# 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
$\rightarrow$ verbessert Programmierstil und Wartbarkeit

Korrekte Verwendung von Variablen kann zur Übersetzungszeit geprüft werden $\rightarrow$ verbessert Korrektheit und Lesbarkeit

Auflösung überladener Operatoren kann zur Übersetzungszeit erfolgen $\rightarrow$ verbessert Ausführungszeit

Kontrolle der Genauigkeit; Platzbedarf der Variablen steht zur Übersetzungszeit fest $\rightarrow$ 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 Diskriminante

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:

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

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

Beispiel für Listen in ML:

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

(C/C++)

```
typedef struct {
    int val;
    int_list* next;
} int_list;
int_list* head;
```

(Ada)

```
type INT_LIST_NODE;
type INT_LIST_REF is access INT_LIST_NODE;
type INT_LIST_NODE is
    record
        VAL: INTEGER;
        NEXT: INT_LIST_REF;
    end;
HEAD: INT_LIST_REF;
```

FIGURE 3.1 Declarations of list elements in C/C++ and Ada

# Unsicherheit von Zeigern

1. Untypisierte Zeiger (z.B. in PL/I)
   $\rightarrow$ Problem durch typisierte Zeiger lösbar

2. Arithmetische Operationen auf Zeigern (z.B. in C)
   $\rightarrow$ solche Operationen verbieten $\rightarrow$ Effizienzverlust

3. Dangling References durch Umgehung von Scopes
   $\rightarrow$ Laufzeitchecks, kein Adressoperator, nur Heap-Objekte

4. Dangling References durch explizite Deallokation
   $\rightarrow$ nur Garbage Collection statt expliziter Deallokation

5. Zeiger in nichtdiskriminanten Unions
   $\rightarrow$ diskriminante Unions verwenden

```
int* px;
void trouble()
{
    int x;          /* allocates x */
    ...
    px = &x;        /* assigns address of x to global px */
    ...
    return;         /* deallocates x */
}
main( )
{
    ...
    trouble();      /* after the call px is dangling */
    ...
}
```

FIGURE 3.2 An example of dangling pointers in C

# 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);
```

```
class point {
public:
    point(int a, int b) { x = a; y = b; }      // initializes the coordinates of a new point
    void x_move(int a) { x += a; }             // moves the point horizontally
    void y_move(int b){ y += b; }              // moves the point vertically
    void reset() { x = 0; y = 0; }             // moves the point to the origin
private:
    int x, y;
};
```

FIGURE 3.3 A C++ class defining point

# 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;
}
```

```
template<class T> class Stack{
public:
    Stack(int sz)        {top = s = new T[size = sz];}
    ~Stack()             {delete[ ] s;}    //destructor
    void push(T el)      {*top++ = el;}
    T pop()              {return *--top;}
    int length()         {return top - s;}
private:
    int size;
    T* top;
    T* s;
};

void foo() {
    Stack<int> int_st(30);
    Stack<item> item_st(100);
    ...
    int_st.push(9);
    ...
}
```

FIGURE 3.4 A C++ generic abstract data type and its instantiation

```
class POINT export
    x_move, y_move, reset
creation
    make_point
feature
    x, y: INTEGER;
    x_move(a: INTEGER) is
            -- moves the point horizontally
    do
        x := x + a
    end; --x_move
    y_move(b: INTEGER) is
            -- moves the point vertically
    do
        y := y + b
    end; --y_move
    reset is
            -- moves the point to the origin
    do
        x := 0;
        y := 0
    end; -- reset
    make_point(a, b: INTEGER) is
            -- sets the initial coordinates of the point
    do
        x := a;
        y := b
    end -- make_point
end; -- POINT
```

