3.3 Standarddatentypen

Ein Datentyp in einer Programmiersprache bezeichnet einen Wertebereich für Variablen. In einer streng getypten Programmiersprache wird bei einer Variablendeklaration ein Typ (explizit oder implizit) angegeben, sodass die Variable nur Werte des angegebenen Typs aufnehmen kann, was durch den Übersetzer geprüft wird.

3.3.1 Grundtypen

Man spricht von einem **Grundtyp** (oder manchmal auch von einem **skalaren Typ**), wenn alle Konstruktoren Konstanten sind. Übliche Grundtypen sind Wahrheitswerte, Zeichen oder Zahlen. Tabelle 3.1 gibt einen Überblick über die Namen und Werte der Grundtypen von Java.

Typ	Werte	Initialwert
`boolean`	`true, false`	`false`
`char`	16bit-Unicode-Zeichen	`\u0000`
`byte`	8bit ganze Zahl mit Vorzeichen	`0`
`short`	16bit ganze Zahl mit Vorzeichen	`0`
`int`	32bit ganze Zahl mit Vorzeichen	`0`
`long`	64bit ganze Zahl mit Vorzeichen	`0`
`float`	32bit-Gleitkommazahl	`0.0f`
`double`	64bit-Gleitkommazahl	`0.0`

Tabelle 3.1: Grundtypen in Java

Darüberhinaus gibt es in verschiedenen Programmiersprachen Konzepte zur Definition weiterer Grundtypen:

Aufzählungstypen Hier ist der Wertebereich eine endliche Menge von **Literalen**, d.h. festen Werten.

Beispiel: Aufzählungstypen in Modula-3:

```
TYPE Color = { Red, Green, Blue, Yellow };
VAR c: Color;
```

Beispiel: Aufzählungstypen in Rust:

```
enum Color { Red, Green, Blue, Yellow }
let mycolor = Color::Red;
```

Hier gehört der Bezeichner `Red` zum Namensraum des Aufzählungstypen `Color`, sodass der Typ als Qualifikator angegeben werden muss. Aufzählungstypen in Rust bieten aber noch mehr Möglichkeiten, als nur Literale aufzuzählen.

Beispiel: Aufzählungstypen in Java:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY;
}
...
Day d = SUNDAY;
```

In Java werden Aufzählungstypen in Klassen Übersetzt und bieten noch mehr Möglichkeiten, wie z.B. Definition von Methoden.

Ausschnittstypen Hierbei ist der Wertebereich eine Teilmenge aufeinanderfolgender Werte eines anderen Typs.

Zum Beispiel in Modula-3:

```
VAR day: [1..31];
day := 25;  (* ok *)
day := day + 10; (* Addition ok, Zuweisung fehlerhaft *)
```

Im Gegensatz zu Grundtypen sind bei **zusammengesetzten Typen** die Werte nicht elementar, sondern haben eine innere Struktur. Zusammengesetzte Typen bieten insbesondere Operationen zum Zugriff auf Teilwerte. Die wichtigsten zusammengesetzten Typen in imperativen Programmiersprachen sind Felder und Verbunde.

3.3.2 Felder

Ein **Feld** ist eine Sammlung von Elementen, wobei die einzelnen Elemente über einen **Index** zugreifbar sind.

Formal: Seien $I_1, \ldots, I_n$ Indexmengen und τ ein Typ. Dann heißt die Abbildung

$$A : I_1, \ldots, I_n \to \tau$$

ein **Feld** mit Indexmengen $I_1, \ldots, I_n$ und Elementtyp τ.
Somit kann ein Feld A als eine indizierte Sammlung von $|I_1| \cdots |I_n|$ Elementen aufgefasst werden.

Die Basisoperation auf einem Feld ist der Zugriff auf ein Feldelement, was meisten durch

$$A[i_1, \ldots, i_n]$$

mit $i_j \in I_j$ für alle $j \in \{1, \ldots, n\}$ bezeichnet wird.

In manchen Programmiersprachen können beliebige skalare Typen als Indexmengen verwendet werden, aber vielen Programmiersprachen sind die Indexmengen eingeschränkt:

- Die Indexmengen sind endliche Ausschnitte aus den natürlichen Zahlen.
- Fortran: Die Indexuntergrenze ist immer 1.
- C, C++, Java: Die Indexuntergrenze ist immer 0.

Zur Modellierung von Feldvariablen müssen wir festlegen, was der *Typ eines Feldes* ist. Dieser beinhaltet auf jeden Fall den Typ der Elemente und die Anzahl der Indizes, nicht aber unbedingt die Indexmengen selbst, da diese eventuell zur Übersetzungszeit noch nicht bekannt sind. Wir unterscheiden daher:

Statisches Feld: Die Indexgrenzen sind statisch bekannt und gehören zum Typ. Damit ist die Anzahl der Elemente zur Compilezeit bekannt.

Dynamisches Feld: Die Indexgrenzen werden erst zur Laufzeit bekannt. Die konkreten Indexgrenzen gehören damit nicht zum statischen Typ.

Flexibles Feld: Die Indexangaben können zur Laufzeit verändert werden. Dies ist ein eher seltenes Konzept. Es kommt z.B. in der Sprache Algol-68 vor. Zeiger in der Programmiersprache C können wie flexible Felder benutzt werden. In Java können Objekte der Klasse `Vector` wie flexible Felder verwendet werden.

Felddeklarationen haben in verschiedenen Programmiersprachen eine recht unterschiedliche Syntax:

- Modula-2: `VAR a: ARRAY [1..100] OF REAL;`
- Fortran: `DOUBLE a(100,100); // 2-dimensionales Feld`
- C: `double a[100,100];`
- Go: `var a[100,100]float64;`
- Java: `double[][] a = new double[100,100];`

  In Java wird die Deklaration (linke Seite) von der Erzeugung des Feldes (rechte Seite) getrennt. Die Zahl bei der Erzeugung bedeutet immer die Anzahl der Elemente in der Dimension, d. h. bei `[100]` sind die Indizes im Intervall `[0 .. 99]`.

Einige Anmerkungen zu Feldern in Java:

- Sie sind immer dynamisch.
- Sie müssen explizit erzeugt werden (zusätzlich zur Variablendeklaration).
- Eine Felderzeugung ist auch durch "initializers" möglich:

  ```
  int[] pow2 = {1,2,4,8,16,32};
  ```

- Die Deklaration und Erzeugung sind unabhängig voneinander:

```
int[][] a;
...
a = new int[42][99];
```

- Der Zugriff auf einzelne Element erfolgt durch Angabe der Indizes: `a[21][50]`
- Beachte: Unterschied zwischen L/R-Werten:

  ```
  a[i-1][j-1] = a[i+1][j+1];
  ```

  Der Ausdruck „`a[i-1][j-1]`“ auf der linken Seite bezeichnet die Referenz auf das Element mit dem Index `(i,j)`. Dagegen bezeichnet der Ausdruck „`a[i+1][j+1]`“ auf der rechten Seiten den Inhalt eines Feldelementes. Bei diesem Ausdruck muss also der L-Wert von „`a[i-1][j-1]`“ und die R-Werte von „`a[i+1][j+1]`“, „`i-1`“, „`j-1`“, „`i+1`“, „`j+1`“ berechnet werden!

Präzisierung von Feldern

Wir präzisieren nun die Bedeutung von Feldern (in Java) durch Angabe einer formalen Semantik für Deklarationen und L- und R-Werte.

Deklaration eines Feldes

$$\langle E \mid M\rangle\ \tau \underbrace{\texttt{[]}\ldots\texttt{[]}}_{n-mal} x\ \langle E; x : array(n,\tau) \mid M\rangle$$

Hier speichern wir im Typ zunächst nur die Dimension des Feldes.

Erzeugung eines Feldes

$$\frac{E \vdash^{lookup} x : array(n,\tau) \quad \langle E \mid M\rangle \vdash^R e_1 : d_1\ \ldots\ \langle E \mid M\rangle \vdash^R e_n : d_n}{\langle E \mid M\rangle\ x\ \texttt{=}\ \texttt{new}\ \tau[e_1]\ldots[e_n]\ \langle E; x : array(l,[d_1,\ldots,d_n],\tau) \mid M'\rangle}$$

wobei $a = d_1 \cdots d_n$, $l, l+1, \ldots, l+a-1 \in free(M)$ und, falls i der Initialwert des Typs τ ist, $M' = M[l/i][l+1/i]\ldots[l+a-1/i]$.

Somit wird beim Erzeugen der notwendige Platz für die einzelnen Feldelemente reserviert und die Anfangsadresse und Dimensionen bei x eingetragen (eigentlich müsste der Eintrag von x in der Umgebung E *verändert* werden, aber der Einfachheit halber fügen wir den veränderten Eintrag nur hinzu).

Zugriff auf ein Feldelement

$$\frac{E \vdash^{lookup} x : array(l,[d_1,\ldots,d_n],\tau) \quad \langle E \mid M\rangle \vdash^R e_1 : i_1\ \ldots\ \langle E \mid M\rangle \vdash^R e_n : i_n}{\langle E \mid M\rangle \vdash^L x[e_1]\ldots[e_n] : l_x}$$

$$\frac{E \vdash^{lookup} x : array(l,[d_1,\ldots,d_n],\tau) \quad \langle E \mid M\rangle \vdash^R e_1 : i_1\ \ldots\ \langle E \mid M\rangle \vdash^R e_n : i_n}{\langle E \mid M\rangle \vdash^R x[e_1]\ldots[e_n] : M(l_x)}$$

mit

$$l_x = l + d_n \cdots d_2 \cdot i_1 + d_n \cdots d_3 \cdot i_2 + \cdots + d_n \cdot i_{n-1} + i_n$$

Zusätzlich muss noch gelten, dass die Indizes in den richtigen Grenzen sind, d.h. $i_j \geq 0$ und $i_j < d_j$ für alle $j = 1, \ldots, n$.

Dies ist eine vereinfachte Darstellung des Feldzugriffs. In Java ist es auch möglich, nicht alle Indizes anzugeben. In diesem Fall ist das Zugriffsergebnis ein Feld über die restlichen Indizes.

Spezialfall: Zeichenketten

Zeichenketten werden in vielen Sprachen als Feld von Zeichen dargestellt, d.h. der Typ `String` könnte identisch zu `char[]` sein. Dies ist z. B. in C der Fall, wobei das letzte Zeichen das Nullzeichen `\000` ist.
In Java gilt dagegen:

- `String` ist ein eigener Typ (verschieden von `char[]`).
- `String`s sind nicht veränderbar, hierfür existiert stattdessen die Klasse `StringBuffer`.
- `String`konstanten haben eine feste Syntax:

  ```
  String s = "Java";
  ```

  Hierbei ist `"Java"` eine Stringkonstante.

- Zur Konkatenation von Strings gibt es einen vordefinierten Operator „+“:

  ```
  System.out.println(s+"2000");
     ⇝   Java2000
  ```

  Es erfolgt also eine automatische Konvertierung von Werten zu Strings bei der Verwendung von „+“. Diese automatische Konvertierung führt manchmal zu vielleicht unerwarteten Ergebnissen:

  ```
  System.out.println(""+1+2);   ⇝ 12
  System.out.println(1+2+"");   ⇝ 3
  ```

3.3.3 Verbunde

Verbund bezeichnen Datenstrukturen, bei denen im Gegensatz zu Feldern Daten unterschiedlichen Typs zusammengefasst werden. Formal ist ein **Verbund** das kartesische Produkt der Typen der einzelnen Komponenten, also ein Tupel. Ein wichtiger Aspekt von Verbunden ist allerdings, dass man einen Zugriff auf die einzelnen Komponenten über einen Namen für jede Komponente erhält.

Beispiel 3.4 (Verbunde in Programmiersprachen)**.**

- Verbunde in Modula-3:

```
TYPE Point = RECORD
               x,y: REAL   (* x und y sind Komponentennamen *)
             END;
VAR pt: Point;
pt.x := 0.0;  (* Selektion der Komponente x von pt mit . *)
```

- Verbunde in Go:

```
type Point struct {
       x,y float64
}
...
var pt Point
pt = Point{1.5, 3.4}
pt.x = pt.x + 0.5
```

- Verbunde in C:

```
typedef struct {double x,y} Point;
Point pt;
pt.x = 0.0;
```

- Verbunde in Rust:

```
struct Point {
       x: f64,
       y: f64
}
...
let mut pt = Point {x: 1.5, y: 3.4};
pt.x = pt.x + 2.3;
```

- Java: keine expliziten Verbunde, sondern Klassen:

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

```
Point pt = new Point();
pt.x = 0.0;
```

3.3.4 Vereinigungstypen

Ein **Vereinigungstyp** bezeichnet die Vereinigung der Werte verschiedener Typen. Da bei einer direkten Vereinigung es unklar ist, welchen konkreten Werttyp eine Variable eines Vereinigungstyps hat, werden in manchen Sprachen Vereinigungstypen nur als **variante Verbunde** angeboten.

Beispiel 3.5 (Variante Verbunde in C).

