

# Kapitel 16

## Cardelli-Typsystem

Wozu Typen und Typsysteme?

*Typen legen die möglichen Werte von Variablen fest*

⇒ Compiler können effizienten Code generieren

⇒ bestimmte Laufzeitfehler werden garantiert verhindert („Well typed programs can't go wrong“ [Milner])

⇒ Verständlichkeit/Wartbarkeit wird besser

in OO-Sprachen:

- im Prinzip Typen  $\simeq$  Klassen, nebst elementaren Typen und Vererbung
- Typsysteme verhindern *illegale Downcasts*

Cardelli-Kalkül:

- elementare Typen: *int*, *real*, *bool*, ...
- Typvariablen:  $\tau, \sigma, \tau', \sigma'', \tau_i, \sigma_j$
- Objekt hat Typ:  $x : \tau$
- Funktionstypen (nur 1 Argument:  $f(x) = E$ ):  $f : \sigma \rightarrow \tau$

Bsp:  $f : real \rightarrow bool$

Schreibweise:  $f = \lambda x. E : \sigma \rightarrow \tau$

$\rightarrow$ -Introduktionsregel:

$$\frac{E : \tau \quad x : \sigma}{\lambda x. E : \sigma \rightarrow \tau}$$

$\rightarrow$ -Eliminationsregel:

$$\frac{f : \sigma \rightarrow \tau \quad x : \sigma}{f(x) : \tau}$$

Introduktions- und Eliminationsregeln gibt es in allen logischen Kalkülen;

typisches Beispiel:  $\forall, \exists, \wedge, \vee$  in Prädikatenlogik

Introduktion ist wie „Investition“, die in Elimination „genutzt“ wird

- Objekttypen:

$$\{m_1: \tau_1, m_2: \tau_2, \dots, m_n: \tau_n\}$$

Bsp:  $\{age: int, geburtstag: int \rightarrow int\}$

Schreibweise für konkrete Objekte:

$$o = \{age = 42, geburtstag = \lambda x. x + 1\}$$

$\{\}$ -Introduktionsregel:

$$\frac{e_1: \tau_1 \quad \dots \quad e_n: \tau_n}{\{m_1 = e_1, \dots, m_n = e_n\}: \{m_1: \tau_1, \dots, m_n: \tau_n\}}$$

$\{\}$ -Eliminationsregel:

$$\frac{x = \{m_1 = e_1, \dots, m_n = e_n\}: \{m_1: \tau_1, \dots, m_n: \tau_n\}}{\forall i = 1..n : e_i: \tau_i}$$

- Methodenaufruf:

$$\frac{o : \{\dots, m: \tau \rightarrow \tau', \dots\} \quad x: \tau}{o.m(x): \tau'}$$

## 16.1 Typkonversionen

- Typkonversion ( $\sigma$  ist Unterklasse von  $\tau$ ):  $\sigma \leq \tau$   
 “Jedes  $\sigma$ -Objekt ist auch ein  $\tau$ -Objekt” (Typ-Konformanz)

Ist Halbordnung:

$$\tau \leq \tau \text{ und } \tau \leq \tau' \wedge \tau' \leq \tau'' \Rightarrow \tau \leq \tau'' \text{ und } \tau \leq \tau' \wedge \tau' \leq \tau \Rightarrow \tau = \tau'$$

entspricht Upcast in Java/C++

- Konversion von elementaren Typen: z.B.  
 $int \leq real$  (Pascal: “jeder Integer ist auch ein Real”)  
 (Achtung: keine Verhaltens-Konformanz wg. Division! Zwar gilt  $real(x +_{int} y) = real(x) +_{real} real(y)$ , aber nicht  $real(x \text{ div } y) = real(x) / real(y)$ ;  
 also auch nicht  $Post(\text{div}) \Rightarrow Post(/)$ )

Manche Sprachen haben solche elementaren Konvertierungen, andere aus Sicherheitsgründen (Verhaltens-Konformanz) nicht. In jedem Fall komplexes Zusammenspiel mit Überladungen!

Gegenbeispiel:  $int_{32} \leq int_{16}$

- Objektkonversion (O1):

$$\frac{\sigma_1 \leq \tau_1 \quad \dots \quad \sigma_n \leq \tau_n}{\{m_1: \sigma_1, \dots, m_n: \sigma_n\} \leq \{m_1: \tau_1, \dots, m_n: \tau_n\}}$$

Objekterweiterung (O2):

$$\begin{aligned} & \{m_1: \tau_1, \dots, m_n: \tau_n, m_{n+1}: \tau_{n+1}, \dots, m_{n+k}: \tau_{n+k}\} \\ & \leq \{m_1: \tau_1, \dots, m_n: \tau_n\} \end{aligned}$$

kann in 1 Regel zusammengefaßt werden.

