CSc 520
Principles of Programming Languages

26: Types — Classification

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg

Enumerable Types

- Also called discrete types or ordinal types.
- Discrete types are countable, or 1-to-1 with the integers.
- Examples:
  1. integer
  2. boolean
  3. char
  4. subranges
  5. enumeration types

Scalar Types

- Also called simple types.
- The scalar types include:
  1. discrete types
  2. real
  3. rational
  4. complex

Composite Types

- Also called constructed types.
- They are created by applying type constructors to other, simpler, types.
- The composite types include:
  1. records
  2. variant records
  3. arrays
  4. sets
  5. pointers
  6. lists
  7. files
Discreet Types — Enumerations

- Pascal, Ada, Modula-2, C have some variant of enumeration types.
- C’s enumerations are just syntactic sugar for integer constants.
- In Pascal and Ada, enumerations are real types, incompatible with other types.
- In Ada and C, enumeration values can be user specified.

```pascal
TYPE Color = (white, blue, yellow, green, red);
TYPE Fruits = (apple=4, pear=9, kumquat=99);
VAR A : ARRAY Color OF Fruit;
FOR c := white TO red DO
  IF c != yellow THEN A[c] := apple;
```

Discreet Types — Subranges

- Subranges can be used to force additional runtime checks.
- Some languages use subrange types as array index types.

```pascal
TYPE S1 = [0..10];
TYPE S2 = ['a'..'z'];
TYPE Color = (white, blue, yellow, green, red);
TYPE S3 = [blue..green];
TYPE A = ARRAY S3 OF INTEGER;
VAR X : S3 := white; (* \(\Leftarrow\) error *)
```
Arrays – Storage Layout

Most languages lay out arrays in row-major order. FORTRAN uses column-major.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A[1,1]</td>
<td>0</td>
<td>A[1,1]</td>
<td>0</td>
<td>A[1,2]</td>
</tr>
</tbody>
</table>

Matrix Row Major Column Major

Array Indexing – 1 Dimensions

How do we compute the address (L-value) of the n-th element of a 1-dimensional array?

A_{elsz} is A’s element-size, A_{addr} is its base address.

VAR A : ARRAY [l .. h] OF T;

\[
L - \text{VAL}(A[i]) = A_{addr} + (i - l) * A_{elsz} \equiv A_{addr} + (l * A_{elsz}) + i * A_{elsz}
\]

C \equiv A_{addr} + (l * A_{elsz})

L - \text{VAL}(A[i]) \equiv C + i * A_{elsz}

Note that C can be computed at compile-time.

Array Indexing – 2 Dimensions

VAR A : ARRAY [l_1..h_1] [l_2..h_2] OF T;

\[
\begin{align*}
  w_1 & \equiv h_1 - l_1 + 1 \\
  w_2 & \equiv h_2 - l_2 + 1 \\
  L - \text{VAL}(A[i_1, i_2]) & \equiv A_{addr} + ((i_1 - l_1) * w_2 + i_2 + l_2) * A_{elsz} \\
  & \equiv A_{addr} + (i_1 * w_2 + i_2) * A_{elsz} - (l_1 * w_2 - l_2) * A_{elsz}
\end{align*}
\]

C \equiv A_{addr} - (((l_1 * w_2 + l_2) * w_3 + l_3) \cdots) * w_n + l_n) * A_{elsz}

\[
L - \text{VAL}(A[i_1, i_2]) \equiv (i_1 * w_2 + i_2) * A_{elsz} + C
\]

C can be computed at compile-time.

Array Indexing – n Dimensions

VAR A : ARRAY [l_1..h_1] \cdots [l_n..h_n] OF T;

\[
\begin{align*}
  w_k & \equiv h_k - l_k + 1 \\
  L - \text{VAL}(A[i_1, i_2, \ldots, i_n]) & \equiv ((i_1 * w_2 + i_2) * w_3 + i_3) \cdots) * w_n + i_n) * A_{elsz} + C
\end{align*}
\]

C can be computed at compile-time.
Record Types

Pascal, C, Modula-2, Ada and other languages have variant records (C’s union type):

```
TYPE R1 = RECORD tag : (red,blue,green);
    CASE tag OF
        red : r : REAL; |
        blue : i : INTEGER; |
        ELSE c : CHAR;
    END;
END;
```

Depending on the tag value, R1 has a real, integer, or char field.

The size of a variant part is the max of the sizes of its constituent fields.

Oberon has extensible record types:

```
TYPE R3 = RECORD
    a : INTEGER;
END;

TYPE R4 = (R3) RECORD
    b : REAL;
END;
```

R4 has both the a and the b field.

Extensible records are similar to classes in other languages.

Pointer Types

In order to build recursive structures, most languages allow some way of declaring recursive types. These are necessary in order to construct linked structures such as lists and trees:

```
TYPE P = POINTER TO R;
```

```
TYPE R = RECORD
    data : INTEGER;
    next : P;
END;
```

Note that P is declared before its use. Languages such as Pascal and C don’t allow forward declarations, but make an exception for pointers.

Procedure Types

C, Modula-2, and other languages support procedure types. You can treat the address of a procedure like any other object.

Languages differ in whether they allow procedures whose address is taken to be nested or not. (Why?)

```
TYPE P = PROCEDURE(x:INTEGER; VAR Y:CHAR):REAL;
VAR z : P; VAR c : CHAR; VAR r : REAL;
PROCEDURE M (x:INTEGER; VAR Y:CHAR):REAL;
    BEGIN...
END;
BEGIN
    z := M; /* z holds the address of M. */
    r := z(44,c);
END.
```
Class Types

Java’s classes are just pointer to record types. Some languages (Object Pascal, Oberon, MODULA-3) define classes just like records.

Nore about classes later.

```
TYPE C1 = CLASS
    x: INTEGER;
    void M() { ... };
    void N() { ... };
END;

TYPE C2 = CLASS EXTENDS C1
    r: REAL; // Add another field.
    void M() { ... }; // Overrides C1.M
    void Q() { ... }; // Add another method.
END;
```

Set Types

Pascal and Modula-2 support sets of ordinal types.

Sets are implemented as bitvectors.

Many implementations restrict the size of a set to 32 (the size of a machine word), or 256 (so you can declare a set of char).

```
type letset = set of 'A' .. 'z';
var x, y, z, w: letset;
begin
    x := ['A'..'Z','a'];
    y := ['a'..'z'];
    z := x + y; (* set union *)
    z := x * y; (* set intersection *)
    w := x - y; (* set difference *)
    if 'A' in z then ...; (* set membership *)
end.
```

Readings and References