subtyp von Integer ist”
- Trotzdem ist Cast erforderlich, weil Typsystem für s.pop() nur den **statischen Typ** Object liefert
- Grund: Konversion zur Superklasse in push() verliert Information

Übergang zur Parametrischen Polymorphie

Bis auf Widerruf nehmen wir ab jetzt zur Vereinfachung an,
dass dynamischer Typ = statischer Typ

- Invariante über Inhalt von `s` hängt nur von Methode `push` ab:
 - Wenn `push` (als einzige Möglichkeit, neue Elemente auf den Stapel zu legen) nur Objekte von einem Typ α einfügt, dann
 - werden immer nur Objekte von Typ α auf dem Stapel liegen
 - kann `Node.value` als α deklariert werden
 - wird `pop` kann sicher α zurückliefern

Annahme: Am Anfang ist Stapel leer

⇒ Ein Stapel kann die Invariante intern aufrecht erhalten

⇒ Er kann dies für alle Typen α tun

Idee zur Generischen Stapel-Klasse

```
∀ $\alpha$ . class Stack {  
    private class Node {  
        private  $\alpha$  value;  
        private Node next;  
    }  
    private Node top;  
    public void push( $\alpha$  x) {  
        Node n = new Node();  
        n.next = top;  
        n.value = x;  
    }  
    public  $\alpha$  pop() {  
         $\alpha$  r = top.value;  
        top = top.next;  
        return r;  
    }  
}
```

Der Stapel mit Java Generics

```
class Stack<E> {
    private class Node {
        private E value;
        private Node next;
    }
    private Node top;
    public void push(E x) {
        Node n = new Node();
        n.next = top;
        n.value = x;
        top = n;
    }
    public E pop() {
        E r = top.value;
        top = top.next;
        return r;
    }
}
```

- **Parametrische Polymorphie:** Der Typ E wird Parameter von Stack
- `stack< α >` ist für jedes Typargument α benutzbar

Anwendung Generischer Stapel

```
Stack<Integer> s = new Stack<Integer>();  
for (int i=0; i<17; ++i)  
    s.push(new Integer(i));  
int sum = 0;  
while (!s.empty()) {  
    sum += s.pop().intValue();  
}
```

- Typargument für Stack gibt erlaubte Elementtypen vor
 - Bei push: Compiler prüft, dass nur Integer-Instanzen eingefügt werden
 - Bei pop: Compiler kann zusichern, dass das Ergebnis ein Integer ist
 - Cast kann entfallen
- ⇒ Invariante über Inhalt von s ausgedrückt

Generics für Container

- Beispiel Stapel verallgemeinert zu Containern
 - Container mit bestimmtem Elementtyp
 - Nehmen nur Elemente dieses Typs auf
 - Enthalten (als Invariante) nur Elemente dieses Typs
 - Geben bei Zugriffen nur Elemente dieses Typs zurück
- ⇒ Prototypischer Anwendungsfall für Generics
- ✓ Frühzeitige Fehlererkennung
 - ✓ Weniger Casts, übersichtlicherer Code

Anwendung: Binäre Methoden

- Erinnerung: Binäre Methoden sind schwierig
- Beispiel: Interface Comparable

```
interface Comparable {  
    boolean equal(Comparable b);  
    boolean less(Comparable b);  
}
```

- Wünschenswert: Das Argument `b` soll den gleichen Typ haben wie `this`.
- Haben: Eine Klasse `A`, die `Comparable` implementieren möchte, muß als Argument Instanzen jeder anderen Klasse akzeptieren, die `Comparable` implementiert.

Anwendung: Binäre Methoden

- Zur Implementierung muss eine Klasse Instanz-Checks durchführen

```
class Int implements Comparable {
    private int i;
    public boolean equal(Comparable b) {
        if (b instanceof Int)
            return ((Int)b).i == i;
        else return false;
    }
    public boolean less(Comparable b) {
        if (b instanceof Int)
            return i < ((Int)b).i;
        else throw new RuntimeException("Int.less");
    }
}
```

⇒ Das Typsystem trägt hier nicht zur Sicherheit bei.

