Recitation - 05

Jahnavi - jp5867@nyu.edu

Type Checking:

Type checking is the process of ensuring that a program obeys the type system's type compatibility rules. A violation of the rules is called a type error or type clash. A program that is free of type errors is considered to be Well-typed.

Languages differ greatly in the way they implement type checking. These approaches can be loosely categorized into

- strong vs. weak type systems
- static vs. dynamic type systems

Strong Vs Weak Type System:

Strong Typing:

A strongly typed language does not allow expressions to be used in a way inconsistent with their types (no loopholes).

- A strongly typed language thus guarantees that all type errors are detected (either at compile-time or at run-time).
- Examples of strongly typed languages include Java, Scala, OCaml, Python, Lisp and Scheme.

Weak Typing:

On the other hand, a weakly typed language allows many ways to bypass the type system (e.g., by using pointer arithmetic or unchecked casts).

- In such languages, type errors can go undetected, causing e.g. data corruption and other unpredictable behavior when the program is executed.
- C and C++ are poster children for weakly typed languages. Their motto is: "Trust the programmer".

Contd..

C is a poster children for weakly typed languages. Their motto is: "Trust the programmer".

```
const int myConstant = 5;

int* myVariable = (int *)&myConstant;

*myVariable = 6;
```

myConstant here is a const variable so it should not be modified once assigned. But in C, since you are referencing it via myVariable which is a pointer to an int by type casting it. So, when you are changing the value of myVariable, myConstant is also updated.

Python Vs JavaScript:

In Python:

```
var = 21;           #type assigned as int at runtime.

var = var + "dot";   #type-error, string and int
cannot be concatenated.

print(var);
```

In Javascript:

```
value = 21;

value = value + "dot";

console.log(value);

/*

This code will run without any error. As Javascript

is a weakly-typed language, it allows implicit conversion

between unrelated types.

*/
```

Static Vs Dynamic Type System:

Static typing: Variables have types

->Detect type errors at compile time
Advantages of static typing:

- more efficient: code runs faster because no run-time type checks are needed; compiler has more opportunities for optimization because it has more information about what the program does.

- better error checking: type errors are detected earlier

- better documentation: easier to understand and maintain code.

Dynamic typing: Variables do not have types

->Detect type errors during the run time
Advantages of dynamic typing:

- less annotation overhead: programmer needs to write less code to achieve the same task because no type declarations and annotations are needed.

- more flexibility

Static Typing in TypeScript:

```
1  function foo(a: number) {
2      if(a > 0) {
3          console.log("Hi");
4      } else {
5          console.log("3" + 5);
6      }
7  }
8  foo(1);
9  foo('1');
```

```
TSError: × Unable to compile TypeScript:
static_type_ts.ts:9:5 - error TS2345: Argument of type 'string' is
not assignable to parameter of type 'number'.

9 foo('1');
      ~~~
```

Dynamic Typing in Python:

Dynamic type system example in Python

```
1   def foo(a):
2       if a > 0:
3           print('Hi')
4       else:
5           print("3" + 5)
6
7   foo(1)
```

```
$ python3.8 dynamic_python
Hi
```

```
1   def foo(a):
2       if a > 0:
3           print('Hi')
4       else:
5           print("3" + 5)
6
7   foo(-1)
```

```
Traceback (most recent call last):
  File "dynamic_python", line 7, in <module>
    foo(-1)
  File "dynamic_python", line 5, in foo
    print("3" + 5)
TypeError: can only concatenate str (not "int") to str
```

Pointers :

A pointer used for low-level memory manipulation, i.e., a memory address.

In C, void is requisitioned to indicate this - A void pointer is a pointer that has no associated data type with it

Any pointer type can be converted to a void *.

```
int a [10];

void *p = & a [5];
```

A cast is required to convert back:

```
int * pi = ( int *) p; // no checks

double * pd = ( double *) p;
```

Problems with pointers:

Memory leaks: A memory leak occurs when all pointers to a value allocated on the heap has been lost.

```
int  isqrt (int i)
{
        int* work = new int;
        *work = i;
        return *work;
}
```

When we return from this function the local variable work is lost.

But that has the only copy of the address of the int that we allocated on the heap.

Each call to this function will leak a bit of memory.

Java does not have this problem due to garbage collection.

Problems with pointers:

Dangling references: refer to a pointer which was pointing at an object that has been deleted.

```
int *p = new int;
int *q=p;
delete p;
```

-> The pointer q still has the address of the object even though the memory for that object has been returned to the system.

-> If the memory allocated to the deleted object is re-used for another purpose,

-> The value visible via q may appear to "spontaneously" change

-> Storing a value via q may corrupt that other data

Duck Typing:

"If it walks like a duck and it quacks like a duck, then it must be a duck"

-> Suitability is determined by the presence of certain methods and properties, rather than the type of object itself.

-> Using Duck Typing, we do not check types at all. Instead, we check for the presence of a given method or attribute.

