

Programmiersprachen
Vorlesung 8 vom 27.11.23
Fortgeschrittene Typsysteme

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Organisatorisches: Vorträge

08.01.2024	Fabian: Abap	Kenneth: COBOL	Carolin: SQL
11.01.2024	Corinna: VHDL	Matthias: Verilog	Mika: Lua
15.01.2024	Moritz: Postscript	Niklas: Julian	Aljoscha: Kotlin
18.01.2024	Niklas: Swift	Finn B: Dart	Daniel: Golang
22.01.2024	Simeon: Elixir	Raphael: Uiua	Yannic: Liquid X
29.01.2024	Kelke: Mojo	Johannes & Finn: cQASM	

Wo sind wir?

- ▶ Einführung
- ▶ Einfache Ausdrücke und Werte
- ▶ Einfache Typen, Anweisungen und Seiteneffekte
- ▶ Variablen und Speichermodelle
- ▶ Aggregierende Typen
- ▶ Ausnahmen und Fehlerbehandlung
- ▶ Prozeduren und Funktionen
- ▶ Fortgeschrittene Typsysteme
- ▶ Datenabstraktion
- ▶ Programmierparadigmen
- ▶ Skriptsprachen
- ▶ Ab Woche 11 (2024): Studentische Vorträge.

Typäquivalenz

- ▶ Grundsätzliche Frage: wann ist $T_1 \cong T_2$? (Wichtig für die Typüberprüfung)
- ▶ **Nominal**: wenn sie den gleichen Namen haben.
- ▶ **Strukturell**: wenn sie die gleiche Struktur haben.

```
typedef struct a {int a;  
                  double d;  
                  } t1;  
  
void f(t1 x) {  
    printf("Double is %f\n", x.d);  
}
```

```
typedef struct b {int a;  
                  double d;  
                  } t2;  
  
void g(t2 y) {  
    f(y);  
}
```

Typäquivalenz konkret

- ▶ C, Rust¹, Haskell, Java nutzen nominale Typäquivalenz
 - ▶ `typedef` in C, `type` in Haskell sind **Typsynonyme**, definiert keinen neuen Typ
- ▶ Python nutzt (schwächere Form der) strukturellen Typäquivalenz
 - ▶ “Duck typing”

Polymorphie

- ▶ Von griechisch $\pi ο λ υ ζ$ (viel), $μ ο ρ φ η$ (Gestalt)
- ▶ Ganz allgemein: Funktionen (Methoden), die auf **mehr als einen** Typ anwendbar sind.
- ▶ Im speziellen:
 - ▶ In **objektorientierten Sprachen**: jede Methode ist auch auf allen Untertypen anwendbar.
 - ▶ **Parametrische Polymorphie**: nach Hindley-Milner, uniform auf **allen** Typen definiert.
 - ▶ **Ad-Hoc Polymorphie**: Überladene Funktionen, auf **einigen** Typen spezifisch definiert.

Polymorphie in Java und Python

- ▶ Methode `f` einer Klasse kann auf allen Untertypen angewandt werden.
- ▶ Klasse des Objektes bestimmt konkrete Methode (**dynamische Bindung**)

```
class C:
    def f(self):
        print("Foo.")
    def g(self):
        print("Baz.")

class D(C):
    def f(self):
        print("Wibble.")
```

Parametrische Polymorphie:

- ▶ Parametrische Polymorphie: Abstraktion über **Typen**
- ▶ Sowohl für Funktionen (Methoden) als auch für Datentypen (Klassen)
- ▶ Beispiel: Listen
 - ▶ Haben für alle Typen die gleiche Struktur
 - ▶ Viele Funktionen auf Listen sind vom Inhalt der Listen unabhängig.
- ▶ Typisierung nach dem Hindley-Milner-Damas-Algorithmus

Parametrische Polymorphie in Java: Generics

▶ Beispiel: Listen

```
class List<T> {  
    public T elem;  
    public List<T> next;  
  
    public List(T el, List<T> tl) {  
        this.elem= el;  
        this.next= tl;  
    }  
}
```