Lösung mit Generics

```
interface Comparable<T> {
    boolean equal(T b);
    boolean less(T b);
}
class Int implements Comparable<Int> {
    private int i;
    public boolean equal(Int b) {
        return b.i == i;
    }
    public boolean less(Int b) {
        return i < b.i;
    }
}
```

- Lies `Comparable< α >` als “vergleichbar mit Typ α ”
 - Implementierende Klasse gibt sich selbst als α an
- ⇒ Keine Casts & `instanceof` notwendig
- ⇒ Typchecker garantiert fehlerfreien Ablauf

Compiler fängt Fehler ab

```
void test(Int i, String s) {  
    i.equal(s);  
}
```

Ergibt:

```
GComp.java:17: equal(Int) in Int cannot be  
                applied to (java.lang.String)  
    i.equal(s);  
      ^
```

Vergleich mit C++ Templates

- Java Generics
 - Generische Klasse wird nur einmal übersetzt
 - Gleicher Bytecode funktioniert für alle Instanzen
 - Typparameter können nur mit Klassen instanziiert werden
- C++ Templates
 - Instanziierung mit Klassen- und Basistypen
 - Neuübersetzung für jede benötigte Instanz
 - Großer Speicherbedarf im erzeugten Code (*code bloat*)
 - Lange Compilier-Zeit
 - Compiler-Optimierung für spezielle Typen
 - ⇒ Effizienz der Standard Template Library
 - Fehler treten erst bei Instanziierung auf
 - Beziehen sich auf Interna der Templates
 - Programmier kann nicht unterscheiden, ob Fehler in Benutzung oder Fehler in Template-Definition

Zwischenstand

- Generics sind eine sinnvolle Ergänzung zu klassischer OO
- Anwendungen:
 - Genaueres Wissen über dynamische Typen als Invarianten bewahren
 - Gleichheit von Typen festhalten
- Jetzt: Konzept “Polymorphie”
 - Ohne konzeptuelles Wissen kein wirkliches Verständnis konkreter Sprachkonstrukte (→ These dieser Vorlesung)
 - Einordnung: Die Grundideen von Generics sind nicht neu
- Schließlich:
 - Kombination von Generics und Subtypen
 - Ecken und Probleme bei Java Generics

Generics bisher

- Motivation: Subtyp-Polymorphie reicht nicht aus
- Beispiel: Stack für beliebige Objekte
 - ⇒ Bei `s.push(new Integer(42))` geht Information verloren
 - ⇒ Brauchen Cast bei `(Integer)s.pop()`
- Invariante: “Im Stapel sind nur Element eines bestimmten Typs E.”

```
class Stack<E> {  
    ...  
    public void push(E x) { ... }  
    public E pop() {  
}  
}
```

- E ist Parameter von Stack → [Parametrische Polymorphie](#)

Generics bisher

- Zweites Beispiel: Binäre Methoden

```
interface Comparable {
    boolean less(Comparable b);
}
class Int implements Comparable {
    private int i;
    public boolean less(Comparable b) {
        if (b instanceof Int) return i < ((Int)b).i;
        else throw new RuntimeException("Int.less");
    }
}
```

Generics bisher

- Besser: Typ für zweites Argument als Parameter

```
interface Comparable<T> {
    boolean less(T b);
}
class Int implements Comparable<Int> {
    public boolean less(Int b) {
        return i < b.i;
    }
}
```

Plan für heute

- Parametrische Polymorphie in Reinform: ML
- Interaktion: Subtypen und Parametrische Polymorphie
- Fallstricke bei Java Generics

Konzept: Polymorphie in ML

- ML hatte schon 1978 polymorphe Funktionen
- Idee: Solange der Code einer Funktion auf einen Teil seiner Daten nicht zugreift soll für diesen Teil auch kein Typ festgelegt werden.

```
# let fst = fun (x,y) -> x
  val fst : 'a * 'b -> 'a = <fun>
```

- `fst` erwartet als Argument ein Paar
- Das rechte Datum im Paar wird nicht benutzt → Typvariable `'b`
- Das linke Datum im Paar wird nur in die Ausgabe durchgereicht → Typvariable `'a` im Argument- und Ergebnistyp

Konzept: Polymorphie in ML

- Wieder die Idee: Funktion arbeitet *für alle* (\forall) Argumenttypen (α, β)

```
# fst (3,1);;  
- : int = 3  
# fst ("Die Antwort: ", 42);;  
- : string = "Die Antwort: "
```

