Fundamentals of Programming Languages

PLs Typing Systems

Lecture 08

sl. dr. ing. Ciprian-Bogdan Chirila

Lecture outline

- Pascal Typing System
  - Predefined types
  - Programmer scalar defined types
  - Structured data types
  - Pointers
  - Type equivalence

Lecture outline

- C Typing System
  - Predefined types
  - Enumeration type constants
  - Structured data types
  - Pointers
  - Recursive structures
  - Types equivalence

Lecture outline

- Ada Typing System
  - Predefined types
  - Programmer scalar defined types
  - Structured data types
  - Pointers
  - Type equivalence
  - Subtypes and derived types

Lecture outline

- Lisp typing system
  - Simple predefined types
  - Lists
  - Vectors and matrixes
  - Vectors and bit matrixes
  - Character strings
  - Type equivalence
  - Subtypes
- Comparisons
- Strongly typed PLs

Pascal Typing System

- Predefined types
  - Numeric: integer, real
  - Non-numeric: boolean, char
- Programmer defined scalar types
  - Enumeration types:

    type section=(automation, computer, electronic, electrotechnical, energetic)
  - Subdomains

    type weak_currents=automation..computer;

    digits='0'..'9';

    month=1..12;

Programmer defined scalar types

- A subdomain is compatible with its base type
- Assignments between weak_currents and sections are allowed and even digits
- The variable range must be checked
- It can be done only at execution

Structured types. Arrays

- arrays implement in Pascal finite projections
- type t:=array[IT] of ET;
- IT – index type;
  - Must be specified at definition time before compile time
- ET- element type;
- The index type including its size is part of the array

Arrays

type

t=array[1..10] of integer;

t1=array[1..20] of integer;

- are different, incompatible types
- general purpose procedures to accept arrays as parameters can not be created
- the only solution is to declare formal parameters of maximum length
- effective length <= maximum length
- rigid and non economical solution

Arrays

- solution ISO Pascal Standard
- conforming arrays
  - a formal parameter array without index limit
  - will conform to an actual with
    - The same no of dimensions
    - The same element type

```
procedure p(var a:array[j..s:integer] of real);
var i:integer;
begin
    for i:=j to s do
        a[i]:=…
end
```

Records

- Implements in Pascal Cartesian products and variable reunions
- Involves specifying for each field its name and its type

```
type person=record
    name:array[1..30] of char;
    day,month,year:integer;
end;
```

Access using the selection mechanism

```
var author:person;
author.name:=”peter”;
author.day:=5;
author.month:=4;
author.year:=1970;
```

Access using the width mechanism

```
with author do
begin
    name:="peter";
    day:=5;
    month:=4;
    year:=1970;
end
```

Variable reunions

```
type person=record
     name:array[1..30] of char;
     day,month,year:integer;
     case man:boolean of
          true:(weight,height:real);
          false:(married:boolean);
     end;
var p:person;
```

Variable reunions with no selection field

```
type person=record
    name:array[1..30] of char;
    day,month,year:integer;
    case boolean of
        true:(weight,height:real);
        false:(married:boolean);
    end;
```

Variable reunions

p.name:="Paul";

p.day:=30; p.month:=10; p.year:=1980;

p.man:=true;

p.weight:=81;

p.man:=false;

if p.married then …

- the least safe in the Pascal PL

The set

- The set type
- Allows describing sets
  - reunion, intersection, difference
  - inclusion tests, membership tests
- The base type must be scalar and not real
- Limited cardinality
  - type t=set of integer; /* is wrong */
  - type t=set of 0..255; /* is correct */

The file

- Sequence of elements of the same type
- Type ftype = file of t;
- The base type can be any type;

Pointers

- In declaring a pointer one must declare the referred type
- Assures type compatibility in pointer related operations
- Null pointer marked with nil keyword
- Anonymous objects referred by pointers are created with the new operator

```
var p:^t;

new(p);
```

Pointers

- Releasing memory zones

```
dispose(p);
```

- Can not refer variables like in C
- Can refer only anonymous objects dynamically allocated

Type equivalence

- The semantics do not specify when two types are equivalent
- Different implementations have different type equivalence
- Pascal ISO standard adopted a definition
  - close to the name equivalence
  - aka declaration equivalence

Type equivalence

```
type
t=record
    n:array[1..30] of char;
    i:integer;
end;
tx=record
    n:array[1..30] of char;
    i:integer;
end;
t1=t;
var x:t;      y:tx;      z:t1;
```