▶ Benutzung umständlich weil Java keine Typen inferiert:

```
List<Integer> l1= new List<>(1, new List<>(2, null));
```

Parametrische Polymorphie in Haskell

- ▶ Typ-Parameter, an Funktionen oder Typen:

```
data List a = Cons a (List a) | Null

map :: (a → b) → List a → List b
map f Null = Null
map f (Cons a l) = Cons (f a) (map f l)
```

- ▶ Haskell leitet Typen ab, vergleicht dann (ggf.) mit deklarierten Typen
- ▶ Elegante Benutzung

Parametrische Polymorphie in Rust

- ▶ Mehr Java als Haskell

- ▶ Beispiel: Option

```
enum Option<T> {  
    Some(T),  
    None,  
}
```

- ▶ Muss an Funktionen **explizit** annotiert werden:

```
fn just<T>(opt: &Option<T>) -> &T {  
    match opt { Some(t) => t }  
}
```

Parametrische Polymorphie in C

- ▶ Der Typ `void *` ist mit `t *` für alle Typen `t` kompatibel.
 - ▶ C-Standard (C-90), 6.3.2.3 Pointers:
A pointer to void may be converted to or from a pointer to any incomplete or object type.
- ▶ Manuelle Typannotation nötig — Typinformation geht verloren (bspw. für `map` und `filter`)
- ▶ Funktionen höherer Ordnung durch Zeiger auf Funktionen.
- ▶ Vergleiche Beispiel.

Theoretische Aspekte:

- ▶ Parametrische Polymorphie ist **entscheidbar** (Hindley-Milner-Damas).
 - ▶ Mit exponentiellem Aufwand.
 - ▶ In der Praxis unerheblich.
- ▶ Wird schnell unentscheidbar:
 - ▶ Konstruktorklassen
 - ▶ Rank-2 Polymorphie
 - ▶ Subtyping

Ad-Hoc-Polymorphie und Überladen

- ▶ **Ad-Hoc-Polymorphie** ist ein Aspekt von **Überladung**: ein Bezeichner, mehrere Funktionen.
 - ▶ Beispiel für Ad-hoc-Polymorphie: Addition + auf verschiedenen numerischen Typen.
 - ▶ Beispiel für Überladen: - unär für Negation, binäre für Subtraktion
- ▶ **Java** erlaubt Überladen, Ad-Hoc-Polymorphie über Interfaces
- ▶ **Rust** erlaubt kein Überladen, Ad-Hoc-Polymorphie über Traits
 - ▶ Gleicher Bezeichner mit mehreren, aber **unterschiedlichen** Signaturen
- ▶ **C** hat überladene numerische Operatoren (+, -, *) und erlaubt sonst **kein** Überladen
- ▶ **Python** erlaubt kein Überladen (hat aber default-Parameter, Traits (Mix-Ins), u.ä.)

Ad-Hoc-Polymorphie in Haskell

- ▶ Haskell hat Typklassen:

```
class Eq a where
  (==) :: a -> a -> Bool
class Eq a => Num a where
  (+) :: a -> a -> a
instance Num Int where
  a + b = ...
```

- ▶ Ansonsten kein Überladen
- ▶ Typklassen für Datentypen (Konstruktorklassen)

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Untertypen (Subtyping)

- ▶ Semantisch ist S ein **Untertyp** von T gdw.

$$S \subseteq T$$

- ▶ Wo immer T gefordert ist, kann auch S benutzt werden.
- ▶ Beispiel numerische Typen

$$\mathbb{N} \subseteq \mathbb{Z} \subset \mathbb{R} \quad (1)$$

- ▶ Siehe Typkonversionen in C: `unsigned int`, `int`, `double` sowie verschiedenen Wortbreiten
- ▶ Verallgemeinert: S ist Untertyp von T wenn es eine **Einbettung** gibt:

$$\iota : S \hookrightarrow T \quad \iota \text{ injektiv}$$

- ▶ Einbettung: **explizite** Konversion
- ▶ Beispiel numerische Typen in Haskell

