

Programmiersprachen
Vorlesung 2 vom 25.10.21
Werte und Typen

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Wo sind wir?

- ▶ Einführung
- ▶ Werte und Typen
- ▶ Anweisungen, Variablen und Zustand
- ▶ Kontrollabstraktion
- ▶ Datenabstraktion
- ▶ Fortgeschrittene Typsysteme
- ▶ Nebenläufigkeit
- ▶ Objektorientierung
- ▶ Skriptsprachen
- ▶ Beispielsprache II
- ▶ Ab Woche 11: Studentische Vorträge.

Ausdrücke und Anweisungen

- ▶ Viele Sprachen unterscheiden **Ausdrücke** und **Anweisungen**
 - ▶ Ausdrücke bezeichnen die zu manipulierenden **Werte**
 - ▶ Anweisungen beschreiben **Kontrollfluß**
 - ▶ **Imperatives** Konstrukt: Programm ist abstrakte Sicht auf Speicher manipulation
- ▶ Funktionale Sprachen, logische Sprachen uvm. haben diese Unterscheidung **nicht**.

Werte

- ▶ Was sind Werte?
“Any entity that can be manipulated by a program.”
- ▶ **Primitive** Werte:
Booleans, ganze Zahlen, Fließkommazahlen, Adressen (Pointer, Referenzen)
- ▶ **Zusammengesetzte** Werte (composite values):
Strukturen, Felder, Listen, Objekte, ...
- ▶ Semantisch gesehen:
 - ▶ Abstraktionen über dem Speicherinhalt
 - ▶ Nichtreduzierbare Ausdrücke

Typen

Typen

- ▶ Was sind Typen?
 - ▶ “A set of values.” (Mengen von Werten)
 - ▶ Durch die Operationen darauf charakterisiert. z.B. {13, *Monday*, *true*} ist kein Typ.
— kann man drüber streiten
- ▶ Arten von Typen:
 - ▶ **Primitive** Typen (primitive types)
 - ▶ **Zusammengesetzte** Typen (composite types)
- ▶ Typüberprüfung und Typsysteme

Vorgegebene Primitive Typen

- ▶ Anwendungsabhängig: COBOL vs. FORTRAN; BCPL vs. C vs. T_EX
- ▶ Ganze Zahlen:
 - ▶ Java: `int`
 - ▶ C: `char`, `short`, `int`, `long`, `long long`, etc.
 - ▶ Haskell: `Int`, `Natural`, `Integer`
- ▶ Booleans:
 - ▶ In C **kein** expliziter Typ (`bool_t` ist `int`)
- ▶ Fließkommazahlen: `float` und `double`

Frage 2.1: Welche Wortbreite haben ganze Zahlen in C, Java, Python, Haskell?

Selbstdefinierte Primitive Typen

- ▶ C, Java, Haskell erlauben **Aufzählungen**:

```
enum colour {red, green, blue}
```

- ▶ In C nur syntaktischer Zucker für `int`, kein separater Typ.
- ▶ In Java: Klasse mit konstanter Anzahl von Objekten (s. hier)
- ▶ In Haskell: algebraischer Typ mit ausschließlich konstanten Konstruktoren.

```
data Colour = Red | Green | Blue
```

- ▶ Ada erlaubt Untermengen von `int`:

```
type Cent is range 0 .. 99;
```

Zusammengesetzte Typen

- ▶ Cartesische Produkte (Tupel und Strukturen)
- ▶ Endliche Abbildungen (Arrays, Dictionaries)
- ▶ Disjunkte Vereinigung (algebraische Typen, discriminated records, Objekte)
- ▶ Rekursive Typen

Cartesische Produkte

▶ Für zwei Typen A und B sind **Tupel** der Typ $A \times B$ mit folgenden **Eigenschaften**:

① Es gibt es zwei Funktionen $\pi_1 : A \times B \rightarrow A, \pi_2 : A \times B \rightarrow B$.

② Für zwei Funktionen $f : C \rightarrow A, g : C \rightarrow B$ gibt es **genau eine** Funktion $\langle f, g \rangle : C \rightarrow A \times B$ so dass $\pi_1 \cdot \langle f, g \rangle = f$ und $\pi_2 \cdot \langle f, g \rangle = g$

▶ Verallgemeinerung auf n Komponenten:

$$A_1, \dots, A_n \quad \text{und} \quad \prod_{i=1, \dots, n} A_i$$

▶ Sonderfall: Tupel der Länge 0

$$T = 1$$

▶ Unit-Typ, in vielen Sprachen nur implizit (Java, C).