Type equivalence

- x and z are compatible
  - Their type is described by the same record
  - It is not the case for name compatibility
- y is not compatible with x or z
  - As it would be in case of structural equivalence
- Subdomain types are compatible with the base types
  - This rule is also against name equivalence

The C typing system

- Predefined types
- enumeration constants
- Structured data types
  - Array
  - Structure
  - Union
- Pointers
- Recursive structures
- Type equivalence

Predefined types

- char – a byte for the local set of characters
- int – the set of integers on the host machine
  - Short int usually on 16 bits
  - Long int on at least 32 bits
- length(short) 16 bits
- length(short)<= length(int)<=length(long)
- signed and unsigned can be applied to char or int
- unsigned char 0..255
- signed char -128..+127
- float, double
- <limits.h> <float.h>

Enumeration constants

- enum boolean {NO,YES};
- enum days {MO=1,TU,WE,THU,FRI,SAT,SUN};

Arrays

- General form
  - element_type array_name[constant_expression]
  - Array size >0
- Example
  - v[10] – 10 integer array
  - Indexes start at zero
  - First element v[0]
  - Last element v[9]
- Initialization
  - int x[]={1,2,3};
- the array size must be known at compile time
  - C arrays are static arrays

Multidimensional arrays

- Is an array of arrays
- int mat[10][10]
  - Matrix with 10 lines and 10 columns
  - The element at (i,j) will be accessed like mat[i][j] and not mat[i,j] like in other PLs
- array formal parameters can be declared incompletely without specifying the first dimension
- int f(char l[],int m[][10]);

Multidimensional arrays

- The effective dimensions of arrays can be specified at function call time
- Functions can have a greater degree of generality than Pascal where
  - formal parameter size and actual parameter size must be equal

Structures

- Implement in C Cartesian products

```
struct point
{
    int x;
    int y;
};
```

- Can be copied by assignment
  - struct point origin={0,0};

Structures

- Field access

```
struct point p;
p.x or p.y
```

- Can be returned by functions

```
struct point f(int x, int y) { }
```

- Can be nested

```
struct rectangle
{
        struct point p1;
        struct point p2;
};
```

- The access can be nested

```
struct rectangle r;
r.p1.x
```

Unions

- Implement variable reunions

```
union
{
   int i;
   float f;
   char c;
} u;
```

- u can be an integer or a float or a char

Unions

- Selection
  - u.i, u.f, u.c
- Can be nested with other unions, structures or arrays
- In memory representations
  - all have a a zero memory offset from the starting address
  - At one moment only one representation is available
- No type checking is made
- All responsibility is on programmers shoulder
- Selecting a bad variant could cause severe programming errors
- The permitted operations are those from the sets
- Can be initialized with a value of the first variant type (integer for u)

Pointers

- a pointer declaration must use the referred type
  - int x=1, y;
  - int *p; /* p is a pointer to an integer */
  - void *p1; /* can store any type of pointer */
- May store object addresses
  - p=&x;
- To access the object referred by the pointer
  - is called unreferentiation
  - y=*p; /* y gets value 1*/
  - *p=0; /* x gets value 0 */
- Synonyms can be created with the known consequences

Pointers

- Allow direct access to an argument memory location

```
void exchange1(int x, int y) /*wrong*/
{
    int aux;
    aux=x; x=y; y=aux;
}
exchange1(a,b);
/*exchanges only copies of a and b*/
```

Pointers

```
void exchange2(int *x, int *y)
{
    int aux;
    aux=*x; *x=*y; *y=aux;
}
exchange2(&a,&b); /*correct call*/
/*exchanges the values of a and b*/
```

Pointers

- can be used together with arrays

```
int a[10];
int *pa;
pa=&a[0]; /*pa will hold the address of a[0]*/
```

- The value of an array is also the value of the first element of the array
- a and pa have the same values

Pointers

- *(pa+i) is the content of a[i]
- *(pa+i) is equivalent with a[i]
- (pa+i) is equivalent with &a[i]
- When an array is transmitted to a function
  - Only the first element address is transmitted
  - The formal parameter is actually a pointer
  - Acts as a variable which contains an address
- int f(char s[]) { … }
- int f(char *s) { … }
- The two forms are equivalent

Pointer arithmetic

- Allowed operations
  - Assigning pointers of the same type
  - Adding or subtracting a pointer with an integer
  - Subtracting or comparing two pointers referring the elements of the same array
  - Assigning or comparing with NULL (zero) or 0

Pointer arithmetic

- Illegal operations
  - Adding two pointers
  - Multiplying or dividing pointers
  - Bit shifting or mask application
  - Adding pointers with real values

Pointers to functions

- Allowed in C
- Can be assigned
- Can be set in arrays
- Can be send as parameters to functions
- Can be returned as values from functions