```
typedef struct {
  int utype;
  union {
    int ival;
    float fval;
    char cval;
  } uval;
} UT;

UT v;
```

Hier dient die Komponente `utype` dazu, die konkreten Varianten dieser Verbundsstruktur zu unterscheiden, denn in der Komponenten `uval` ist zu jedem Zeitpunkt nur eine der Varianten `ival`, `fval` oder `cval` vorhanden. Zum Beispiel erhalten wir durch `v.uval.ival` einen `int`-Wert und durch `v.uval.fval` einen `float`-Wert. Zu beachten ist aber, dass die Komponenten `ival` und `fval` *an der gleichen Stelle* gespeichert werden. Es ergibt sich die folgende Speicherstruktur:

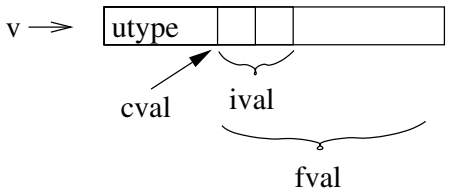

Abbildung 3.1: Speicherstruktur eines varianten Verbundes in C

Aus diesem Grund muss der Programmierer kontrollieren (z. B. mittels `utype`), welche Variante jeweils aktuell ist. Ein Problem kann sich leicht durch falsche Variantenbenutzung ergeben:

```
v.uval.fval = 3.14159;
```

```
v.uval.ival = 0;
```

Hierdurch wird ein Teil des `float`-Wertes 3.14159 mit 0 überschrieben. Der sich ergebende Wert von `v.uval.fval` ist implementierungsabhängig. Eine bessere und sichere Alternative zu varianten Verbunden sind Klassen und Unterklassen, bei denen das System den korrekten Zugriff kontrolliert (vgl. Kapitel 4).
In der Programmiersprache Rust, die eine sicherere Alternative zu C oder C++ darstellt, wird das obige Problem dadurch vermieden, dass jede Alternative mit einem unterschiedlichen Konstruktor (ähnlich wie algebraische Datentypen in funktionalen Sprachen) deklariert wird. Hierbei werden die schon erwähnten Aufzählungstypen in einer erweiterten Form eingesetzt:[4]

```
enum Value {
  Int(i32),
  Float(f64),
  Char(char)
}
```

Nun kann man z.B. die Float-Variante durch explizite Angabe des Konstruktors `Float` erzeugen:

```
let valf = Value::Float(3.14159);
```

Das obige Problem von C wird dadurch verhindert, dass man nicht einfach auf Teile eines Aufzählungswertes zugreifen kann, ohne explizit zu überprüfen, um welche Variante es sich handelt. Hierzu kann man eine Fallunterscheidung verwenden, wobei Rust noch die Möglichkeit bietet (wie in deklarativen Sprachen), Muster anzugeben, wodurch bei einem erfolgreichen Mustertest Variablen gebunden werden. Zum Beispiel können wir eine Funktion zur Umwandlung eines `Value` in eine ganze Zahl in Rust wie folgt definieren:

```
fn int_or_zero(x: Value) -> i32 {
   match x {
       Value::Int(i)   => i,
       Value::Float(f) => f as i32,
       Value::Char(c)  => c as i32,
   }
}
```

3.3.5 Mengen

In einigen Programmiersprachen kann man **Mengen** definieren und darauf spezielle Mengenoperationen anwenden.

[4] Anstelle der Tupelnotation „`Int(i32)`“ könnte man auch einen Verbund „`Int {ival: i32 }`“ deklarieren.

Beispiel 3.6 (Mengen in Modula-3)**.**

```
TYPE Color = {Blue, Green, Red, Yellow};
     ColorSet = SET OF Color;
```

Für Variablen vom Typ `ColorSet` sind nun die üblichen Mengenoperationen vordefiniert.

Da es für Mengen je nach Anwendungsfall unterschiedlich effiziente Implementierungen gibt, sind bei den meisten modernen Sprachen Mengen nicht mit einer speziellen Syntax vordefiniert, sondern man benutzt stattdessen geeignete Bibliotheken.

3.3.6 Listen

Listen sind Sequenzen beliebiger Länge und bilden in logischen und funktionalen Sprachen die wichtigste vordefinierte Datenstruktur. Eine Liste ist entweder leer oder besteht aus einem Element (Listenkopf) und einer Restliste. Listen sind in imperativen Sprachen mit Verbünden und Zeigern definierbar. Aus diesem Grund betrachten wir zunächst einmal Zeigerstrukturen.