Kapitel 17

Generizität, Abstraktion, Rekursion

17.1 Generische Klassen

Fragen:

- was ist der Typ einer generischen Klasse?
- Kann man zwischen generischen Klassen Vererbung definieren?
- Kann man zwischen generischen Instanzen Vererbung definieren?

Zu praktischen Gesichtspunkten generischer Klassen, insbesondere in Java 1.5, siehe Kapitel 10

parametrisierte Klassen haben *polymorphe Typen*:

$$\forall \alpha \leq \sigma.\tau$$

"alle Typen, die entstehen, wenn in τ jedes α durch irgendeinen Typ σ' ersetzt wird, wobei $\sigma' \leq \sigma$ "

Beispiel:

```
class G<P extends Point> {
   P move (P x, int i) {...};
   P set(int i) {...}; }
}
```

hat Typ

$$G\colon \forall \alpha \leq Point.\{move\colon \alpha \to int \to \alpha, set\colon int \to \alpha\}$$

Instantiierung: Sei $\tau' = \forall \alpha \leq \sigma.\tau$, $\sigma' \leq \sigma$, so

$$\tau'[\sigma'] = \tau[\alpha \leftarrow \sigma']$$

analog β-Reduktion im λ-Kalkül!

σ' darf selbst kein polymorpher Typ sein
(vgl Damas-Milner Typsystem)!

parametrisierte Klassen können als *Funktoren* aufgefaßt werden: sie nehmen eine Klasse (zB `StackElem`) und erzeugen daraus eine neue (`Stack<StackElem>`)

Bsp: `G<SP>`: $\{move\colon SP \to int \to SP, set\colon int \to SP\}$

$\Rightarrow$ generische Klassen entsprechen *benutzerdefinierten* Typkonstruktoren

17.2 Vererbung und Generizität

1. Vererbung von generischen Instanzen (vgl Array-Anomalie)

Sei `A` $\leq$ `B`. Gegeben sei generische Klasse `class C<X> { X x;}`

Annahme: `C<A>` $\leq$ `C<B>`. Betrachte

```
C<B> o = new C<A>();
o.x = new B();
```

ist statisch korrekt, führt aber zu illegalem Downcast.

Annahme: `C<B>` $\leq$ `C<A>`. Betrachte

```
C<A> o = new C<B>();
A a = o.x;
```

führt ebenfalls zu illegalem Downcast!

$\Rightarrow$ es kann zwischen generischen Instanzen keine Vererbung geben!

2. Vererbung von generische Klassen bzw polymorphen Typen

Bsp: `class H<P extends Point> extends G<P> {...}`

Vorschlag: Konversionsregel (Kontravarianz in der Typschranke, da α Funktor-Argument!):

$$\frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{\forall \alpha \leq \sigma . \tau \ \leq \ \forall \alpha \leq \sigma' . \tau'}$$

Leider funktioniert das nicht.

Gegenbeispiel: Sei $S \leq P$

$$\begin{aligned} S \leq P \;\Rightarrow\; & \forall \alpha \leq S.\{f : int \to \alpha, g : \alpha \to int\} \\ & \geq \forall \alpha \leq P.\{f : int \to \alpha, g : \alpha \to int\} \\ \Rightarrow\; & \{f : int \to S, g : S \to int\} \\ & \geq \{f : int \to P, g : P \to int\} \\ \Rightarrow\; & int \to S \geq int \to P \quad \wedge \quad S \to int \geq P \to int \\ \Rightarrow\; & S \geq P \quad \wedge \quad P \geq S \text{ Kontravarianz!} \\ \Rightarrow\; & S = P \end{aligned}$$

Analog falls Kovarianz in der Typschranke. Ergo müssen die Typschranken gleich sein:

$$\frac{\tau \leq \tau'}{\forall \alpha \leq \sigma.\tau \;\leq\; \forall \alpha \leq \sigma.\tau'}$$

Beispiel:

$$\begin{aligned} & \{u : \alpha, v : int\} \leq \{u : \alpha\} \\ \Rightarrow\; & \forall \alpha \leq \sigma.\{u : \alpha, v : int\} \leq \forall \alpha \leq \sigma.\{u : \alpha\} \end{aligned}$$

Mithin ist

```
class C<X extends S> { X u;}
class D<X extends S> extends
      C<X extends S> { int v;}
```

ok. Es muss in beiden Klassen `X` heißen, da α einheitlich ersetzt werden muss.

17.3 Schnitt-Typen

in C++ ist Signatur nicht Teil des Namens

⇒ überladene Bezeichner haben mehrere Typen gleichzeitig!

Schreibweise: $\tau \wedge \sigma$

Beispiel: überladene Methode f

$$f : (int \to int) \wedge (real \to real)$$

Konversionsregel:

$$\tau \leq \sigma \text{ und } \tau' \leq \sigma \;\Rightarrow\; \tau \wedge \tau' \leq \sigma$$

Introduktions/Eliminationsregel: Übung!

