Typen

Typen dienen der Strukturierung von Daten im Programm

Typen klassifizieren Daten nach verschiedenen Kategorien

Typen sind Wertemengen und Operationen darauf

Beispiel BOOLEAN:	Wertemenge:	TRUE, FALSE
	Operationen:	NOT, AND, OR
	Erzeugung:	Vergleichsoperationen

Grundlagen: freie Algebren und andere Formalismen

Es wird zwischen vordefinierten und benutzerdefinierten Typen unterschieden

Vordefinierte Typen

Vordefinierte Typen bieten eine Schnittstelle zur Hardware an und abstrahieren über die Hardware.

Beispiel: Je nach Typ wird der Inhalt "01001010" einer Variablen als ganze Zahl "74", als Zeichen "J", etc. interpretiert.

Übliche vordefinierte Typen:

- BOOLEAN: zweiwertige Bool'sche Algebra
- Zeichen; z.B. die Menge der ASCII-Zeichen
- Zahlen; z.B. die ganzen Zahlen im Bereich [-32768, 32767]
- Fließkommazahlen vordefinierter Größe und Genauigkeit

Gründe für die Verwendung von Typen

Verstecken der Maschinen-Repräsentation; Bitmuster zur Darstellung eines Wertes nicht zugreif- bzw. manipulierbar
→ verbessert Programmierstil und Wartbarkeit

Korrekte Verwendung von Variablen kann zur Übersetzungszeit geprüft werden → verbessert Korrektheit und Lesbarkeit

Auflösung überladener Operatoren kann zur Übersetzungszeit erfolgen → verbessert Ausführungszeit

Kontrolle der Genauigkeit; Platzbedarf der Variablen steht zur Übersetzungszeit fest → verbessert Speichereffizienz

Elementare Typen

Oft fallen vordefinierte und elementare Typen zusammen.

Ausnahmen:

"String" in Ada ist vordefiniert, aber nicht elementar:

type String **is array** (Positive **range** <>) **of** Character;

Aufzählungstypen sind elementar, aber nicht vordefiniert:

type color = (red, blue, green);	Pascal
type color **is** (red, blue, green);	Ada
enum color {red, blue, green};	C

Aggregate und Typkonstruktoren

Aggregate bestehen aus mehreren elementaren Datenobjekten; z.B. Arrays und Records.

In älteren Sprachen (FORTRAN, COBOL) werden Aggregate direkt erzeugt; es gibt keine Typen von Aggregaten.

Auch Routinen können als Aggregate elementarer Instruktionen angesehen werden.

In neueren Sprachen (Pascal, C, Ada) können neue Typen aus primitiveren Typen durch Typkonstruktoren erzeugt werden; Aggregate sind Instanzen solcher Typen.

Instanzen konstruierter Typen werden durch Konstruktoren — in einigen Sprachen auch direkt — erzeugt.

Kartesische Produkte

Das kartesische Produkt $A_1 \times \cdots \times A_n$ von n Mengen $A_1, \ldots, A_n$ ist die Menge aller n-Tupel $(a_1, \ldots, a_n)$, wobei $a_i \in A_i$.

In Programmiersprachen werden statt der Indexe Feld-Namen verwendet. Beispiele sind Records bzw. Strukturen.

```
typedef struct {
   int no_of_edges;
   float edge_size;
} reg_polygon;
reg_polygon a_pol = {3, 3.45};
```

Zur Auswahl von Feldern dient die Punkt-Notation:

```
a_pol.no_of_edges = 4;
```


Endliche Abbildungen

Eine mathematische Funktion bildet eine Menge von Werten (Domain) in eine andere Menge von Werten (Range) ab.

Beispiel: f: integer $\rightarrow$ real

Endliche Abbildung = Funktion mit endlicher Domain

Definition als Routine ist intensional.
(Wert der Funktion durch Regeln festgelegt)

Definition als Array ist extensional.
(Wert der Funktion durch vollständige Aufzählung festgelegt)

Vollständige Aufzählung nur für endliche Abbildungen möglich

Arrays (1)

Beispiel in C:

```
char digits[10];
for (i=0; i<10; ++i)
   digits[i] = ' ';
```

Es ist ein Fehler wenn ein Index nicht in der Domain liegt. Solche Fehler sind im allgemeinen erst zur Laufzeit erkennbar.

