

Programmiersprachen  
Vorlesung 5 vom 05.11.23  
Aggregierende Typen

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

### 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.

### Aggregierende Typen

- ▶ Auch genannt: Compound Types, Aggregate Types, Composite Types
- ▶ Kartesische Produkte (Tupel und Strukturen)
- ▶ Endliche Abbildungen (Arrays, Dictionaries)
- ▶ Vereinigungstypen (algebraische Typen, discriminated records, Objekte)
- ▶ Rekursive Typen

### Kartesische Produkte

- ▶ Mathematisch:
  - ▶ Paare:  $A \times B = \{(a, b) \mid a \in A, b \in B\}$
  - ▶ n-Tupel:  $\prod_i A_i = \{(a_1, \dots, a_n) \mid a_i \in A_i\}$
- ▶ Fast alle Programmiersprachen haben kartesische Produkte
- ▶ Direkt als Tupel (Haskell, Scala),
- ▶ `struct` sind kartesische Produkte mit **benannten** Projektionen

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

- ▶ Tupel sind "Listen fester Länge"

### Tupel und Funktionen

- ▶ Ungenaue Notation bei Funktionsaufruf:

Haskell: `f1 :: Int -> Int -> Int`      C: `int f(int x, int y)`  
`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

### Endliche Abbildungen

- ▶ Eine endliche Abbildung ist eine Funktion  $A \rightarrow B$  mit  $A$  endlich.
- ▶ 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  $n \dots m$
- ▶ Wenn  $A$  eine Menge von **Bezeichnern**, dann ist  $A \rightarrow B$  ein **Dictionary** (Python)

```
a = { "foo" : 12, "baz" : "hello", "bah" : { "a" : 1, "b" : False } }
a.get("foo")
a["foo"]
a["bah"]["a"]
```

### Disjunkte Vereinigung

- ▶ Mathematisch:

- ▶ Binär:  $A + B = (\{1\} \times A \cup \{2\} \times B)$
- ▶ Verallgemeinert:  $\sum_i A_i = \cup_i \{i\} \times A_i$

- ▶ Disjunkte Vereinigungen werden sehr **heterogen** gehandhabt:

- ▶ C kennt **union**
- ▶ Meist kann man nicht einfach zwei Typen vereinigen.
- ▶ In Java über Subtyping.

### 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.b); break
}
```

## Disjunkte Vereinigung in Java

- ▶ Unterklassen einer Oberklasse sind eine Vereinigung (aber nicht disjunkt)
- ▶ 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 Either a b = Left a | Right b

foldEither :: (a -> c) -> (b -> c) -> Either a b -> c
foldEither l r (Left a) = l a
foldEither l r (Right b) = r b
```

- ▶ Unterschiedliche Konstruktoren für einen Typen sind eine disjunkte Vereinigung (wie Left, Right oben).

## Sonderfälle

- ▶ Leere Vereinigung — der **leere** Typ:  $T = \emptyset$
- ▶ void in C und Java, Nothing in Scala; in Haskell nicht vordefiniert und nutzlos
- ▶ Leeres Tupel – der **einelementige** Typ:  $T = ()$
- ▶ () in Haskell und Rust, Null in Scala

## 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:** 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 alles<sup>1</sup> **explizit** eine Referenz (Pointer), daher eigentlich ähnlich C einschließlich null für den Unit-Typ.
- ▶ Polymorphie durch Object, mehr dazu später.

<sup>1</sup>Bis auf primitive Typen.

## Rekursive Typen in Haskell

- ▶ Hier können wir die Domängleichungen 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 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.

## Variablen von aggregierendem Typ

- ▶ Variablen von aggregierendem Typ belegen einen Block von mehreren Speicherzellen
- ▶ Bei Feldern zusammenhängend
- ▶ Bei Tupeln nicht notwendigerweise
- ▶ Speicherlayout und alignment
  - ▶ Nur für systemnahe Programmiersprachen (e.g. C)
- ▶ Totales und selektives Update

```
struct date { int y, m, d; } d1, d2;
d1.m= 11; // selektiv
d2= d1; // total
```

- ▶ C erlaubt **Speicherarithmetik**: `a[i] == *(a+i)`

## Felder

- ▶ Statische Felder haben **feste, unveränderliche** Länge (C, Rust, Java)
- ▶ Bei **dynamischen** Felder kann die Länge verändert werden (Haskell, Vec in Rust, Vector in Java)
- ▶ Bei **flexiblen** Feldern ist die Länge variabel (aber fest)

```
double a1[] = {2.0, 3.0, 5.0};

static void prtVec(double [] v) {
    for (int i= 0; i< v.length; i++)
        System.out.println(v[i]+" ")
}
```