FIGURE 3.5 An Eiffel class defining point

# 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();
```

```
class NON_AXIAL_POINT export
    x_move, y_move
creation
    make_point
feature
    x, y: INTEGER;
    x_move(a: INTEGER) is
            -- moves the point horizontally
    require
        x + a /= 0
    do
        x := x + a
    ensure
        x /= 0
    end; --x_move
    y_move(b: INTEGER) is
            -- moves the point vertically
    require
        y + b /= 0
    do
        y := y + b
    ensure
        y /= 0
    end; --y_move
    make_point(a, b: INTEGER) is
            -- sets the initial coordinates of the point
    require
        a * b /= 0
    do
        x := a;
        y := b
    end -- make_point
invariant
    x * y /= 0
end; -- NON_AXIAL_POINT
```

```
class STACK[T] export
    push, pop, length
creation
    make_stack
feature
    store: ARRAY[T];
    length: INTEGER;

    make_stack(n: INTEGER) is
        do store.make(1, n);
                --this operation allocates an array with bounds 1, n
            length := 0
        end; --make_stack

    push(x: T) is
        do length := length + 1;
            put(x, length)
                --element x is stored at index length of the array
        end; --push

    pop: T is
        do Result := store@(length);
            -- the element in the array whose index is length is copied in the
            -- language predefined object Result, which always contains the
            -- value returned by the function
            length := length - 1
        end --pop
end --class STACK
```

FIGURE 3.7 An Eiffel abstract data type definition

# 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;

```
type s1 is struct {
    int y;
    int w;
};
type s2 is struct {
    int y;
    int w;
};
type s3 is struct {
    int y;
};
s3 func(s1 z)
{
...
};
...
s1 a, x;
s2 b;
s3 c;
int d;
...
a = b;          --(1)
x = a;          --(2)
c = func(b);  --(3)
d = func(a);  --(4)
```

FIGURE 3.8 A sample program

# 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);
```

```
type Int_Vector is array (Integer range < >) of Integer;
type Var_Rec(Tag: Boolean) is
    record X: Float;
        case Tag of
            when True =>   Y: Integer;
                           Z: Real;
            when False=>   U: Char;
        end case;
    end record;
subtype Vec_100 is Int_Vector(0..99);
    --this subtype constrains the bounds of the array to 0..99
subtype Var_Rec_True is Var_Rec(True);
    --this subtype freezes the variant where Tag = True;
    --objects of the subtype thus have fields X, Y, and Z;
subtype SMALL is Integer range -9..9;
    --this subtype defines a small set of integers
```