Beispiele in Pascal:

var x: **array**[2..5] **of** integer;
type manufacturer = (ibm, dec, hp, sun);
type m_data = **array**[manufacturer] **of** integer;
var m_profits, m_empl: m_data;

Arrays (2)

Initialisierung (Beispiele in C und Ada):

```
char digits[5] = {'a', 'b', 'c', 'd', 'e'};
```

X: **array** (INTEGER **range** 2..6) **of** INTEGER :=
(0, 2, 0, 5, -33);

Mehrdimensionale Arrays (Beispiele in C und Ada):

```
int y[10][20];
```

Y: **array** (1..27, M..N) **of** INTEGER;

Slicing (Beispiel in Ada):

X(2..5) := X(3..6);

Assoziative Datenstrukturen

In dynamisch typisierten Sprachen brauchen nicht alle Array-Einträge vom selben Typ sein.

Beispiel in SNOBOL4:

```
T = TABLE()
T<'RED'> = 'WAR'
T<6> = 25
T<4.6> = 'PEACE'
```

`T` ist eine assoziative Datenstruktur.

`T<6>` liefert `25` und `T<'RED'>` liefert `'WAR'`

Binden von Indexgrenzen

Die Indexgrenzen von Arrays können zu unterschiedlichen Zeitpunkten gebunden werden:

Übersetzungszeit: Indexgrenzen sind statisch festgelegt (C, Pascal, FORTRAN)

Zeitpunkt der Objekt-Erzeugung: Grenzen werden bei Initialisierung der Variablen festgelegt — dynamische Arrays (ALGOL 60, Ada)

Zeitpunkt der Objekt-Veränderung: Die Grenzen flexibler Arrays sind stets erweiterbar (SNOBOL4, APL, Algol 68, CLU)

Union

Definition einer Union in C:

```
union address {
   short int offset;
   long unsigned int absolute;
};
```

Man muss sich merken, welches Feld einer Union gesetzt ist:

```
enum descriptor {abs, rel};
typedef struct {
  address location;
  descriptor kind;
} safe_address;
```

Union mit Deskriminante

Varianter Record in Pascal:

```
type natural = 0..maxint;
     address_type = (absolute, offset);
     safe_address =
       record
         case kind: address_type of
           absolute: (abs_addr: natural);
           offset: (off_addr: integer)
       end
```


Powerset

Powerset = Menge aller Teilmengen einer gegebenen Menge

In Programmiersprachen: Menge ist Typ

Beispiel in Pascal:

```
type option = (list, optimize, save, exec);
     option_set = set of option;
var active_options: option_set;
...
active_options := [optimize, save];
if exec in active_options then ...
```


Rekursive Datentypen

Ein Datentyp T ist rekursiv wenn T zur Definition von T verwendet wird.

Beispiele:

bin_tree = $\{nil\} \cup (integer \times bin\_tree \times bin\_tree)$

int_list = $\{nil\} \cup (integer \times int\_list)$

Beispiel für Listen in ML:

```
fun find(el, nil) = false
|   find(el, x::xs) =
        if el=x then true
        else find(el, xs);
```


Unsicherheit von Zeigern

1. Untypisierte Zeiger (z.B. in PL/I)
   → Problem durch typisierte Zeiger lösbar
2. Arithmetische Operationen auf Zeigern (z.B. in C)
   → solche Operationen verbieten → Effizienzverlust
3. Dangling References durch Umgehung von Scopes
   → Laufzeitchecks, kein Adressoperator, nur Heap-Objekte
4. Dangling References durch explizite Deallokation
   → nur Garbage Collection statt expliziter Deallokation
5. Zeiger in nichtdiskriminanten Unions
   → diskriminante Unions verwenden

Zusammengesetzte Werte

Beispiele in C:

```
int_list x = {5, NULL};
char hello[] = {'h', 'e', 'l', 'l', 'o', 0};
```

Beispiele in Ada:

X:INT_LIST_NODE :=
    (VAL=>5, NEXT=>**new** INT_LIST_NODE(0, **null**));

type MATRIX **is array** (0..N, 0..M) **of** BOOLEAN;

Y:MATRIX := (1..N−1=>(0..M=>TRUE),**others**=>FALSE);

Y:MATRIX := (0|N=>(0..M=>FALSE), **others**=>TRUE);

Verwendung einer Klasse in C++

```
point p1(1, 3);                // instanziert p1
point p2(55, 0);               // instanziert p2
point* p3 = new point(0, 0); // p3 zeigt auf neuen Punkt
p1.x_move(3);                  // verschiebt p1 horizontal
p2.y_move(99);                 // verschiebt p2 vertikal
p1.reset();                    // positioniert p1
```

Copy Constructor:

```
point(point& p) {
   x = p.x;
   y = p.y;
}
```


Verwendung eines Typs in Eiffel

```
p1, p2: POINT;
...
!!p1.make_point(4, 7);
!!p2.make_point(55, 0);
p1.move_x(3);
p2.move_y(99);
p1.reset();
```


Typfehler und Typüberprüfung

Jeder Versuch, ein Objekt durch illegale Operationen zu manipulieren, ist ein *Typfehler*.