⇒ Schreibe auch **Typschema** für `fst` als $\forall \alpha, \beta. \alpha \times \beta \rightarrow \alpha$

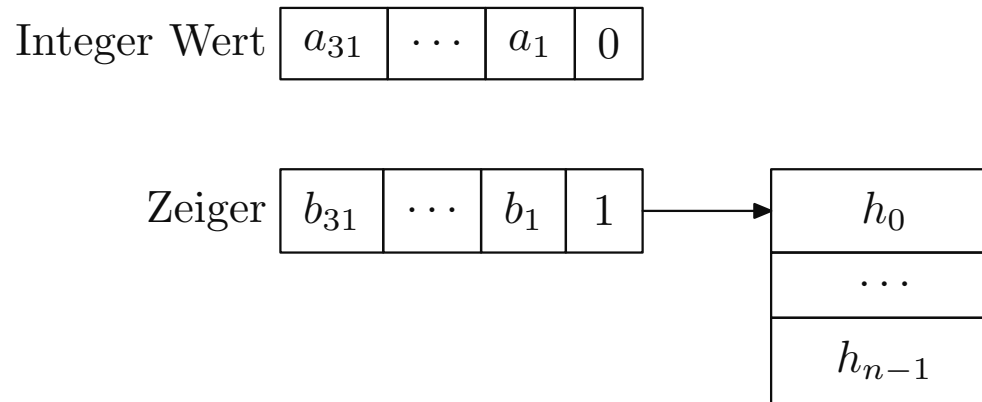
- Anwendungen:
 - (Homogene) Listen mit beliebigen Elementen
 - Funktionen höherer Ordnung (Funktoren)

```
# let twice f x = f (f x);;  
val twice : ('a -> 'a) -> 'a -> 'a = <fun>  
# twice (fun x -> 2 + x) 5;  
- : int = 9
```

ML-Polymorphie: Der technische Ansatz

- ? Woher weiss der Compiler, dass eine Funktion “für alle Typen” funktioniert?
- ! Er hat doch den Code selbst erzeugt und kann sicherstellen, dass darin keine Operationen auftreten, die nur bestimmte Typen gültig sind.
- ⇒ Alle Daten müssen in einheitlichem Format abgespeichert sein
- ⇒ Kopien unabhängig vom Typ anlegen
 - Problem: Keine direkte Verbindung zum Typsystem

ML-Polymorphie: Der technische Ansatz



- Mögliche Realisierung: Alle Daten sind Maschinenworte
 - Kleine Daten (`int`, `char`) haben $\text{Bit}_0 = 0$
(nur 31 Bit für Integer-Darstellung)
 - Alle anderen Daten liegen auf dem Heap
 - Register enthalten Zeiger mit $\text{Bit}_0=1$
 - Zeiger verweist auf Daten in einem Heap-Block
 - Kopie eines Wertes für Funktionsaufruf = Kopie des Maschinenwortes
- \Rightarrow Referenzsemantik von grossen Werten

ML-Polymorphie: Der Logische Ansatz

Die folgenden Folien verschweigen beinahe die gesamten technischen Details, sie sollen nur einen ersten Eindruck vermitteln.

? Woher weiss der Compiler, dass eine Funktion “für alle Typen” funktioniert?

• Gegenfrage: Wie beweist man, dass eine Aussage P für alle x gilt?

! Beweise P , ohne Annahmen über das x zu machen

• Aus der Analysis I: “Zu zeigen ist, dass f stetig bei z_0 ist. Nach Definition der Stetigkeit ist zu zeigen

$$\forall \varepsilon > 0. \exists \delta > 0. |z - z_0| \leq \delta \implies |f(z) - f(z_0)| \leq \varepsilon$$

Sei also ε beliebig aber fest gegeben”

Der \forall -Quantor in der Prädikatenlogik

- Schreibe also

$$\Gamma \vdash P$$

für “ P kann unter den Annahmen Γ bewiesen werden.”