## Copy Semantics vs Reference Semantics

- ▶ Was passiert bei einer Zuweisung `x= e`, wenn `x` einen zusammengesetzten Typ hat?
- ▶ **Copy semantics**: `x` enthält danach eine **Kopie** von `e`, alle Komponenten von `e` werden in die Komponenten von `x` kopiert
- ▶ **Reference semantics**: `x` ist eine **Referenz** auf `e`
- ▶ C kopiert (Referenzen sind in der Sprache **explizit**)
- ▶ Java und Python referenzieren (alles ist eine Referenz, Kopie explizit über `clone`, `copy`, `deepcopy`)
- ▶ Haskell referenziert, aber Werte sind **unveränderlich**

## Verwandt damit: Gleichheit

- ▶ Identität vs. strukturelle Gleichheit
- ▶ Identität: Referenz auf das gleiche Objekt im **Speicher**
- ▶ Strukturelle Gleichheit: gleicher "Inhalt"
  - ▶ Java: `==` für Identität (der Referenzen), `equals` für strukturelle Gleichheit
  - ▶ Python: `is` für Identität (der Referenzen), `==` für strukturelle Gleichheit
  - ▶ C: `==` auf Referenzen für Identität, `==` auf zusammengesetzten Typen für strukturelle Gleichheit
  - ▶ Haskell: **nur** strukturelle Gleichheit (`==`, Typklasse `Eq`)

## Referenzen

- ▶ In C sind Referenzen **explizit**: `struct c *x` deklariert `x` als Referenz auf `struct c`.
- ▶ Zwei Operatoren:
  - ▶ `&e` erzeugt Referenz auf `e` (muss ein l-value sein)
  - ▶ `*e` dereferenziert `e` (gibt ein l-value zurück)
- ▶ Erlaubt **dynamische Datenstrukturen**
- ▶ Der NULL-Pointer: Tony Hoares "One-Billion Dollar Mistake"<sup>2</sup>
- ▶ Alle anderen Sprachen: **implizite Referenzen**

<sup>2</sup><https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/>

## Semantik von Aggregierten Typen

## Was brauchen wir?

- ▶ Erweiterung der **Sprache**: mehr Typen, mehr Ausdrücke
- ▶ Erweiterung des **Speichermodells**
- ▶ Semantik für die neuen **Ausdrücke**

## Erweiterung der Sprache

```
t ::= int | bool | array n of t | struct(i : t)*
l ::= i | l.i | l[e]
e ::= Z | true | false | l
    | e1 + e2 | e1 - e2 | e1 * e2 | e1 / e2
    | e1 == e2 | e1 < e2
    | ! e | e1 && e2 | e1 || e2
    | l := e
    | let x = e1 in e2
```

## Größe eines Typs

$$\begin{aligned} \text{size}(\mathbf{int}) &= 1 \\ \text{size}(\mathbf{bool}) &= 1 \\ \text{size}(\mathbf{array } n \text{ of } t) &= n \cdot \text{size}(t) \\ \text{size}(\mathbf{struct}(f_i : t_i)_{i=1, \dots, n}) &= \sum_{i=1}^n \text{size}(t_i) \end{aligned}$$

Offset eines Feldes  $g$  in einer Struktur:

$$\text{offset}_{\mathbf{struct}(f_i : t_i)_{i=1, \dots, n}}(g) \stackrel{\text{def}}{=} \sum_{k=1}^{i-1} \text{size}(t_k) \quad f_i = g$$

## Erweiterung des Speichermodells

- Kann eigentlich bleiben:

$$\Sigma = \text{Loc} \rightarrow \mathbf{V}, \text{Loc} = \mathbb{N}, \mathbf{V} = (\mathbb{B} + \mathbb{Z} + \mathbb{N})_{\perp}$$

- Braucht zwei Hilfsfunktionen:

$$\text{mem\_cp}(\sigma, n, k, m) = \begin{cases} \sigma & k \leq 0 \\ \text{mem\_cp}(\sigma[n \mapsto \sigma(m)], n+1, k-1, m+1) & k > 0 \end{cases}$$

$$\text{mem\_alloc}(\sigma, n, k) = \begin{cases} \sigma & k \leq 0 \\ \text{mem\_alloc}(\sigma[n \mapsto \perp], n+1, k-1) & k > 0 \end{cases}$$