17.4 Existential Types

In Java, C++ gibt es abstrakte Klassen

Damit ein Programm funktioniert, muß es dazu Implementierung(en) geben

⇒ abstrakte Klassen haben existentielle Typen:

$$\exists \alpha \leq \sigma.\tau$$

τ steht für die Signatur der abstrakten Klasse, α für die Implementierung der abstrakten Klasse

Beispiel 1: abstrakte int-Liste

```
abstract class L {
    int car(); L cons(int x); L cdr();
}
```

$L\colon \exists \alpha \leq \top.\{car :\rightarrow int, cons : int \rightarrow \alpha, cdr :\rightarrow \alpha\}$

Instantiierung: ist L' Implementierung zu L mit
$L\colon \exists \alpha \leq \sigma.\tau$, $L' : \sigma' \leq \sigma$, so gilt

$$L' : \tau[\alpha \leftarrow \sigma']$$

im Beispiel:
$L' : \{car :\rightarrow int, cons : int \rightarrow L', cdr :\rightarrow L'\}$

Beispiel 2: abstrakte sortierte polymorphe Liste

```
abstract class L<B extends Ordinal>  {
    B car(); L cons(B x); L cdr();
}
```

$L\colon \exists\alpha.\forall\beta \leq \mathrm{Ordinal}.\{car :\rightarrow \beta, cons : \beta \rightarrow \alpha, cdr :\rightarrow \alpha\}$

Die Reihenfolge der Quantoren ist bedeutsam: α ist eine universelle Implementierung für alle generischen Instanzen β (α-Code ist unabhängig von β-Instanz)

Die umgekehrte Instantiierungsreihenfolge ist schwächer: bei $\forall\beta.\exists\alpha...$ wird β zuerst instantiiert, d.h. es entstehen lauter abstrakte Klassen, die dann verschieden implementiert werden können (nicht universell, unabhängig von β)

Dies entspricht den Regeln der gewöhnlichen Prädikatenlogik:

$$\exists x.\forall y.P(x,y) \Rightarrow \forall y.\exists x.P(x,y)$$

(vgl. Stetigkeit / gleichmäßige Stetigkeit in der Analysis: „universelles δ_ϵ“ vs „individuelles δ_ϵ“)

Bem.: Man kann auch für abstrakte Klassen Vererbung und $\leq$ definieren (Übung)

17.5 Rekursive Typen (Cook 1990)

Objekte enthalten oft Verweise auf andere Objekte desselben Typs

⇒ (indirekt) rekursive Typen:

$$\tau = \{\ldots, m : \tau, \ldots\}$$

Bsp: $list = \{key : int, next : list, cdr : list \to list\}$

rechte Seite ist Typkonstruktor:

$$\alpha : \tau \mapsto \{key : int, next : \tau, cdr : \tau \to \tau\}$$

Idee: suche Fixpunkt von α, also σ mit $\alpha(\sigma) = \sigma$

denn Fixpunkt löst Rekursionsgleichung:

$$\alpha(list) = \{key : int, next : list, cdr : list \to list\} = list$$

Tatsächlich existiert Fixpunkt immer ($\rightsquigarrow$ Domaintheorie). Dazu definieren wir `null: void`, wobei `void` mathematisch der leeren Menge entspricht (s.o.: ein Typ legt die Menge der möglichen Werte fest)

Dann kann man die Fixpunktkonstruktion analog Knaster-Tarski machen:

$$\begin{aligned} F_0 &= void \\ F_{i+1} &= \alpha(F_i(t)) \cup void \\ Fix(\alpha) &= \lim_{n\to\infty} F_n = \bigcup_{i=0}^{\infty} \alpha^i(void) \end{aligned}$$

denn Objekte können auch **`null`** sein

Es ist

$$\begin{aligned}
\alpha(Fix(\alpha)) &= \alpha(\bigcup_{i=0}^{\infty} \alpha^i(void)) = \bigcup_{i=0}^{\infty} \alpha(\alpha^i(void))) \\
&= \bigcup_{i=1}^{\infty} \alpha^i(void)) = \varnothing \cup \bigcup_{i=1}^{\infty} \alpha^i(void)) \\
&= \alpha^0(void) \cup \bigcup_{i=1}^{\infty} \alpha^i(void)) = \bigcup_{i=0}^{\infty} \alpha^i(void)) \\
&= Fix(\alpha)
\end{aligned}$$

im Beispiel (Mengenklammern sind weggelassen):

$$\begin{aligned}
F_0 &= void = \varnothing \\
F_1 &= \{key : int, next : void, cdr : void \to void\}, void \\
F_2 &= \{key : int, next : \{key : int, next : void, \\
&\qquad cdr : void \to void\}, cdr : \{\ldots\} \to \{\ldots\}\} \cup F_1 \\
F_3 &= \{key : int, next : \{key : int, next : \{key : int, \\
&\qquad next : \{key : int, next : void, cdr : ...\}, cdr : ...\}\}\} \\
&\qquad \cup F_2
\end{aligned}$$

