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 ≃ Klassen, nebst elementaren Typen und Vererbung
- Typsysteme verhindern *illegale Downcasts*

Cardelli-Kalkül:

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

  Bsp: $f : real \to bool$
  Schreibweise: $f = \lambda x.E : \sigma \to \tau$

  →-Introduktionsregel:

$$\frac{E : \tau \qquad x : \sigma}{\lambda x.E : \sigma \to \tau}$$

  →-Eliminationsregel:

$$\frac{f : \sigma \to \tau \qquad x : \sigma}{f(x) : \tau}$$

  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:

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

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

Schreibweise für konkrete Objekte:
$o = \{age = 42,\ geburtstag = \lambda x.x + 1\}$

{}-Introduktionsregel:

$$\frac{e_1 : \tau_1 \qquad \ldots \qquad e_n : \tau_n}{\{m_1 = e_1, \ldots, m_n = e_n\} : \{m_1 : \tau_1, \ldots, m_n : \tau_n\}}$$

{}-Eliminationsregel:

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

- Methodenaufruf:

$$\frac{o : \{\ldots, m : \tau \to \tau', \ldots\} \qquad x : \tau}{o.m(x) : \tau'}$$

16.1 Typkonversionen

- Typkonversion (σ ist Unterklasse von τ): $\sigma \leq \tau$

  "Jedes σ-Objekt ist auch ein τ-Objekt" (Typ-Konformanz)

  Ist Halbordnung:

  $\tau \leq \tau$ und $\tau \leq \tau' \wedge \tau' \leq \tau'' \Rightarrow \tau \leq \tau''$ 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: $\text{int}_{32} \leq \text{int}_{16}$

- Objektkonversion (O1):

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

  Objekterweiterung (O2):

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

  kann in 1 Regel zusammengefaßt werden.

  Bsp:

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

  $\tau \leq \tau'$: Jedes τ-Objekt kann auch als τ'-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:

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

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:

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

Beweis: Sei $x \in int$. Dann ist
$real(f(real(x))) = real(trunc(real(x)) +_{int} 1) =$
$real(x +_{int} 1) = real(x) +_{real} 1.0$

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:

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

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

$\Rightarrow$ konvertierbare Funktionen sind *kontravariant*:

$$\frac{\sigma \leq \sigma' \qquad \tau' \leq \tau}{\sigma' \to \tau' \ \leq \ \sigma \to \tau}$$

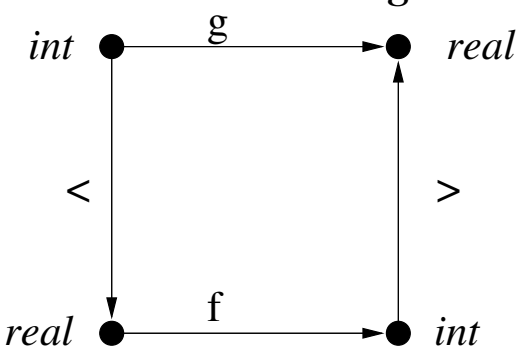
Falls also $f : \sigma' \to \tau', \ \ g : \sigma \to \tau$, so $f \leq g$

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

$$\frac{int \leq real \qquad int \leq real}{real \to int \ \leq \ int \to real}$$

und deshalb $f \leq g$

Darstellung als kommutierendes Diagramm:



in Wirklichkeit weniger für Basistypen als für Klassentypen verwendet

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

$$\sigma' = \{a : int, b : int, c : bool\},\ \sigma = \{a : int, b : int\}$$
$$\tau = \{u : int, v : real, w : bool\},\ \tau' = \{u : int, w : bool\}$$

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

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

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

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 $O :: f : B \to A$, $U :: f : A \to B$, ergo

$$\frac{B \leq A \qquad B \leq A}{A \to B \leq B \to A} \text{ mithin } U :: f \leq O :: f$$

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:

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

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[]`≤`Student[]`, so wäre Zuweisung `pa = sa;` verboten ⇒ 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)

Beispiel für Anomalie bei zuweisbaren Unterkomponenten (Annahme: dynamische Bindung auch für Instanzvariablen)

```
class A extends B { ... }

class O { A x; }
class U extends O { B x; }

O o = new U();
A a = o.x;  // illegal downcast

class R { B x; }
class S extends R { A x; }

R r = new  S();
r.x = new B(); // illegal downcast
```

⇒ redefinierte Instanzvariablen müssen stets denselben Typ haben wie in der Oberklasse

$U \leq O \Leftrightarrow A = B$

Das Problem tritt sogar ohne dynamische Bindung von Instanzvariablen auf, wenn man in diesem Beispiel generische Klassen verwendet. S.u.