- Für den \forall -Aussage gilt dann die Herleitungsregel:

$$\frac{\Gamma \vdash P}{\Gamma \vdash \forall x.P} \quad \text{wobei } x \text{ selbst nicht frei in } \Gamma \text{ vorkommt}$$

(Bemerkung: Die Herleitbarkeit von $\forall x.P$ hängt formal nicht vom speziellen x ab, da diese Variable als gebundene Variable beliebig umbenannt werden kann, so dass sie nicht in Γ vorkommt. Im Stetigkeitsbeispiel müssten wir dies tun, wenn wir den Namen ε schon in einem anderen Zusammenhang vergeben hätten.)

Polymorphes let in ML

- Übertragung auf ML-Polymorphie: Zeige $f : \forall\alpha.t$ für

$$e_1 = \text{let } f = e \text{ in } e'$$

- Beachte:

- e_1 ist im allgemeinen Teilausdruck eines größeren Ausdrucks e_0
- e kann Variablen enthalten, die weiter außen in e_0 gebunden sind

⇒ Benütze **Typannahmen** $\Gamma = \{x_1 : s_1, \dots, x_n : s_n\}$ für freie Variablen in e

- $\Gamma \vdash e : t$ bedeutet nun: “Unter den Typannahmen Γ (über die freien Variablen in e) hat e selbst den Typ t ”
- Polymorphie: Konstruktion analog zur Prädikatenlogik

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e : \forall\alpha.t} \quad \alpha \text{ kommt nicht frei in } \Gamma \text{ vor}$$

Typinferenz

- Der ML Compiler kann **alle** Typen von Funktionsargumenten und let-gebundenen Variablen berechnen → **Typinferenz** (oder **Typrekonstruktion**)
 - Beachte: Ein C/C++/Java Compiler berechnet die Ergebnistypen von Ausdrücken, trotzdem spricht man hier nicht von Typinferenz
 - Essentiell für Erfolg von Typinferenz in ML:
 - Jeder Teilausdruck besitzt einen allgemeinsten Typ (principal type)
 - Beachte: Durch Typvariablen kann Ausdruck mehr als einen Typ haben
 - Alle möglichen Typen des Ausdrucks ergeben sich durch Ersetzen von Typvariablen im Principal Type
 - Der Principal Type ist der “beste” oder “genaueste” Typ, den man dem Ausdruck zuordnen kann
- ⇒ Der Compiler braucht einmal berechneten Typ nicht mehr zu korrigieren
- ? Wir berechnet man den Principal Type?

Einschub: Gleichungen und Unifikation

- Bei Funktionsaufruf muss Typ des Arguments gleich Parameter-Typ sein
- Aber: Typvariablen dürfen ersetzt werden
- Löse Gleichung $s = t$ (für Argumenttyp s und Parametertyp t)
- Formal: Finde Ersetzung σ von Variablen durch Typen mit

$$\sigma(s) = \sigma(t)$$

- Wir sagen, dass σ eine Lösung oder ein **Unifikator** der Gleichung ist

Satz (Robinson, 1965): Wenn eine Gleichung $s = t$ eine Lösung besitzt, dann besitzt sie auch eine Lösung σ_0 , so dass sich jede andere Lösung σ schreiben lässt als $\sigma = \sigma' \circ \sigma_0$ für ein σ' . σ_0 heisst auch der **allgemeinste Unifikator** von $s = t$. σ_0 ist eindeutig bis auf Umbenennung von Variablen und wir schreiben $\sigma_0 = \text{mgu}(s, t)$.

Typinferenz: Der Funktionsaufruf

- Gesucht: Funktion ti für Typinferenz
 - Eingabe sind Typannahmen Γ und Ausdruck e
 - Typinferenz muss Gleichungen lösen
 - Ergebnis ist Paar von Typ und Unifikator
 - Hinweis: Unifikator muss in allen späteren Schritten angewandt werden
- Für Funktionsaufruf $ti(\Gamma, (f e))$
 - Berechne $\langle s, \sigma_s \rangle = ti(\Gamma, f)$
 - Berechne $\langle t, \sigma_t \rangle = ti(\sigma_s(\Gamma), e)$
 - Löse $\sigma = mgu(\sigma_t(\sigma_s(s)), \alpha \rightarrow \beta)$ für neue Variablen α, β
 - Löse $\sigma' = mgu(\sigma(\alpha), \sigma(\sigma_t(t)))$ $\Rightarrow \sigma' \circ \sigma \circ \sigma_t \circ \sigma_s$ ist Lösung für $s = \alpha \rightarrow \beta$ und $\alpha = t$
 - Ergebnis: $\langle s_2, \sigma' \circ \sigma \circ \sigma_t \circ \sigma_s \rangle$

