Fundamentals of Programming Languages

PLs Typing Systems
Lecture 08
conf. dr. ing. Ciprian-Bogdan Chirila

November 15, 2022

Lecture outline

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

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

PLs Typing Systems

1. C typing system
   - Predefined types
   - Enumeration type constants
   - Structured data types
   - Pointers
   - Recursive structures
   - Type equivalence
2. Python typing system
   - Predefined types
   - Strings
   - Booleans
   - Lists
   - Tuples

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
  - 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 the Cartesian products

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

- can be copied by an 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

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

Unions

- 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 de-referentiation
  - 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

Theoretical type compatibilities

- 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(...);`
  - `realloc(...);`
- 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`

PLs Typing Systems

1. C typing system
   - Predefined types
   - Enumeration type constants
   - Structured data types
   - Pointers
   - Recursive structures
   - Type equivalence
2. Python typing system
   - Predefined types
   - Strings
   - Booleans
   - Lists
   - Tuples

Python Typing System

- Text Type: str
- Numeric Types: int, float, complex
- Sequence Types: list, tuple, range
- Mapping Type: dict
- Set Types: set, frozenset
- Boolean Type: bool
- Binary Types: bytes, bytearray, memoryview
- None Type: NoneType

Predefined types

```
x = 5
print(type(x))
<class 'int'>
---
x = "Hello World" <class 'str'>
x = 20 <class 'int'>
x = 20.5 <class 'float'>
x = 1j <class 'complex'>
x = ["apple", "banana", "cherry"] <class 'list'
x = ("apple", "banana", "cherry") <class 'tuple
x = range(6) <class 'range'>
```

Predefined types

```
x = {"name" : "John", "age" : 36} <class ’dict’>
x = {"apple", "banana", "cherry"} <class ’set’>
x = frozenset({"apple", "banana", "cherry"})
<class ’frozenset’>
x = True <class ’bool’>
x = b"Hello" <class ’bytes’>
x = bytearray(5) <class ’bytearray’>
x = memoryview(bytes(5)) <class ’memoryview’>
x = None <class ’NoneType’>
```

Casting

```
# integers
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
# floats
x = float(1)      # x will be 1.0
y = float(2.8)    # y will be 2.8
z = float("3")    # z will be 3.0
w = float("4.2") # w will be 4.2
```

Casting

```
# strings
x = str("fcpl") # x will be 'fcpl'
y = str(2)     # y will be '2'
z = str(3.0)   # z will be '3.0'
```

Strings

```
print("Hello FCPL")
print('Hello FCPL')
```

Multiline Strings

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""

a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
```

Slicing Strings

```
b = "Hello, World!"
print(b[2:5])
#llo
print(b[:5])
#Hello
print(b[2:])
#llo, World!
print(b[-5:-2])
#orl
```

Modifying Strings

```
s=" Hello FCPL "
s.upper()
s.lower()
s.strip()
s.replace("H", "J")
s.split(",")
a="Alfa"
b="Romeo"
c=a+" "+b
```

Booleans

```
print(10 > 9)
print(10 == 9)
print(10 < 9)

bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])
# will return True
```

Booleans

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
# will return False
```

Lists

```
mylist = ["alfa", "beta", "gamma"]
print(len(thislist))
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
list4 = ["abc", 34, True, 40, "male"]
nextlist = list(("alfa", "beta", "gamma"))
print(nextlist[1]) # beta
print(nextlist[-1]) # gamma
print(nextlist[1:2]) # ["beta", "gamma"]
```

Accessing lists

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)

islist = ["apple", "banana", "cherry", "orange"
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)

thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

Adding items to lists

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)

thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Removing items from lists

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)

thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)

thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Iterating lists

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

Tuples

```
mytuple = ("apple", "banana", "cherry")
print(mytuple[1])
print(mytuple[-1])
print(mytuple[1:2])
```

Updating tuples

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

Unpacking tuples

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

Looping tuples

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Sets

```
myset = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
set4 = {"abc", 34, True, 40, "male"}
```

Set Methods

- add clear copy
- difference
- discard intersection
- isdisjoint issubset issupperset
- pop remove
- union update

Dictionary

```
thisdict =
{
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
thisdict["color"] = "red"
thisdict.update({"color": "blue"})
```

Iterating dictionaries

```
for x in thisdict:
  print(thisdict[x])

for x in thisdict.values():
  print(x)

for x in thisdict.keys():
  print(x)

for x, y in thisdict.items():
  print(x, y)
```

PLs Typing Systems

1. C typing system
   - Predefined types
   - Enumeration type constants
   - Structured data types
   - Pointers
   - Recursive structures
   - Type equivalence
2. Python typing system
   - Predefined types
   - Strings
   - Booleans
   - Lists
   - Tuples

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 matbits 1 2 0) 1))
1
```

Bit vectors and bit matrixes

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

> (bit-not vbits)
#* 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 types did not exist
- Type dynamic linking avoids static checking
- The only checking is made when an operator executes its operands >`(+ 1 "5")`

Subtypes

- Numerical types
  - 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
```

Bibliography

1. Brian Kernighan, Dennis Ritchie, C Programming Language, second edition, Prentice Hall, 1978.
2. Carlo Ghezzi, Mehdi Jarayeri – Programming Languages, John Wiley, 1987.
3. Horia Ciocarlie – Universul limbajelor de programare, editia 2-a, editura Orizonturi Universitare, Timisoara, 2013.