Record Subtyping

- ▶ Vererbung erzeugt semantisch gesehen keine Subtypen

```
class Point {  
    double x;  
    double y;  
}
```

```
class ColouredPoint  
    extends Point {  
    Colour col;  
}
```

- ▶ `Point` ist $\mathbb{R} \times \mathbb{R}$, `ColouredPoint` ist $\mathbb{R} \times \mathbb{R} \times \textit{Colour}$
- ▶ Wir können aus jedem `ColouredPoint` einen `Point` machen
 - ▶ Allerdings nicht injektiv
 - ▶ Solange wir **nur** auf die **Felder** zugreifen
- ▶ Deshalb können wir `ColouredPoint` als Untertyp von `Point` **definieren** (“record subtyping” oder “structural subtyping”)

Inheritance is not Subtyping

- ▶ Record Subtyping funktioniert nur bei Argumenten:

```
void move(Point p) {  
    this.x += p.x;  
    this.y += p.y;  
}
```

- ▶ Kann auch auf ColouredPoints angewandt werden.

```
static Point  
    fromPolar(double phi, double r) {  
        return new Point(r*Math.cos(phi),  
                          r* Math.sin(phi));  
    }  
Point move1(Point p) {  
    return new Point(this.x+ p.x,  
                    this.y+ p.y);  
}
```

- ▶ Welche Farbe soll der Ergebnistyp haben?
- ▶ Wenn $A \subseteq A'$, dann ist $A \rightarrow B \subseteq A' \rightarrow B$, aber $B \rightarrow A \not\subseteq B \rightarrow A'$

Subtyping und Parametrische Polymorphie (Generics)

- ▶ Frage: sind Typkonstruktoren F **monoton**

$$A \subseteq B \implies F(A) \subseteq F(B)?$$

- ▶ Ganz allgemein gilt:

$$A \subseteq A' \implies \begin{array}{l} A \times B \subseteq A' \times B \\ B \times A \subseteq B \times A' \\ A + B \subseteq A' + B \\ B + A \subseteq B + A' \\ A' \rightarrow B \subseteq A \rightarrow B \\ B \rightarrow A \subseteq B \rightarrow A' \end{array}$$

- ▶ **Polynomiale** Typkonstruktoren sind Datentypen aus Produkt und Koprodukt (vgl. `data` in Haskell ohne Funktionsräume)
- ▶ Polynomiale Typkonstruktoren sind monoton.
- ▶ Funktionsräume $A \rightarrow B$ machen den Unterschied.

Kovarianz und Kontravarianz

- ▶ Ein Typkonstruktor $F(X)$ ist **kovariant**, wenn X nur in **Argumentposition** des Funktionsraums \rightarrow auftaucht.
- ▶ Ein Typkonstruktor ist $F(X)$ ist kontravariant, wenn X nur in **Resultatposition** des Funktionsraums \rightarrow auftaucht.
- ▶ Wenn F **kovariant**, dann ist F monoton: $A \subseteq B \implies F(A) \subseteq F(B)$
- ▶ Wenn F **kontravariant**, dann ist F anti-monoton: $A \subseteq B \implies F(B) \subseteq F(A)$

Subtyping und Generics

- ▶ Praktische Auswirkungen — ein klassisches Beispiel:

```
class Ref<T> {  
    private T curr;  
    Ref(T init) { this.curr= init; }  
  
    T get() { return curr; }  
    void set(T x) { curr= x; }  
}
```

- ▶ Consider this:

```
Ref<String> c1 = new Ref<>("foo");  
Ref<Object> c2 = c1; // String ⊆ Object, also Ref<String> ⊆ Ref<Object>  
c2.set(1); // Ändert auch c1  
String s = c1.get(); // Liest c1, wo aber jetzt eine Zahl steht... ⚡
```