## Semantik: Auswertung von L-Werten

$$\Gamma \vdash \langle l, \sigma \rangle \rightarrow_{\text{Lexp}} \langle n, \sigma' \rangle$$

$$\frac{}{\Gamma \vdash \langle l, \sigma \rangle \rightarrow_{\text{Lexp}} \langle \Gamma(l), \sigma \rangle}$$

$$\frac{\Gamma \vdash \langle l, \sigma \rangle \rightarrow_{\text{Lexp}} \langle n, \sigma' \rangle \quad \Gamma \vdash l : \mathbf{array } n \text{ of } t \quad \Gamma \vdash \langle e, \sigma' \rangle \rightarrow_{\text{Exp}} \langle v, \sigma'' \rangle, v \in \mathbb{Z}}{\Gamma \vdash \langle [e], \sigma \rangle \rightarrow_{\text{Lexp}} \langle l + \text{size}(t) \cdot v, \sigma'' \rangle}$$

$$\frac{\Gamma \vdash \langle l, \sigma \rangle \rightarrow_{\text{Lexp}} \langle n, \sigma' \rangle \quad \Gamma \vdash l : \mathbf{struct}(f_1 : t_1 \dots f_n : t_n) \quad \exists k. f_k = i}{\Gamma \vdash \langle l.i, \sigma \rangle \rightarrow_{\text{Lexp}} \langle n + \text{offset}_{\mathbf{struct}(f_1 : t_1 \dots f_n : t_n)}(i), \sigma' \rangle}$$

## Semantik: Ausdrücke

- Zu was wertet ein Ausdruck von aggregiertem Typ aus?
  - Ein **strukturierter Wert**? (Python, Haskell)
  - Eine **Referenz**? (Java)
- Ausdruck von aggregiertem Typ benötigen Sonderfälle für
  - L-Werte als Ausdrücke (i.e. auf der **rechten** Seite von Zuweisungen)
  - Zuweisungsausdrücke
- Wie allozieren wir **Variablen** aggregierten Typs?
  - Speicherblock von der Größe des Typs auf dem Stack?
  - Referenz in den Heap?

## Semantics: Die Regeln

$$\frac{\Gamma \vdash l : \mathbf{int} \text{ or } \Gamma \vdash l : \mathbf{bool} \quad \Gamma \vdash \langle l, \sigma \rangle \rightarrow_{\text{Lexp}} \langle n, \sigma' \rangle}{\Gamma \vdash \langle l, \sigma \rangle \rightarrow_{\text{Exp}} \langle \sigma'(n), \sigma' \rangle}$$

$$\frac{\Gamma \vdash l : \mathbf{array } n \text{ of } t \text{ or } \Gamma \vdash l : \mathbf{struct} \quad \Gamma \vdash \langle l, \sigma \rangle \rightarrow_{\text{Lexp}} \langle n, \sigma' \rangle}{\Gamma \vdash \langle l, \sigma \rangle \rightarrow_{\text{Exp}} \langle n, \sigma' \rangle}$$

$$\frac{\Gamma \vdash e : \mathbf{int} \text{ or } \Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash \langle l, \sigma \rangle \rightarrow_{\text{Lexp}} \langle n, \sigma' \rangle \quad \Gamma \vdash \langle e, \sigma \rangle \rightarrow_{\text{Exp}} \langle v, \sigma' \rangle}{\Gamma \vdash \langle l := e, \sigma \rangle \rightarrow_{\text{Exp}} \langle v, \sigma'[n \mapsto v] \rangle}$$

$$\frac{\Gamma \vdash e : t \text{ with } t = \mathbf{array } n \text{ of } t_1 \text{ or } t = \mathbf{struct} \quad \Gamma \vdash \langle l, \sigma \rangle \rightarrow_{\text{Lexp}} \langle n_1, \sigma' \rangle \quad \Gamma \vdash \langle e, \sigma \rangle \rightarrow_{\text{Exp}} \langle n_2, \sigma' \rangle}{\Gamma \vdash \langle l := e, \sigma \rangle \rightarrow_{\text{Exp}} \langle n_2, \text{mem\_cp}(\sigma', n_1, \text{size}(t)n_2) \rangle}$$

## Allokation von Variablen

$$\frac{\Gamma[x \mapsto l] \vdash \langle c, \text{mem\_alloc}(\sigma, l, \text{size}(t)) \rangle \rightarrow_{\text{Stmt}} \sigma' \quad \{l, \dots, l + \text{size}(t) - 1\} \cap \text{dom}(\sigma) = \emptyset}{\Gamma \vdash \langle \mathbf{new } x : t \text{ in } c, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma' \setminus \{l, \dots, l + \text{size}(t) - 1\}}$$

- Java: Aggregierte Werte (**Objekte**) nur auf dem Heap
- C: Referenzen als **expliziter Typ** ("pointer-to")

## Zusammenfassung

- Aggregierte Typen sind zusammengesetzt:
  - Kartesische Tupel
  - Endliche Abbildungen (dictionaries)
  - Disjunkte Vereinigungen
  - Rekursive Typen
- Bei der Behandlung von aggregierten Typen gibt es verschieden Möglichkeiten:
  - Zu was werten Ausdrücke aggregierten Typs aus?
  - Was passiert bei Zuweisungen und Gleichheit?
  - Wie werden Variablen behandelt?