FIGURE 3.9 Examples of Ada types and subtypes

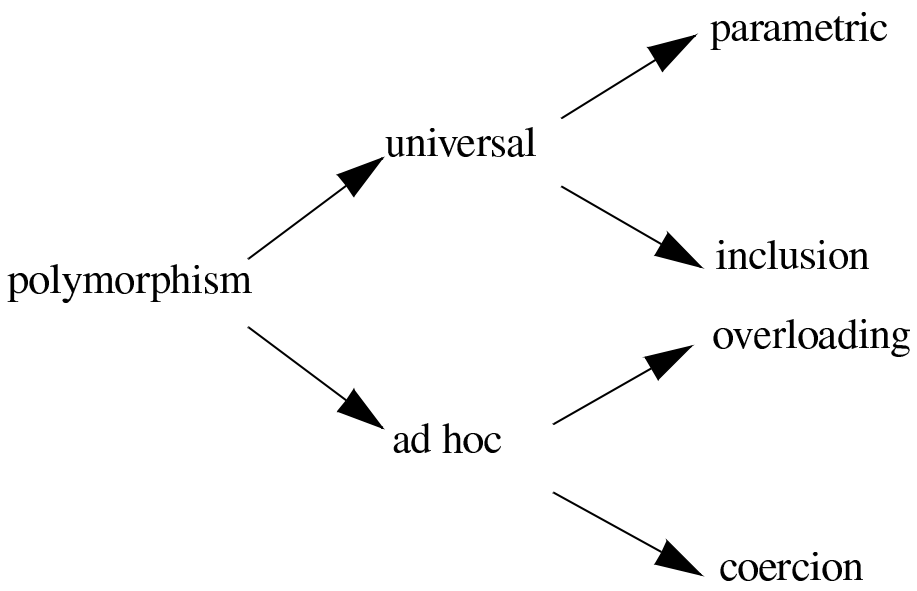


FIGURE 3.10 A classification of polymorphism

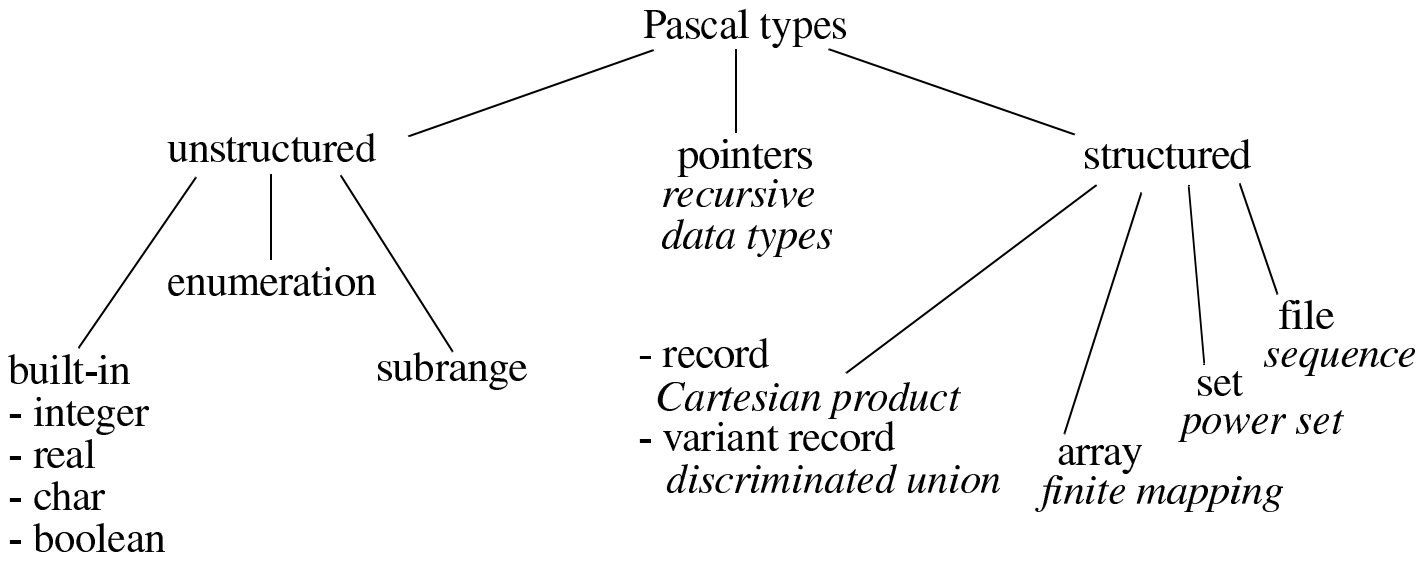


FIGURE 3.11 The type structure of Pascal

```
procedure sort(var a: array[low..high: integer] of CType);
var i: integer;
    more: boolean;
    temp: CType;
begin
    more := true;
    while more do begin
        more := false;
        for i := low to high - 1 do begin
            if a[i] > a[i + 1] then begin {move down element}
                temp := a[i];
                a[i] := a[i + 1];
                a[i + 1] := temp;
                more := true
            end
        end
    end
end
```