- ▶ Problem ist zweite Zeile, $\text{Ref}\langle\text{String}\rangle \not\subseteq \text{Ref}\langle\text{Object}\rangle$

Arrays und Generics

- ▶ Lösung: Generics sind **invariant** (Typkonstrukturen nicht monoton)

- ▶ Typ `<? extends T>`, `<? super T>`

- ▶ Arrays sind **nicht** invariant:

```
String[] c1 = { "foo" };  
Object[] c2 = c1;  
c2[0] = 99;  
String s = c1[0];  
System.out.println(s); // What will happen?
```

Arrays und Generics

- ▶ Lösung: Generics sind **invariant** (Typkonstruktoren nicht monoton)

- ▶ Typ `<? extends T>`, `<? super T>`

- ▶ Arrays sind **nicht** invariant:

```
String[] c1 = { "foo" };  
Object[] c2 = c1;  
c2[0] = 99;  
String s = c1[0];  
System.out.println(s); // What will happen?
```

- ▶ Grund: Arrays sind **reifizierbar** — Typ wird zur Laufzeit **nicht** gelöscht.
- ▶ Bei Generics wird der Typ zur Laufzeit gelöscht (type erasure) — effizientere Ausführung.

Subtyping, Overloading and Dynamic Binding

- ▶ Java kombiniert Subtyping, Overloading und dynamisches Binden.
- ▶ Im ersten Argument (Methodenauswahl) wird die Methode **dynamisch** ausgewählt.
- ▶ Ansonsten wird der Typ **statisch** bestimmt.
- ▶ **Overloading** wird **statisch** aufgelöst.
- ▶ Siehe Beispiel.

Abhängige Typen (Dependent Types)

- ▶ Abhängige Typen vermischen Typen und Terme
- ▶ Typen können mit Termen gebildet werden
- ▶ Beispiel (Idris): `Vec l` sind Listen der Länge `l :: Int`

```
data Vect : Nat -> Type -> Type where  
Nil    : Vect Z a  
(::)   : a -> Vect k a -> Vect (S k) a  
  
(++)   : Vect n a -> Vect m a -> Vect (n + m) a  
Nil ++ ys    = ys  
(x :: xs) ++ ys = x :: xs ++ ys
```

- ▶ Nicht mehr entscheidbar.
- ▶ Erlaubt Kodierung von **Spezifikation** im Typ.

Abhängige Paare

- ▶ Einfach wenn die Länge statisch berechenbar ist:

```
map : (a -> b) -> Vect n a -> Vect n b
map f []           = []
map f (x :: xs) = f x :: map f xs
```

- ▶ Was ist, wenn wir die Länge nicht kennen:

```
filter : (a -> Bool) -> Vect n a -> (m ** Vect m a)
filter p Nil = ( _ ** [] )
filter p (x :: xs) with (filter p xs)
  | ( _ ** xs' ) = if (p x) then ( _ ** x :: xs' ) else ( _ ** xs' )
```

- ▶ Typ gegeben als abhängiges Paar:

```
data DPair : (a : Type) -> (P : a -> Type) -> Type where
  MkDPair : {P : a -> Type} -> (x : a) -> P x -> DPair a P
```

Zusammenfassung

- ▶ Fortgeschrittene Aspekte der Typsysteme
- ▶ Arten der Polymorphie
 - ▶ Polymorphie über Subtypen
 - ▶ Java und Python
 - ▶ Parametrische Polymorphie
 - ▶ In Haskell, Java (Generics); rudimentär in C (`void *`)
 - ▶ Ad-Hoc-Polymorphie und Overloading
 - ▶ Overloading in Java, Ad-Hoc-Polymorphie in Haskell
- ▶ Subtyping
 - ▶ Record Subtyping in Java, Python.
 - ▶ Kombination mit parametrischer Polymorphie delikat.