Cartesische Produkte

- ▶ Fast alle Programmiersprachen haben cartesische Produkte
- ▶ Entweder direkt als Tupel (Haskell, Scala) oder
- ▶ `struct` sind cartesische Produkte mit **benannten** Projektionen

```
struct pair {  
    int fst;  
    int snd;  
}
```

- ▶ Tupel sind “Listen fester Länge”

Frage 2.2: Wieso gibt es dann keine Funktion, um diese Listen aneinanderzuhängen?

Tupel und Funktionen

- ▶ Ungenaue Notation bei Funktionsaufruf:

Haskell:

```
f1 :: Int → Int → Int  
f2 :: (Int, Int) → Int
```

f1 hat zwei Argumente, f2 eines (ein Tupel)

- ▶ Es gilt $A \times B \rightarrow C \cong A \rightarrow B \rightarrow C$ ("Currying")
- ▶ Wird nicht von allen Programmiersprachen unterstützt
- ▶ $A \rightarrow B \rightarrow C$ ist eine Funktion höherer Ordnung

C:

```
int f(int x, int y)
```

f hat **zwei** Argumente, nicht eins.

Endliche Abbildungen

- ▶ Eine endliche Abbildung ist eine Funktion $A \rightarrow B$ mit A endlich.
- ▶ Dargestellt als linkseindeutige und rechtstotale Relation

$$A \rightarrow B \subseteq A \times B$$

- ▶ Wenn $A = 0, \dots, n$ für $n \in \mathbb{N}$, dann ist $A \rightarrow B$ ein **Feld** (Array)
 - ▶ Felder können effizient als zusammenhängende Speicherbereiche implementiert werden
 - ▶ Indizierung ($a[i]$) sehr billig $O(1)$
 - ▶ Manchmal Indizierung auch ab 1 oder von $n \dots m$
 - ▶ Mehrdimensionale Felder durch Felder von Feldern (C, Java) oder Indizierung mit Tupeln (Haskell)
- ▶ Wenn A eine Menge von **Bezeichnern**, dann ist $A \rightarrow B$ ein **Dictionary** (Python)
 - ▶ Im Unterschied zu `structs` sind Dictionaries in Python zur Laufzeit definierbar und erweiterbar

Disjunkte Vereinigung

Für zwei Typen A und B ist die **disjunkte Vereinigung** der Typ $A + B$ mit folgenden **Eigenschaften**:

- 1 Es gibt zwei Funktionen $in_1 : A \rightarrow A + B$ und $in_2(b) : B \rightarrow A + B$.
- 2 Für zwei Funktionen $f : A \rightarrow C$, $g : B \rightarrow C$ gibt es **genau eine** Funktion $[f, g] : A + B \rightarrow C$ so dass $[f, g] \cdot in_1 = f$ und $[f, g] \cdot in_2 = g$.

Verallgemeinerung auf n Komponenten:

$$A_1, \dots, A_n \quad \text{und} \quad \coprod_{i=1, \dots, n} A_i$$

- ▶ Disjunkte Vereinigungen werden sehr **heterogen** gehandhabt.
- ▶ Sonderfall: leere Vereinigung — der **leere** Typ

$$T = \emptyset$$

- ▶ `void` in C und Java, `Nothing` in Scala; in Haskell nicht vordefiniert und nutzlos

Frage 2.3: Warum nutzlos? Warum ist 'void' in C eigentlich semantisch falsch?

Disjunkte Vereinigung in C

- ▶ Der Union-Typ vereinigt alle Komponenten an der gleichen Adresse:

```
union { int x; double y; } u;
```

- ▶ Extrem fehleranfällig
- ▶ Keine **disjunkte** Vereinigung
- ▶ Zur Unterscheidung (discriminated records in Ada):

```
enum u_tag { u_a, u_b };  
struct { enum u_tag tag; union { int a; double b; } cont; } u;  
  
switch (u.tag) {  
    case u_b: printf("Double: %f\n", u.cont.y); break  
}
```

Disjunkte Vereinigung in Java

- ▶ Sehr indirekt durch Subtyping:

```
class A {  
    C f() { ... };  
}  
class B {  
    C g() { ... };  
}
```

```
abstract class AB {  
    private class InlA extends AB {  
        private A a;  
        C fg() { a.f(); }  
    }  
    private class InrB extends AB {  
        private B b;  
        C fg() { b.g(); }  
    }  
    C fg();  
}
```

Disjunkte Vereinigung in Haskell

- ▶ Algebraische Datentypen (mit Fallunterscheidung):

```
data A
f  :: A → C
```

```
data B
g  :: B → C
```

```
data AB = Inl A | Inr B
```

```
fg  :: AB → C
```

```
fg ab = case ab of Inl a → f a; Inr b → g b
```

Rekursive Typen

- ▶ Rekursive Typen sind durch **Gleichungen** definiert:

$$T = F(T)$$

- ▶ Beispiele:

- ▶ Listen: $L(A) = 1 + A \times L(A)$
- ▶ Binäre Bäume: $T(A) = 1 + T(A) \times A \times T(A)$
- ▶ Variadische Bäume: $R(A) = A \times L(R(A))$

Frage 2.4: Wieso definieren diese Gleichungen den Typ?