FIGURE 3.12 An example of conformant arrays in Pascal

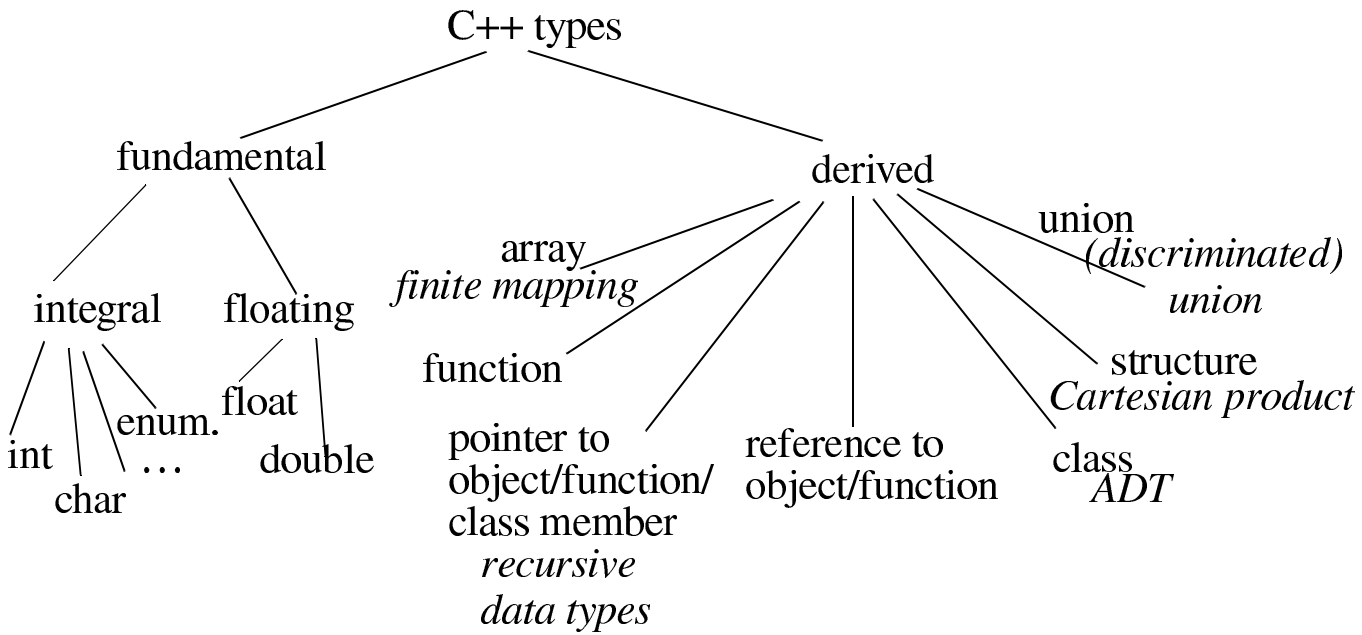


FIGURE 3.13 The type structure of C++

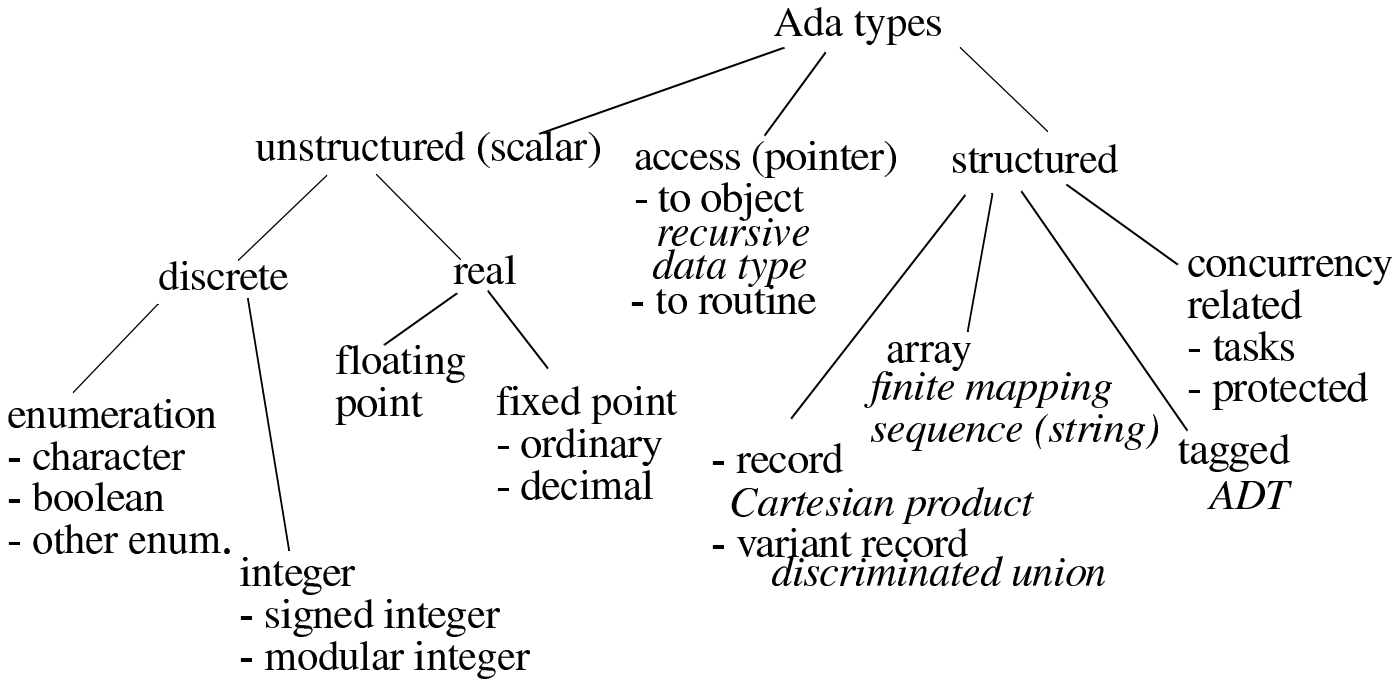


FIGURE 3.14 The type structure of Ada

# 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 => ' ');
```

```
function Sum(X: Int_Vector) return Integer;
Result: Integer := 0; --declaration with initialization
begin
    for I in X'First..X'Last loop
        --First and Last provide the lower and upper bounds of the index
        Result := Result + X(I);
    end loop;
    return Result;
end Sum;
```

FIGURE 3.15 A sample Ada function

```
type Address_Type is (Absolute, Offset);
type Safe_Address is record (Kind: Address_Type := Absolute)
    case Kind is
        when Absolute =>
            Abs_Addr: Natural;
        when Offset =>
            Off_Addr: Integer;
    end case;
end record;
```

FIGURE 3.16 An example of a discriminated union in Ada

```
type Bin_Tree_Node; --incomplete type declaration
type Tree_Ref is access Bin_Tree_Node;
type Bin_Tree_Node is
    record
        Info: Character;
        Left, Right: Tree_Ref:
    end;
```

FIGURE 3.17 Binary tree declaration in Ada

# 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;
```