Bsp:

$$\{a: int, b: int, c: bool\} \leq \{a: int, b: real\}$$

$\tau \leq \tau'$ : Jedes  $\tau$ -Objekt kann auch als  $\tau'$ -Objekt verwendet werden (Typkonformanz)

in Java nur O2, dh  $\sigma_i = \tau_i$  wg Subobjekt-Implementierung

## 16.2 Kontravarianz

Ziel: Verallgemeinerung von  $\leq$  auf Methoden

Motiv:

1. Simulation einer Funktion durch eine andere
2. Modifikation der Methodensignatur in Unterklassen-Redefinitionen; Grund: mehr Flexibilität

Beide Motive führen auf die sog. Kontravarianz

Zu Motiv 1:

Idee von Cardelli:  $f \leq g$  genau dann wenn  $g$  nur durch  $f$  sowie Upcasts simuliert werden kann

Beispiel:

$$\begin{aligned} f &= \lambda x. \text{trunc}(x) +_{int} 1 : real \rightarrow int \\ g &= \lambda x. \text{real}(x) +_{real} 1.0 : int \rightarrow real \end{aligned}$$

Angenommen, ich habe  $f$ , brauche aber  $g$ . Wg  $int \leq real$  kann  $x : int$  immer verlustfrei nach  $real$  konvertiert werden (Annahme: ideale Rechnerarithmetik), umgekehrt nicht:

$$f \leq g \Leftrightarrow \forall x \in int : g(x) = real(f(real(x)))$$

Beweis: Sei  $x \in int$ . Dann ist

$$\begin{aligned} real(f(real(x))) &= real(\text{trunc}(real(x)) +_{int} 1) = \\ &real(x +_{int} 1) = real(x) +_{real} 1.0 \end{aligned}$$

kompaktere Schreibweise:  $g = (real) \circ f \circ (real)$

Mithin: “ $f$  kann auch als  $g$  verwendet werden” oder “ $f$  kann  $g$  simulieren” oder “ $f$  ist in  $g$  konvertierbar”:  $f \leq g$

Das umgekehrte gilt nicht!

Konvertierungsprozeß: habe  $f$ , brauche  $g$

1. Konvertiere  $g$ 's Argument nach  $real$
2. Wende  $f$  darauf an
3. Konvertiere Ergebnis nach  $real$

Gegenbeispiel für Konvertierbarkeit:

$$\begin{aligned} f &= \lambda x.x + 1 : int \rightarrow int \\ g &= \lambda x.x + 1.0 : real \rightarrow real \end{aligned}$$

Denn  $\exists x \in real : g(x) \neq real(f(int(x)))$  zB für  $x = 3.14$

$\Rightarrow$  konvertierbare Funktionen sind *kontravariant*:

$$\frac{\sigma \leq \sigma' \quad \tau' \leq \tau}{\sigma' \rightarrow \tau' \leq \sigma \rightarrow \tau}$$

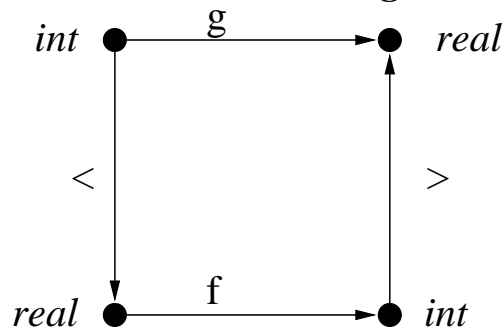
Falls also  $f : \sigma' \rightarrow \tau'$ ,  $g : \sigma \rightarrow \tau$ , so  $f \leq g$

im obigen Beispiel ist  $f : real \rightarrow int$ ,  $g : int \rightarrow real$  also

$$\frac{int \leq real \quad int \leq real}{real \rightarrow int \leq int \rightarrow real}$$

und deshalb  $f \leq g$

Darstellung als kommutierendes Diagramm:



in Wirklichkeit weniger für Basistypen als für Klassentypen verwendet

entsprechendes Beispiel:  $f: \sigma' \rightarrow \tau'$ ,  $g: \sigma \rightarrow \tau$

$$\sigma' = \{a : int, b : int, c : bool\}, \sigma = \{a : int, b : int\}$$

$$\tau = \{u : int, v : real, w : bool\}, \tau' = \{u : int, w : bool\}$$

Es ist  $\sigma' \leq \sigma$ ,  $\tau \leq \tau'$ , also  $\sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'$

$\Rightarrow g$  kann verlustfrei auf Typ  $\sigma' \rightarrow \tau'$  „geliftet“ werden:

$$f = (\tau') \circ g \circ (\sigma) : \sigma' \rightarrow \tau', \quad g \leq f$$

Hingegen ist *nicht*  $f \leq g$ , denn das umgekehrte  $(\tau) \circ g \circ (\sigma')$  kann abstürzen (unsichere Downcasts!)