Dynamic memory allocation and relocation

- Dynamic allocation of anonymous objects of specified size
  - malloc(...)
  - calloc(...)
- Releases the allocate memory
  - free()
- Memory releases can create fake references

Recursive structures

- Based on pointers
- Allow describing lists or trees

```
struct node
{
    type info;
    struct node *left;
    struct node *right;
};
```

- recursive structures must use pointers
- a type can not contain its own instantiation

Type equivalence

- Based on structural equivalence
- Exceptions
  - struct
  - union
- are different types even they have the same structure
- type conversions are allowed through casting
- (type) expression

Ada typing system

- Predefined types
  - numerical
    - Integer
      - short_integer, long_integer
    - Float
      - short_float, long_float
- Non-numerical
  - character
  - boolean

Programmer defined scalar types

- Numerical
  - type under_hundred is range 0..99
  - type real is digits 7;
  - type small_real is digits 7 range 0.0..100.0;
  - Type centimes is delta 0.01 range 0.0..1.0;
    - 0, 0.01, 0.02, 0.03, 0.04, …, 1
- Enumeration types
  - Type sections is (automation, computer, electronics, electrotechnical, energetics)

Structured data types. Arrays

- can be declared statically
- type student_no is array[sections] of integer;
- type tab is array(1..10) of integer;
- No index limits must be specified
- only the base type is mandatory
- no restrictions array type
- type st_no is array(sections range<>) of integer;
- type matrix is array(integer range<>, integer range<>) of real;
- type bits is array(integer range<>) of boolean;

Arrays

- weak_currents_stud_no:st_no(automation..electronics)
- tab:matrix(1..5,1..5);
- Index restrictions must not be specified statically
- We can use expressions which values are computed at runtime
- Dynamic tables with
  - unspecified size at runtime
  - specified size at execution

Arrays

- mask : bits(1..n);
- mat : matrix(1..n,1..m);
- Array type
  - Element type
  - No of dimensions
  - Index base type on each dimension

Arrays

```
type vect is array(integer range<>) of real;
procedure p(a:in out vect) is
      temp:vect(a'first..a'last)
      i:integer;
begin
      for i in a'first..a'last loop
            temp(i):=a(i);
      end loop;
end p;
```

Arrays

- procedure p can be called with any actual parameter of random size
- first, last
  - Attributes that return the inferior and superior array limit

Articles

- Implement
  - Cartesian products
  - Variable reunions

```
type person is
    record
        name:string(1..30);
        day,month,year:integer;
    end record;
```

Articles

```
type person(man:boolean:=true) is
record
      name:string(1..30);
      day,month,year:integer;
      case man is
            when true=>weight,height:float;
            when false=>married:boolean;
      end case;
end record;
```

Articles

- the selector field is mandatory
- during execution time is checked the validity of the field reference based on the selector value
- any person object will have the man field set on true
- the weight and height fields will be accessible
- pm:person
- pf:person(false);

Articles

- illegal instructions
  - pm.man:=false;
  - pf.man:=true;
- it is possible to do
  - pm:=(false,”john”,25,05,1958,true);
  - pm:=pf;

Pointers

type ref is access t;

reference : ref;

----

reference:=new t;

- the null value is present

Pointers

- to avoid fictive references
  - Automatic memory deallocation of dynamic objects
  - Only when there is no pointer referring it
- unchecked_deallocation
  - Done manually by the programmer
  - Similar to Pascal dispose

Type equivalence

- Name equivalence
- Subtype facility
  - Avoids a rigid typing system
- Any type declaration introduces a new type
- x and y are incompatible

type price is range 0..integer'last;

type under_hundred is range 0..99;

x:price;

t:under_hundred;

----

y:=x; --it is ilegal

Subtyping

subtype under_hundred is price range 0..99;

t:under_hundred;

----

y:=x; --it is legal but 0<=y<=99

Subtyping

subtype no_of_students is new price;

a:price;

b:no_of_students;

a:=b; ----is illegal

Lisp Typing System

- Includes data types
- There is no variable in the classic sense
- Variables are replaced by symbolic atoms or symbols
- Symbols have a name which is an array of letters and do not represent a number
- Lisp is designed for symbolic computation

Lisp Typing System

- In imperative languages
  - To a variable we assign a value of a certain type
  - Referring the value is made through the variable name
- In Lisp
  - A symbol is a name attached to an entity for a certain amount of time
  - Data type does not refer to symbols but to the bound values
  - A symbol can represent at different times different values of different types