Typinferenz: Funktionen und let

- Für Funktion $\text{ti}(\Gamma, \text{fun } x \rightarrow e)$
 - Wähle neue Typvariable $'a \notin \text{FV}(\Gamma)$
 - Berechne $\langle t, \sigma \rangle = \text{ti}(\Gamma_x \cup \{x : 'a\}, e)$
 - Ergebnis ist $\langle 'a \rightarrow t, \sigma \rangle$
- Beachte: $'a$ kann ersetzt werden, je nach Verwendung von x in e
- Für Bindung $\text{ti}(\Gamma, \text{let } x = e \text{ in } e')$
 - Berechne $\langle s, \sigma \rangle = \text{ti}(\Gamma, e)$
 - Berechne $\{ 'a_1 \dots 'a_n \} = \text{FV}(\sigma(s)) \setminus \text{FV}(\sigma(\Gamma))$
(\rightarrow Alle Variablen, in denen e polymorph ist)
 - Berechne $\langle t, \sigma' \rangle = \text{ti}(\sigma(\Gamma_x) \cup \{x : \forall 'a_1 \dots 'a_n. \sigma(s)\}, e')$
 - Ergebnis: $\langle t, \sigma' \circ \sigma \rangle$

Typinferenz am Beispiel

$ti(\Gamma, \text{let } f\text{st} = \text{fun } (x, y) \rightarrow x)$

- Wähle Typvariablen α, β für x und y
 - Ergebnistyp ist α
 - α, β sind frisch gewählt, kommen sicher nicht in Γ vor
- \Rightarrow Typ von $f\text{st}$ ist $\forall \alpha, \beta. \alpha \times \beta \rightarrow \alpha$
- Ersetzungen sind nicht notwendig (keine Funktionsanwendung)

Typinferenz am Beispiel

```
let f = fun x ->  
  let g = fun b y ->  
    if b then x else y  
  in g true x
```

- Wähle Typvariablen α, β, γ für x, b, y
- Kontext von g enthält Typannahme für x
- Berechne $ti(\{x : \alpha\}, \text{fun } b \ y \ -> \ \text{if } b \ \text{then } x \ \text{else } y)$
- Berechne $ti(\{x : \alpha, b : \beta, y : \gamma\}, \text{if } b \ \text{then } x \ \text{else } y)$
- Löse Gleichungen $\{\beta = \text{bool}, \alpha = \gamma\} \rightarrow \sigma = \{\beta \mapsto \text{bool}, \alpha \mapsto \gamma\}$
- Haben für g Typ $\text{bool} \rightarrow \gamma \rightarrow \gamma$ in Kontext $\sigma(\{x : \alpha\}) = \{x : \gamma\}$
- Typ von g kann *nicht* über γ quantifiziert werden,
da $FV(\text{bool} \rightarrow \gamma \rightarrow \gamma) \setminus FV(\{x : \gamma\}) = \emptyset$

ML-Typinferenz

- Typinferenz rekursiv über Syntaxbaum
 - Für jeden Teilausdruck kann ein Principal Typ berechnet werden
 - Idee: Lösung von Typ-Gleichungen durch Ersetzung von Typvariablen
 - Polymorphie bei `let`: \forall -“Quantifizierung” derjenigen Typvariablen, die nicht im Kontext vorkommen
- ⇒ Funktion arbeitet dann mit allen Typen

Subklassen-Polymorphie

- OO-Sprachen besitzen schon Polymorphie: Wenn Klasse B Subklasse von A sind, kann eine B -Instanz überall dort benutzt werden, wo eine A -Instanz erwartet wird
 - Prinzip: Statischer Typ ist Supertyp des dynamischen Typs
 - Methoden arbeiten auf mehreren Typen
 - Konversion zu Superklasse ist erlaubt
 - Beziehung Subtyp-Beziehung ist transitiv, reflexiv und antisymmetrisch
- ⇒ Halbordnung $s \leq t$ auf Typen: s ist Subtyp von t (evtl. $s = t$)