*Funktionen sind nur auf kontravariante Weise konvertierbar!*



### 16.3 Kontravarianz und dynamische Bindung

Wird Methode redefiniert, so darf eine eventuelle Signaturänderung nur kontravariant erfolgen! Bsp:

```

class A {...}
class B extends A {...}
class O {
  A f(B x){...};
  B g(A x){...};
}
class U extends O {
  B f(A x){...}; // Kontravarianz, in Java nicht moeglich
  A g(B x){...}; // Kovarianz, in Java nicht moeglich
}
A a; B b;
O x = new U();
a = x.f(b); // OK
b = x.g(a); // illegaler Downcast

```

Wir haben  $O :: f : B \rightarrow A$ ,  $U :: f : A \rightarrow B$ , ergo

$$\frac{B \leq A \quad B \leq A}{A \rightarrow B \leq B \rightarrow A} \text{ mithin } U :: f \leq O :: f$$

für  $g$  ist die Sache genau andersrum  $\Rightarrow$  Absturz!

$\Rightarrow$  Verhaltenskonformanz erzwingt Kontravarianz!

Bem. In Java gibt es das überhaupt nicht, wg. Interferenz mit Überladungen/Arrays. in C++ nur für Ergebnistypen.

## 16.4 Typkonstruktoren

Typkonstruktor: dient zur Beschreibung eines komplexen Typs, der aus Komponenten zusammengesetzt ist

kann als „Funktion“ aufgefaßt werden, die aus gegebenen Typen neue konstruiert

typische Beispiele:

$$\begin{array}{ll}
 list : \mathcal{TYP}E \rightsquigarrow \mathcal{TYP}E & \tau \mapsto list(\tau) \\
 pair : \mathcal{TYP}E \times \mathcal{TYP}E \rightsquigarrow \mathcal{TYP}E & \tau_1, \tau_2 \mapsto pair(\tau_1, \tau_2) \\
 array : \mathcal{TYP}E \times Int \times Int \rightsquigarrow \mathcal{TYP}E & \tau, n, m \mapsto \\
 & array(n, m, \tau) \\
 & \text{oder kurz} \\
 & array(\tau) \\
 \rightarrow : \mathcal{TYP}E \times \mathcal{TYP}E \rightsquigarrow \mathcal{TYP}E & \tau, \tau' \mapsto (\tau \rightarrow \tau')
 \end{array}$$

Bem. Typkonstruktoren sind *homogen*: alle Listen/Arrayelemente müssen vom selben Typ sein

Wie ist es mit Vererbung bei Typkonstruktoren?

Wenn  $\tau \leq \tau'$ , ist dann  $list(\tau) \leq list(\tau')$  oder

$list(\tau') \leq list(\tau)$ ?

Wie ist es mit Arrays? Gibt es einen Unterschied zwischen Arrays und Listen?

## 16.5 Die Array-Anomalie in JAVA

Array-Konstruktoren in JAVA sind nicht kontravariant: Falls  $\tau \leq \tau'$ , so ist lt. Sprachdefinition

$array(\tau) \leq array(\tau')$  (Kovarianz!)

Grund: sonst keine „generischen“ Array-Bibliotheksfunktionen möglich (z.B. arraycopy)

Hintergrund: Zuweisung kopiert nur Array-Referenz. Beispiel:

```

class Student extends Person {...}
...
Person[] pa = new Person[17];
Student[] sa = new Student[42];
pa = sa; // erlaubt nach Sprachdef:
         // pa ist nun Student-Array
pa[1] = new Person(); // Laufzeit-Typfehler!!
                       // ArrayStoreException

```

Laufzeit: Downcast *Person*  $\rightarrow$  *Student* geht schief!

wäre hingegen  $Person[] \leq Student[]$ , so wäre Zuweisung  $pa = sa$ ; verboten  $\Rightarrow$  Laufzeitfehlerfreiheit

```
for i=0; i<pa.length; i++ pa[i] = sa[i];
```

wäre weiterhin erlaubt (aber dann ist natürlich  $pa[i]$  eine *Person* und kein *Student*)

Jedoch:  $array(\tau') \leq array(\tau)$  funktioniert auch nicht ...

Zusammenhang mit Funktions-Kontravarianz:

Stellen wir uns vor, Arrayzugriff würde intern durch (Bibliotheks)funktion  $s$  implementiert:

```
class B { B[] t;
        void s(int i, B x) { t[i]=x; }
      }
class A extends B { A[] t;
        void s(int i, A x) { t[i]=x; }
      }
```

Hier soll  $array(A) \leq array(B)$  wie in Java gelten

$\Rightarrow$  wg. Kontravarianz im 2. Argument von  $s: B \leq A$ , insgesamt also  $A = B$ , was im Beispiel Unsinn ist

Ergo:

$$array(\tau) \leq array(\tau') \Leftrightarrow \tau = \tau'$$

Dies gilt für *alle* Typkonstruktoren, bei denen Komponenten durch Zuweisung geändert werden können! (Cardelli)

Hingegen können Typkonstruktoren, die keine Modifikation von Komponenten anbieten, kovariant definiert werden (wie in Java)