Lisp Typing System

- From the implementation point of view
  - Dynamic linking of several types to the very same variables is possible
  - Because Lisp variables are references (pointers) to entities which can be of several types
- In imperative languages
  - Variable is a name given to a memory location
  - With fixed dimension
  - Equal with the variable type

Binding a value to an atom

Replaces the assignment operation

Implemented by functional forms setq and setf

```
> (setq x 10)
10
> (setq x 'Lisp)
LISP
> (setq x '(a b c))
(A B C)
```

Lisp Typing System

- The type is specific to the object represented by the symbol
- But not the symbol itself
- It is the case for weak typing PLs
- At compile time is impossible to say what is the type of a variable
- Dynamic processing facilities are favored instead of type correspondence verifications during compile time

Predefined simple types

- Numerical
  - Integer
    - Fixnum
    - Bignum
  - Ratio
    - 10/3
    - 10/2
    - 10/4
    - (* 5/2 5/3)
    - 25/6

Predefined simple types

- Numerical (continued)
  - float
    - short-float
    - single-float
    - double-float
    - long-float
  - complex

  a+bi -> #c(a b)

  > (sqrt -1)

  #c(0 1)

  > (* #c(01) #c(0 2))

  -2
- Nonnumerical
  - character

Lists

- Non-atomic compound expressions are lists
- (red yellow blue)
- (1 2 -4 1.5)
- ((red yellow blue) (1 2 -4 1.5))
- The organization is linear, sequential
- Implemented as dynamic data structures
- In imperative languages
  - Dynamic allocation and deallocation of list elements
  - Done manually by the programmer
- In Lisp allocation and deallocation is done automatically

Lists

- Adding an element into a list using cons
  - (cons ‘d ‘(a b c))
  - (d a b c)
- Dynamic allocation for d
- Linking d into the list
- Are invisible operations for the programmer
- Two fields
  - car – pointer towards the first element of the list
  - cdr – pointer to the rest of the elements of the list

Vectors and matrixes

> (setq mat (make-array ‘(2 3 2):initial-contents
‘(((1 2)(3 4)(5 6)) ((7 8)(9 10)(11 12))))

#3A(((1 2)(3 4)(5 6)((7 8)(9 10)(11 12)))

Vectors and matrixes

```
> (setq vect (vector 0 1 2 3 4 5 6 7 8 9))

#(0 1 2 3 4 5 6 7 8)

> (aref mat 0 0 0)
1
> (aref mat 1 2 0)
11
```

Bit vectors and bit matrixes

```
> (setq matbits (make-array ‘(2 3 2)
:initial-element 0:element-type ‘bit))
#3A ((#*00 #*00 #*00) (#*00 #*00 #*00))

> (setq (aref matbiti 1 2 0) 1))
1
```

Bit vectors and bit matrixes

```
> (setq vbiti #*01010101)
#* 01010101

> (bit-not vbiti)
#* 10101010
```

- bit-not
- bit-and
- bit-ior, bit-xor
- bit-eqv - equivalence
- bit-orcl - implication

Strings

- Subtype of vectors

```
>(length "abcd")
4
>(aref "abcd" 2)
#\c
```

- String comparison

```
> (string = "abcd" "abcd")
T
> (string < "abcd" "abdd")
2
```

Strings

- Transforming an atom into a string

> (string ‘abcd)

“ABCD”

- Searching a substring in a string

> (search “cd” “abcd”)

2

Type equivalence. Subtypes

- Lisp programmer must no be aware of data types
- In older versions type did not exist
- Type dynamic linking avoids static checking
- Only checking is made when an operator executes its operands

>(+ 1 “5”)

Subtypes

- Numerical types
  - Number
    - rational
      - integer
        - fixnum
        - bignum
      - ratio
    - float
      - short-float
      - single-float
      - double-float
      - long-float
    - complex

Subtypes

- Vector types
  - vector
    - string
    - bit-vector
- Operators
  - type-of 1 arg
  - type-p() 2 args
  - subtype-p() 2 args

Types example

```
> (type-of 1)
FIXNUM
> (type-of #*01000111)
(SIMPLE-BIT-VECTOR 8)
> (type-of #\a)
CHARACTER
> (type-of ”abcd”)
SIMPLE-STRING
```

Subtypes example

```
> (typep 1 ‘number)
T
> (typep 1 ‘integer)
T
> (typep 1 ‘fixnum)
T
> (typep 1 ‘bignum)
NIL
```

Subtypes example

```
> (typep (a b c) 'sequence)
T
> (typep (a b c) 'list)
T
> (subtypep 'integer 'number)
T
> (subtypep 'array 'sequence)
NIL
```