Ein *typsicheres* Programm enthält garantiert keine Typfehler.

Typsicherheit wird durch Typüberprüfungen sichergestellt.

Dynamische Typüberprüfungen erfolgen zur Laufzeit, *statische Überprüfungen* zur Übersetzungszeit.

Einige Typüberprüfungen können nur dynamisch durchgeführt werden.

Nicht jedes der Spezifikation widersprechende Verhalten eines Programms ist ein Typfehler.

Starke und statische Typsysteme

Ein Typsystem ist eine Menge von Regeln, die verhindern sollen, dass typunsichere Programme geschrieben werden.

Ein Typsystem ist *stark*, wenn es Typsicherheit garantiert. Eine Sprache mit starkem Typsystem ist *stark typisiert.* Der Übersetzer kann sicherstellen, dass keine Typfehler auftreten.

Der Typ jeden Ausdrucks in einer *statisch typisierten* Sprache ist zur Übersetzungszeit bekannt.

Jede statisch typisierte Sprache ist stark typisiert,
aber nicht jede stark typisierte Sprache ist statisch typisiert.

Typkompatibilität

Unterscheidung zwischen Typkompatibilität aufgrund von Namensgleichheit bzw. Strukturgleichheit.

Namensgleichheit ist strenger als Strukturgleichheit; Namen können die Intention des Programmierers ausdrücken.

Beispiele:

C verwendet Strukturgleichheit, ausgenommen Strukturen.

Ada verwendet Namensgleichheit, ausser Unterbereichstypen:

IA: **array** (1..100) **of** INTEGER;
IB: **array** (1..100) **of** INTEGER;

Typumwandlung

Implizite Typumwandlung (coercion) erfolgt in einigen Sprachen (C, C++, Algol 68) wenn ein aktueller Parameter einen zum formalen Parameter inkompatiblen Typ hat.

Beispiel in C:

```
int x;
float z;
...
x = x + z;
x = x + (int)z;
```

In Ada gibt es nur explizite Typumwandlungen:

I := INTEGER(X);

Skalare Typen in Ada

type Small_Int **is range** -10..10;
type Two_Digit **is mod** 100;

subtype Natural **is** Integer **range** 0..INTEGER'LAST;
subtype Positive **is** Integer **range** 1..INTEGER'LAST;

type Celsius **is new** Integer;
type Farenheit **is new** Integer;

type Float_1 **is digits** 10;
type Fix_Pt **is delta** 0.01 **range** 0.0..100.0;
type Dec_Pt **is delta** 0.01 **digits** 3;

type Week **is** (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
type Daily **is array** (Week) **of** Integer;
type At_Work **is array** (Week **range** Mon..Fri) **of** Integer;

Unbeschränkte Array-Typen in Ada

```
type Some_Days is array (Week range <>) of Integer;
type Int_Vector is array (Integer range <>) of Integer;
type Bool_Matrix is array
        (Integer range <>, Integer range <>) of Boolean;
type String is array (Positive range <>) of Character;

Useful_Work: Some_Days (Mon..Wed);
Z: Int_Vector (-100..100);
W: Int_Vector (20..40);
Y: Bool_Matrix (0..N, 0..M);
Line: String(1..80) := (1..80 => ' ');
```


Zeiger in Ada

```
T: Tree_Ref;
...
T := new Bin_Tree_Node;
T.all := (Info=>0, Left=>null, Right=>null)

type Message_Routine is access procedure(M: String);
Give_Message: Message_Routine;
...
Give_Message := Print_This'Access;
Give_Message.all (This is not an error");

Structure: array (1..10) of aliased Component;
type ComponentPtr is access all Component;
Mine, Yours: ComponentPtr;
...
Mine := Structure(1)'Access;
Yours := Structure(2)'Access;
```