Offenbar ist F_i der Typ aller Listen mit Länge $\leq i$

Schreibweise:

$$list = \mu\tau.\{key : int, next : \tau, cdr : \tau \to \tau\}$$

bzw allgemein

$$Fix(\alpha) = \mu\tau.\alpha(\tau) = \mu\tau.\{\ldots, m : \tau, \ldots\}$$

Problem: Rekursive Klassen und Vererbung

Konversionsregel: (induktiv)

$$\frac{\forall \sigma, \tau : \ \sigma \leq \tau \ \Rightarrow \ \alpha(\sigma) \leq \beta(\tau)}{\mu\sigma.\alpha(\sigma) \ \leq \ \mu\tau.\beta(\tau)}$$

Intuitiv: Die „unendliche Aufrollung“ von α ist Subtyp der „unendlichen Aufrollung“ von β

Beispiel: sei $\alpha(\tau) = \{a : i, b : \tau, c : \tau \to \tau\}$,
$\beta(\tau) = \{a : i, b : \tau\}$. Dann

$$\tau \leq \sigma \Rightarrow \alpha(\tau) = \{a : i, b : \tau, c : \tau \to \tau\} \leq \beta(\sigma) = \{a : i, b : \sigma\}$$

deshalb

$$Fix(\alpha) = \mu\tau.\{a : i, b : \tau, c : \tau \to \tau\} \leq Fix(\beta) = \mu\sigma.\{a : i, b : \sigma\}$$

Beweis der Regel: nach Vor. gilt

$$\begin{aligned}
& void \leq void \\
\Rightarrow \ & \alpha(void) \leq \beta(void) \\
\Rightarrow \ & \alpha(\alpha(void)) \leq \beta(\beta(void)) \\
\Rightarrow \ & \ldots \text{ Induktion!} \\
\Rightarrow \ & \forall i : \alpha^i(void) \leq \beta^i(void) \\
\Rightarrow \ & \bigcup_{i=0}^{\infty} \{\alpha^i(void)\} \ \leq \ \bigcup_{i=0}^{\infty} \{\beta^i(void)\} \\
\Rightarrow \ & \mu\sigma.\alpha(\sigma) \ \leq \ \mu\tau.\beta(\tau)
\end{aligned}$$

Nun betrachte

$$\begin{aligned}
\alpha(\tau) &= \{a : i, b : \tau, c : \tau \to \tau\} \\
Fix(\alpha) = T &= \mu\tau.\{a : i, b : \tau, c : \tau \to \tau\} \\
T_1 &= \{a : i, b : T, c : T \to T, d : b\} \\
\beta(\tau) &= \{a : i, b : \tau, c : \tau \to \tau, d : b\} \\
\gamma(\tau) &= \{a : i, b : \tau, c : T \to \tau, d : b\} \\
Fix(\gamma) = T_2 &= \mu\tau.\{a : i, b : \tau, c : T \to \tau, d : b\} \\
Fix(\beta) = T_3 &= \mu\tau.\{a : i, b : \tau, c : \tau \to \tau, d : b\}
\end{aligned}$$

Satz. 1. $T_1 \leq T$.
Beweis: $T = Fix(\alpha)$, also $\alpha(T) = T = \{a : i, b : T, c : T \to T\}$, also $T_1 \leq \alpha(T) = T$

2. $T_2 \leq T$.
Beweis: Unter der Annahme $\tau \leq T$ ist auch

$\gamma(\tau) = \{a : i, b : \tau, c : T \to \tau, d : b\} \leq \{a : i, b : T, c : T \to T\} = \alpha(T)$ und deshalb $fix(\gamma) = T_2 \leq T = fix(\alpha)$ nach Fixpunktregel

3. $T_3 \not\leq T$.
Beweis: wg. $T_3 = Fix(\beta)$ ist $\beta(T_3) = T_3$. Angenommen, $T_3 \leq T$. Dann ist auch $\beta(T_3) = T_3 \leq T = \alpha(T)$, also $\{a : i, b : T_3, c : T_3 \to T_3, d : b\} \leq \{a : i, b : T, c : T \to T\}$. Daraus folgt für c: $T_3 \to T_3 \leq T \to T$, also $T_3 = T$ (Kontravarianz), also Widerspruch!!

Betrachten wir nun

```
class List {
  int a; List n; List cdr (List x);
}
class DList extends List {
  bool d;
}
class DList' extends List {
  DList' n;  DList' cdr(DList' x); bool d;
}
```

Natürlich ist `DList` Subtyp von `List` (vgl $T_1 \leq T$), aber `DList'` ist kein Subtyp von `List` (vgl $T_3 \not\leq T$) !

$\Rightarrow$ Compiler muss T_3 Interpretation ablehnen !

$\Rightarrow$ Subtyping $\neq$ Inheritance !! (`Dlist'` ist kein Subtyp, erbt aber a)

Java lehnt auch T_3-Interpretation ab: in Java hat ein `DList'`-Objekt zwei n-Members und nicht eins wie in T_3

Dh in Java kann man T_3 nicht definieren wg „Redefinition nur mit gleicher Signatur“-Prinzip

in Eiffel kann man aber mittels `LIKE` **`Current`** T_3 erzeugen!

Tatsächlich führt $T_3 \leq T$ zu Laufzeitfehler wg verletzter Kontravarianz für cdr!