Rekursive Typen in C

- ▶ C erlaubt **keine** direkt rekursiven Typen
 - ▶ Umweg über Zeiger und “incomplete types”:

```
typedef struct list_el {  
    void *head;  
    struct list_el *tail;  
} *list;
```

```
typedef struct tree_el {  
    struct tree_el *le; void *node; struct tree_el *ri;  
} *tree;
```

- ▶ Der NULL-Pointer übernimmt die Rolle des Unit-Typen.
- ▶ Polymorphie durch `void *`.

Rekursive Typen in Java

Rekursive Typen durch rekursive Klassen:

```
class List {  
    public Object head;  
    public List tail;  
}
```

```
class Tree {  
    public Tree left;  
    public Object node;  
    public Tree right;  
}
```

- ▶ In Java ist alles¹ **implizit** eine Referenz (Pointer), daher eigentlich ähnlich C einschließlich `null` für den Unit-Typ.
- ▶ Polymorphie durch `Object`, mehr dazu später.

¹Bis auf primitive Typen.

Rekursive Typen in Haskell

- ▶ Hier können wir die Domänengleichungen direkt abschreiben:

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

```
data Tree a = Node { le :: Tree a, node :: a, ri :: Tree a }
```

```
data NTree a = Node a (List (Ntree a))
```

- ▶ Listen sind mit syntaktischem Zucker vordefiniert.

Rekursive Typen in Python

- ▶ Listen sind vordefiniert (Typ `list`)
- ▶ Definition von rekursiven Typen nicht direkt möglich, nur als Klasse.
- ▶ Binäre Bäume:

```
class Tree:  
    def __init__(self, node):  
        self.left = None  
        self.right = None  
        self.node = node
```

```
class NTree:  
    def __init__(self, node):  
        self.node = node  
        self.children = []
```

- ▶ `__init__` ist der Konstruktor, der Typ selbst wird dynamisch definiert.

Zeichenketten

- ▶ Zeichenketten (Strings) spielen **meist** eine Sonderrolle
- ▶ Konzeptionell: Array oder Liste von Zeichenketten, e.g. C und Haskell:

```
char txt1 [] = "Foo";           type String = [Char]
char txt2 [4] = {'F', 'o', 'o', 0};
```

- ▶ Aus Effizienzgründen: **Unveränderliche** Strings
 - ▶ Java und Python
 - ▶ `bytestring` in Haskell

Typsysteme

Typsysteme

- ▶ Was gibt es für Typen?
- ▶ Wie überprüfen wir Typen?
 - ▶ Statisch vs. dynamisch
- ▶ Gleichheit von Typen
- ▶ Anforderung an ein Typsystem:
 - ▶ Entscheidbarkeit
 - ▶ Effizienz

Typüberprüfung

- ▶ **Statische** Typsysteme:
 - ▶ Typen werden zur Compile-Zeit geprüft.
 - ▶ Zur Laufzeit werden keine Typinformationen benötigt (“type erasure”)
 - ▶ Sicher und effizient, aber unflexibel
 - ▶ Bsp: C, Haskell, Java
- ▶ **Dynamische** Typsysteme
 - ▶ Typen werden zur Laufzeit geprüft.
 - ▶ Flexibel, aber fehleranfällig.
 - ▶ Bsp: Python, Java (dynamische Bindung von Methoden)

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, 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”

Implizit vs. explizit

- ▶ Explizit oder manifeste Typsysteme: alle Typen werden **explizit** angegeben
 - ▶ Compiler muss nur prüfen
 - ▶ Beispiel: Java
- ▶ Implizite Typsysteme: Typen werden abgeleitet
 - ▶ Compiler muss Typ inferieren
 - ▶ Beispiel: Haskell, Hindley-Milner-Typsystem
 - ▶ Problem: Entscheidbarkeit, bspw. mit Subtyping unentscheidbar

Frage 2.5: Ist Python implizit oder explizit?

Zusammenfassung

Zusammenfassung

- ▶ Werte sind ...
- ▶ Typen sind **Mengen von Werten**
- ▶ Primitive Typen: Zahlen, Zeichen
- ▶ Zusammengesetzte Typen:
 - ▶ Tupel, endliche Abbildungen, disjunkte Vereinigung
 - ▶ Rekursive Typen
- ▶ Typsysteme:
 - ▶ Statisch vs. dynamisch
 - ▶ Nominal vs. strukturell
 - ▶ Implizit vs. explizit