Vergleich: Subklassen- / ML-Polymorphie

- Aspekt: Invarianten
 - Rückgabetyt einer polymorphen Methode hängt nicht vom aktuellen Argumenttyp ab
 - ⇒ Subtyp-Polymorphie verliert Information
 - Parametrische Polymorphie erhält Informationen
 - Zwischen den Signaturen verschiedener Methoden → Klasseninvariante
 - Zwischen Argumenten und Rückgabe einer Methode → Invariante der Methode über Verwendung der Argumente
- Aspekt: Beziehungen zwischen Typen
 - ML kennt nur Typgleichheit → Lösung von Gleichungen bei Funktionsaufruf
 - OO-Sprachen: Methodenauf, Zuweisung, Lesen definieren **Ungleichungen** zwischen Typen

Einschub: Kann es Typinferenz für Java geben?

- Ziel: Compiler berechnet Typen für Parameter, lokale Variablen, Felder
- Naive Idee: Durchlaufe rekursiv durch die gesamte Klasse
 - Wähle für jede Variable eine neue Typvariable
 - Sammle Ungleichungen auf, die erfüllt sein müssen z.B. für Zuweisung $x = e$ (mit $x : \alpha$, $e : t$) gilt $t \leq \alpha$
 - Löse Ungleichungen
 - Ersetze Typvariablen durch gefundene Typen

Probleme bei Typinferenz für Java

- Ziel: Schwierige Beispiele angeben
- Ungleichungen besitzen keine “allgemeinste” Lösung
 - Parameter $x : \alpha$ werde zugewiesen an $y : \beta$ und $z : \gamma$
 - Haben $\alpha \leq \beta, \alpha \leq \gamma$
 - Ohne Mehrfachvererbung muss entweder $\beta \leq \gamma$ oder $\gamma \leq \beta$ sein
 - Mit Interfaces: Sollen β und/oder γ Interfaces sein?
 - ⇒ Compiler kann nicht für den Programmierer entscheiden
 - Variable $x : \alpha$ erhalte durch Zuweisung Werte aus $y : \beta$ und $z : \gamma$
 - $\beta \leq \alpha$ und $\gamma \leq \alpha \rightarrow \alpha$ ist Superklasse von β und γ
 - ⇒ Es kann mehrere mögliche Supertypen geben (mit Interfaces!)
 - ⇒ Compiler kann nicht für den Programmierer entscheiden
- ⇒ Compiler kann lokal keine Entscheidung über Lösungen treffen
 - Der Typ eines Teilausdrucks kann von allen anderen Typen abhängen
 - ⇒ Keine Principal Types, wir müssen Ungleichungssysteme schlimmstenfalls global für das gesamte (!) Programm lösen
 - ⇒ Äquivalent zu globaler Datenflussanalyse → rechenintensiv

Probleme bei Typinferenz für Java

- Pragmatisch: Programmierer muss häufig Casts einfügen; diese **widersprechen** gerade den Ungleichungen, sie leiten sich aus Invarianten über das Programm ab und können daher nicht sinnvoll automatisch eingefügt werden
- Technisch: Methodenaufruf $a.m(e)$ erfordert statischen Typ von a , um dort Signatur von m nachzuschlagen, wir haben aber nur Ungleichungen für a
Lösung: Führe Ungleichung $a \leq \{m : s \rightarrow t\}$ ein mit Aussage: “ a ist ein Record-Typ, in dem mindestens die Methode m vorkommt.”

Ansätze

- **Mache starke Annahmen über die Klassenhierarchie: Verband**

Rehof, Mogensen: “Tractable constraints in finite semilattices” Science of Computer Programming 35, 1999
- **Vereinfache nur die zu lösenden Ungleichungen, statt eine einzige Lösung zu finden**

Pottier: “Simplifying Subtyping Constraints: A Theory”, Information and Computation 170(2), 2001
Eifrig, Smith, Trifonov: “Sound Polymorphic Type Inference for Objects”, OOPSLA 1995
- **Lokale Typinferenz: Kontext der Typinferenz begrenzen**

Pierce, Turner: “Local Type Inference”, Transactions on Programming Languages and Systems 22(1), 2000
- **Heuristiken: Schränke Typinferenz so ein, daß die Erwartungen des Programmierers erfüllt werden**