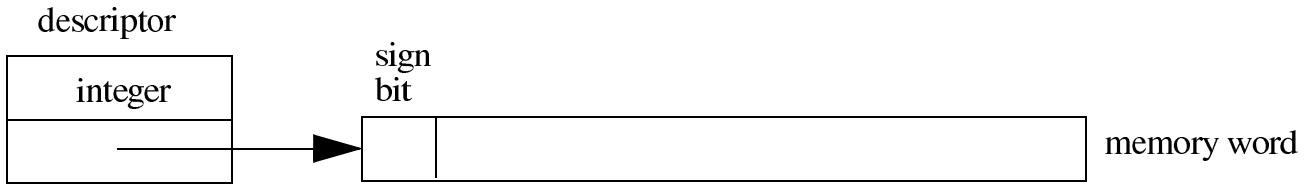


FIGURE 3.18 Representation of an integer variable

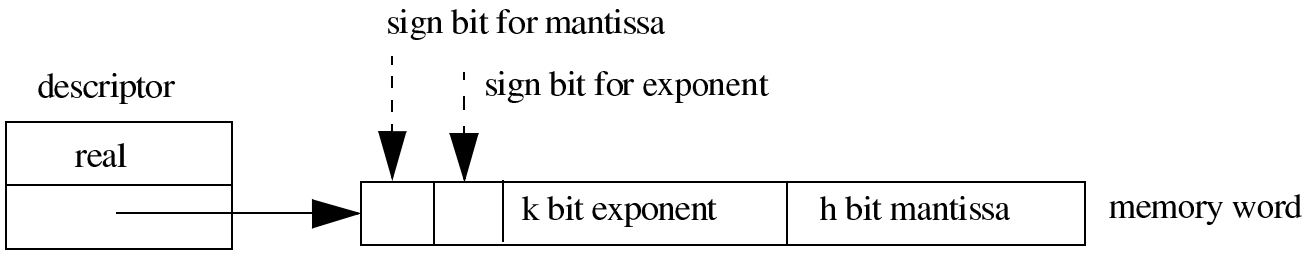


FIGURE 3.19 Representation of a floating-point variable