Cardelli: “An implementation of $F_{<}$ ” DEC Technical Report, 1993

Zusammenfassung: Typinferenz für OO-Sprachen

- Ungleichungen machen Typinferenz schwierig, da keine allgemeinsten Lösungen existieren
- Wenn dann muss das gesamte Programm betrachtet werden
- Es gibt einige pragmatische Probleme: Casts & Methoden-Signaturen

Schaue sehr kritisch, wenn jemand behauptet, eine OO-Sprache mit Typinferenz anzubieten — irgendwo muss diese Typinferenz das eigentliche Problem sehr stark beschränken, um noch zu Lösungen zu kommen.

Verbindung: Bounded Parametric Polymorphism

- Zurück zu vollständiger Typannotation!
- Häufig arbeitet Methode / Klasse auf einem beliebigen Typ, *solange* dieser nur ein Subtyp einer bestimmten Klasse ist
- Kombiniere das beste aus beiden Arten der Polymorphie
 - Parametrische Polymorphie kann Typ von Argument zu Ergebnis übertragen
 - Subtyp-Anforderung bietet Subtyp-Polymorphie mit Dynamic Dispatch

Beispiel: Bewegung eines Punktes

```
class Point {
    public int x;
    public int y;
};
class ColorPoint extends Point {
    public Color c;
};
class Mover {
    public static <T extends Point>
        T moveBy(T p, int dx, int dy) {
        p.x += dx;
        p.y += dy;
        return p;
    }
    public void test(ColorPoint p) {
        ColorPoint q = moveBy(p,1,2);
    }
};
```

Allgemeiner: Änderung per Seiteneffekt an Teil des Zustandes

F-bounded Polymorphism

```
interface Comparable<T> {
    boolean less(T b);
}
class Int implements Comparable<Int> { ... }
class Sorter<T extends Comparable<T> > {
    boolean test(T x, T y) {
        return x.less(y);
    }
};
```

- Schon gesehen: Lösung für binäre Methoden
 - Neu: Sorter beschränkt Parameter T durch Interface, das T selbst als Typparameter nimmt
- ⇒ Sorter hat Parameter der Form $t \leq F[t]$
- Canning, Cook, Hill, Olthoff, Mitchell: “F-Bounded Polymorphism for Object-Oriented Programming” FPLCA '89

Einschub: Array-Typen in Java

- `A[]` ist Array mit Objekten, die gemeinsame Superklasse `A` besitzen
- Idee: Wollen Methode schreiben, die mit `A`-Arrays umgeht

```
public static void f(A a[]) {  
    A x = a[0];  
    ...  
}
```

- Java ist großzügig: Wenn `B` eine Subklasse von `A` ist, darf man auch ein `B`-Array an `f` übergeben, schliesslich sind ja alle Elemente tatsächlich Instanzen auch von `A` (per Polymorphie)

```
B b[] = new B[2];  
f(b);
```

- In Java sind Arrays **covariant** getypt: Wenn $A \leq B$, dann ist auch $A[] \leq B[]$

⇒ Sehr praktisch in vielen alltäglichen Programmiersituationen

Einschub: Array-Typen in Java

- Problem: f kann auch in das A Array schreiben

```
public static void f(A a[]) {  
    A x = a[0];  
    a[1] = new A();  
}
```

- Nach Rückkehr aus Funktion steht an b[1] eine A-Instanz!

```
B b[] = new B[2];  
f(b);  
B y = b[1];
```

- Der Typchecker hat diese Fehlermöglichkeit übersehen

```
Exception in thread "main" java.lang.ArrayStoreException: A
```

- Java ist *nicht* streng gettypt, da trotz erfolgreichem Typcheck Laufzeitfehler wegen falscher dynamischer Typen auftreten können.