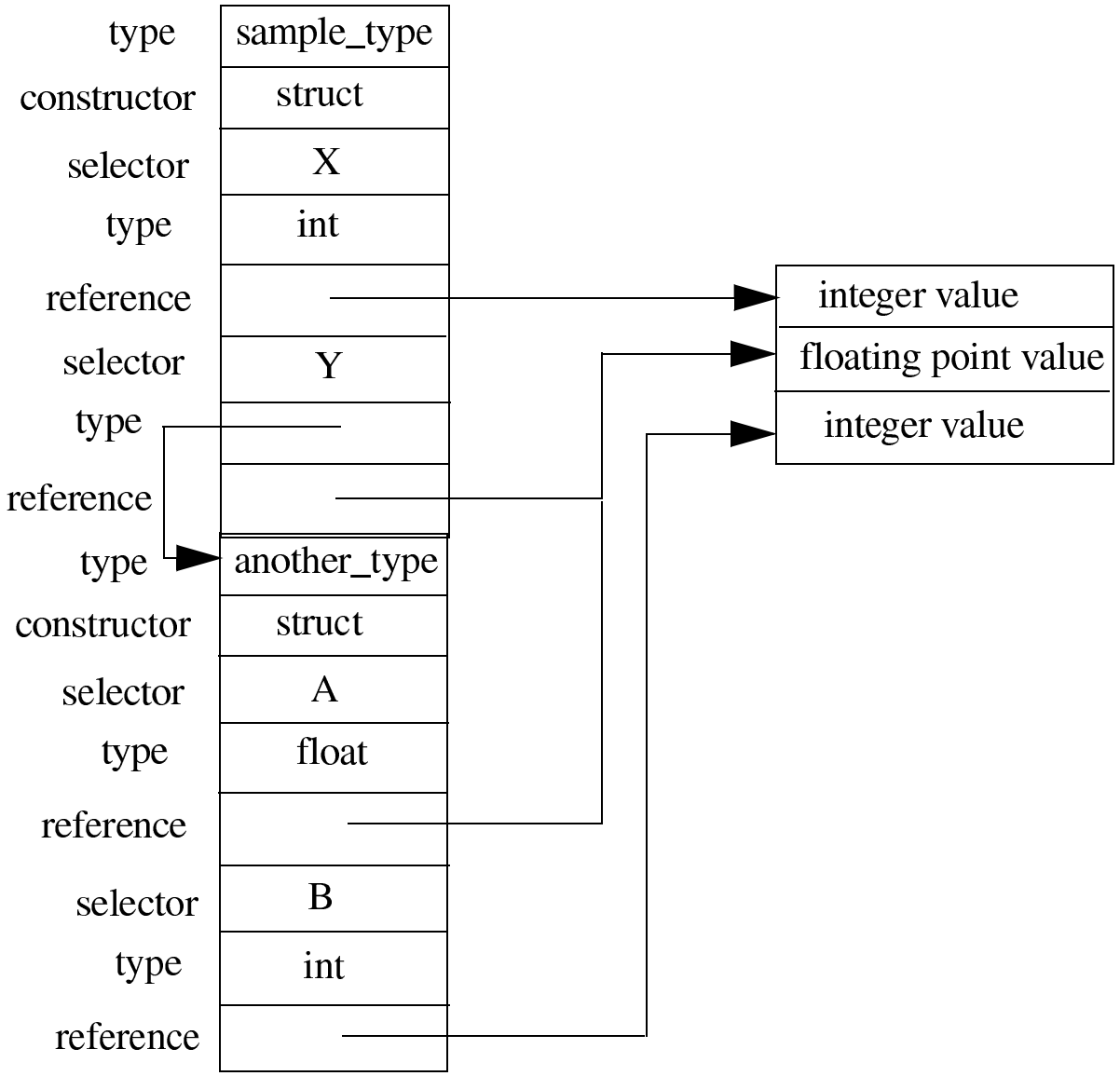


FIGURE 3.20 Representation of a Cartesian product

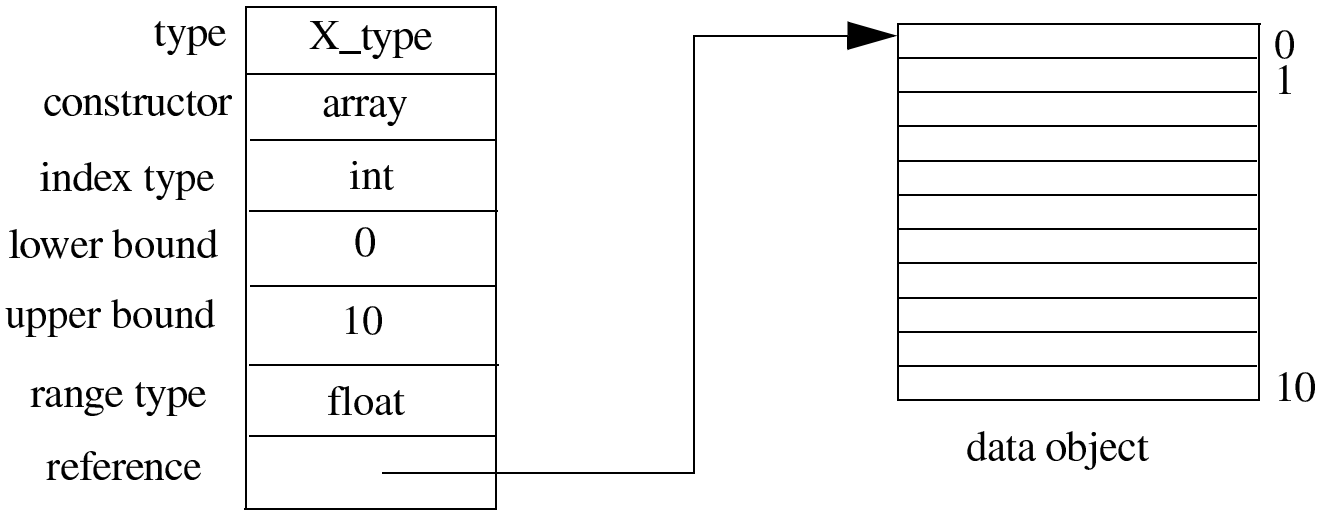


FIGURE 3.21 Representation of a finite mapping

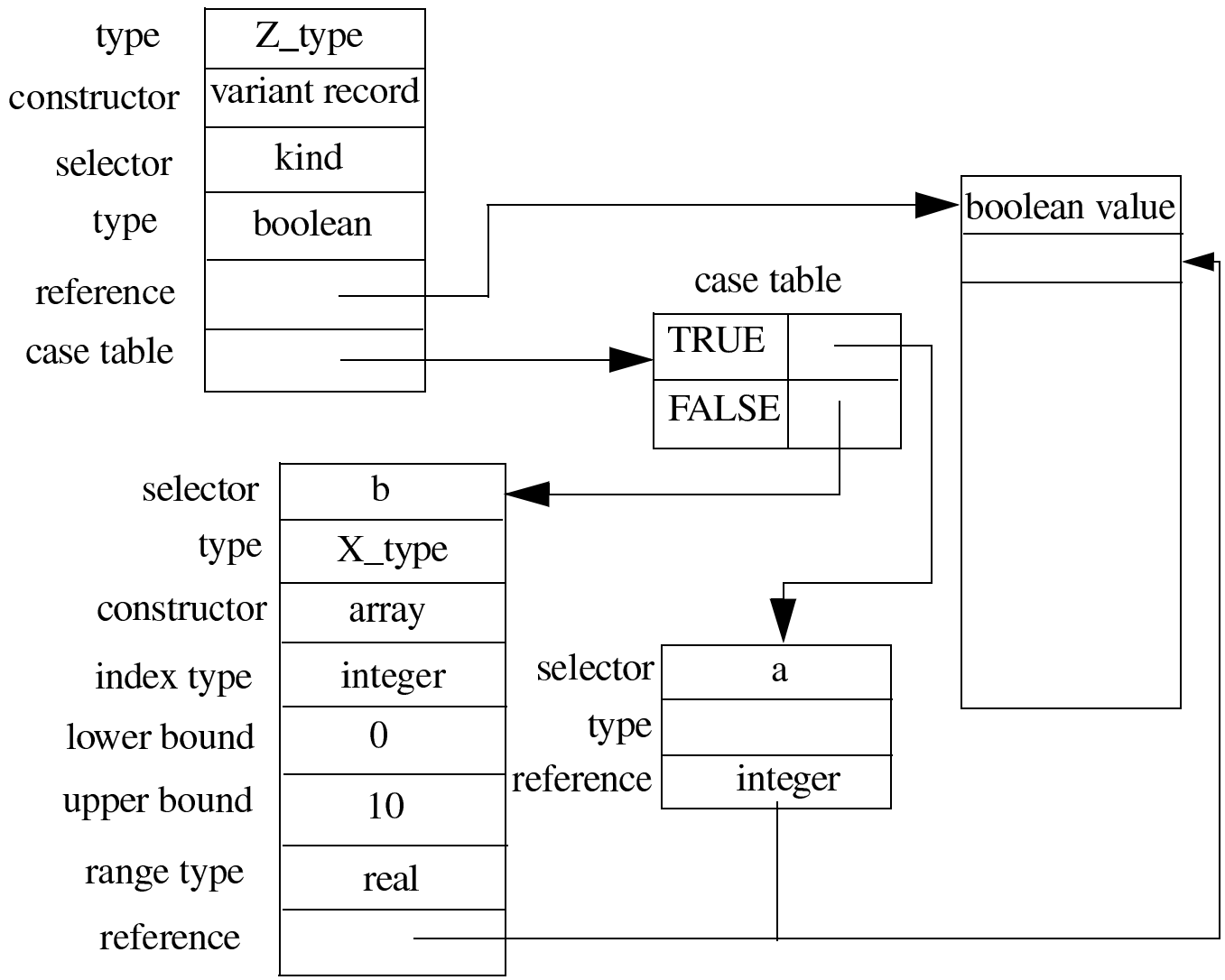


FIGURE 3.22 Representation of a discriminated union

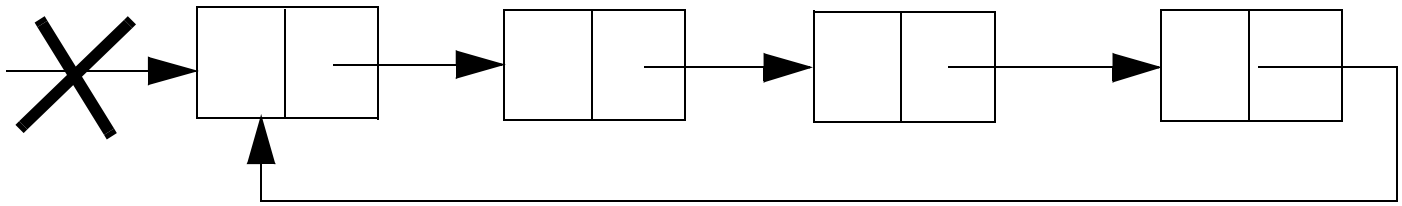

FIGURE 3.23 A circular heap data structure