Einschub: Array-Typen in Java

- Typchecker prüft nicht alle Fehlermöglichkeiten

⇒ Laufzeitprüfung muss nachfolgen

- Array muss zur Laufzeit speichern, welche Typen eingefügt werden können
- Array enthält Verweis auf die Klasse, für die es angelegt wurde
- Bei jedem schreibenden Zugriff wird die Invariante geprüft, dass das neue Element tatsächlich eine Instanz dieser Klasse (oder einer ihrer Subklassen) ist

Folgerung: Keine Varianz für Typparameter

- Manchmal gut: $\text{List}\langle A \rangle \leq \text{List}\langle B \rangle$ falls $A \leq B$
- ⇒ `List` sollte covariant in Typparameter sein
- Kein Problem, solange alle B -Felder in $\text{List}\langle B \rangle$ nur gelesen werden
- ⇒ Kann nicht sichergestellt werden
- (Überlege Dir an einem Beispiel, dass auch die **Contravarianz**-Regel $\text{List}\langle A \rangle \leq \text{List}\langle B \rangle$ falls $B \leq A$ zu Laufzeit-Fehlern führt.)
- ⇒ Generische Klassen sind immer **invariant** in ihren Parametern
- ⇒ $C\langle A \rangle$ und $C\langle B \rangle$ sind völlig verschiedene Typen, unabhängig davon ob A und B Subklassen voneinander sind
- Abhilfe: Wildcards = unbenannte / unbekannte Typparameter

Design-Entscheidung zu Java Generics

- Oberstes Designziel von Java Generics: Abwärtskompatibilität
 - Nicht-generischer Code läuft mit neuer Standard-Bibliothek
 - Code für Generische Klassen sollte auch noch auf älterer JVM laufen
 - Der Bytecode-Verifier der JVM soll wie bisher funktionieren
- ⇒ `List<A>` muss die gleiche Implementierung haben wie `List` selbst
- ⇒ Typparameter `A` darf in der Implementierung keine Rolle spielen, nur in der Schnittstelle

Type Erasure

Übersetzung einer Klasse, die Generics benutzt:

- Typcheck für generischen Code mit Typparametern
- ⇒ Code respektiert Invarianten, die durch Typparameter gegeben sind
- Lösche Typparameter aus dem Code
 - Ersetze alle Typparameter durch ihre Bounds
 - Füge wo nötig Casts ein (diese schlagen nie fehl)
 - Übersetze entstandenes Programm, es enthält keine Generics mehr

Folgen der Type Erasure

- Zuweisung von `List<A>` an `List` erlaubt
- Zuweisung von `List` an `List<A>` ergibt nur Warnung
- Keine Laufzeit-Typinformationen zu Typ-Parametern
- Daraus resultieren Einschränkungen
 - Es können keine Arrays über Typ-Parametern angelegt werden (wohl aber verarbeitet)
 - ⇒ Generische Vector-Klasse muss intern Casts anwenden
 - Alle Instanzen von `C<A>` teilen sich die `static`-Felder
 - ⇒ Typen für `static`-Members können keine Typ-Parameter enthalten
 - Casts & `instanceof` sind nicht gegen Typparameter möglich

Keine Generischen Arrays

```
class Vec<E> {
    private E store[];
    private int capacity;
    private int size;
    public int getSize() { return size; }
    private void ensureExists(int i) {
        ... new E[capacity] ...
    }
    public void put(int i, E x) {
        ensureExists(i);
        store[i] = x;
        if (i >= size) size = i + 1;
    }
    public E get(int i) {
        if (i < size) return store[i];
        else return null;
    }
}
```

Zusammenfassung

- Java Generics gründen im Konzept der Polymorphie
- Polymorphie ist ein sauber fundierter, wohluntersuchter Begriff
- Typinferenz für OO-Sprachen ist schwer, wenn nicht unmöglich
- Subtypen und Parametrische Polymorphie ergänzen sich
- Java Generics
 - Lösen viele alltägliche Probleme elegant
 - Sind abwärtskompatibel zu existierendem Code
 - Haben viele Ecken (→ Typparameter zur Laufzeit nicht verfügbar)

Am Donnerstag

- Die Vorlesung anhand von Thesen
 - Interaktiv: Wir bringen die Thesen, ihr diskutiert sie
 - Wir bleiben Moderatoren
 - Bitte jeweils 1-2 Themenblöcke kurz nacharbeiten
- Lernstrategien und -ziele für die Diplomprüfung
 - Unsere Vorstellung von “Verstandenem Stoff”
 - Unsere Strategien zur Verwirklichung der Ziele
- Prüfungssimulation
 - Holger & Martin spielen Prüfer und Prüflinge
 - Wir diskutieren zusammen über Ziele der Prüfung