Duck Typing:

```
class Duck:
    def swim(self):
        print("Duck swimming")

    def fly(self):
        print("Duck flying")

    def quack(self):
        print("Duck quacking")


class Swan:
    def swim(self):
        print("Swan swimming")

    def fly(self):
        print("Swan flying")


class Whale:
    def swim(self):
        print("Whale SWimming")
```

i. If it can swim, it's a "duck" ⇒ {Duck, Swan, Whale} are "duck" objects

```
for obj in Duck(), Swan(), Whale():
    obj.swim()
Result:
    Duck swimming
    Swan swimming
    Whale SWimming
```

ii. If it can swim and fly, it's a "duck" ⇒ {Duck, Swan} are "duck" objects

```
for obj in Duck(), Swan(), Whale():
    obj.fly()
Result:
    Duck flying
    Swan flying
    Traceback (most recent call last):
    File "duck.py", line 26, in <module>
        obj.fly()
    AttributeError: 'Whale' object has no attribute 'fly'
```

iii. If it can swim, fly and quack, it's a "duck" ⇒ {Duck} is "duck" object

```
for obj in Duck(), Swan(), Whale():
    obj.quack()
Result:
    Duck quacking
    Traceback (most recent call last):
    File "duck.py", line 26, in <module>
        obj.quack()
    AttributeError: 'Swan' object has no attribute 'quack'
```

Type Equivalence:

Structural equivalence : Two types are equal if they have the same structure

Name equivalence : Two types are equal if they have the same name

Why do we need type equivalence ?

In any language, if we use assignment operator x = y, the compiler/interpreter should make sure that both x and y are same in some sense.

Type equivalence:

```
typedef struct {
        int data[100];
        int count;
        } Stack;

typedef struct {
        int data[100];
        int count;
        } Set;

Stack x, y;
Set r, s;
```

If **name equivalence** is used in the language then x and y would be of the same type and r and s would be of the same type, but the type of x or y would not be equivalent to the type of r or s.

So, valid operations: x = y; r = s;
Invalid operations: x = r;

If **structural equivalence** is used, the two types Stack and Set would be considered equivalent, which means that a translator would accept statements such as

```
x = r;
```

Type Conversion Vs Coercion:

Type Conversion: An explicit request by the programmer for the compiler to insert code into the program to convert one type to another, compatible type. For example:

```
int x;

float y = 6.3;

x = (int)y;
```

Type Coercion: The compiler performs an implicit conversion from one type to another, compatible type without informing the user. This conversion could cause a logic error at run-time if it is not dynamically checked. For example:

```
int x;

short y;

y = x;
```

The C compiler does an implicit coercion by taking the low-order 15 bits of x and assigning them to y. It also preserves the sign bit. This works well if the value of x fits in 15 bits and poorly otherwise.

Variant records in Ada:

A variant record is a record type that can have different sets of fields (ie variations) in different variables. The variations are chosen using a specified field as a tag

Example: Modeling a transaction:

All transactions have an amount field

Variant fields:

Cash transactions: Discount

Check transactions: CheckNumber

Credit transactions: Card Number and Expiration Date

Variant types in Ada:

```
type PaymentType is (Cash, Check, Credit);
 -- The_Type is called the discriminant of the type
 type Transaction(The_Type: PaymentType := Cash) is record
     Amount: Integer;
     case The_Type is
         when Cash =>
             Discount: boolean;
         when Check =>
             CheckNumber: Positive;
         when Credit =>
             CardNumber: String(1..5);
             Expiration: String(1..5);
     end case;
 end record;
```

Discriminant records in Ada:

A discriminated record type includes one or more special elements, called discriminants, that are used to parameterize the declaration. Their effects are felt during elaboration-time or execution-time.

They can be used, for example, to specify an array length within one component or the existence of other components called variants.

When a record type with discriminants does not include default values for its components, it is known as an **indefinite subtype**.

When variants are present, a case-like structure is part of the record type declaration.

Record types with discriminants and variants are known as variant records.

```
type text (length:positive:=20) is
    record
        value:string(1..length);
    end record;
```

Union in C:

A union is a special data type available in C that allows to store different data types in the same memory location.

```c
#include <stdio.h>
#include <string.h>

union Data {
    int i;
    float f;
    char str[20];
};
int main(void) {
    union Data data;
    data.i = 10;
    data.f = 10.01;
    strcpy( data.str, "C Free Union" );
    printf( "data.i : %d\n", data.i );
    printf( "data.f : %f\n", data.f );
    printf( "data.str : %s\n", data.str );
}
```

```
data.i : 1917198403
data.f : 3924290174042134016468900118528.00000
data.str : C Free Union
```

Subtyping:

A type defines a set of objects. A subtype (S <: T) defines a set of objects (S) that have at least the methods and fields of T.

- A subtype is a subset of the set defined by its parent type.
- A subclass inherits all methods and fields of superclass.
- If the values of S are a subset of T, then an expression expecting T values will not be unpleasantly surprised to receive S values.
- i.e. If S <: T , then every value of type S is also a value of T.
- In java, the Object class is the supertype of all classes, and it has the fewest methods and fields. Thus, for all class types C in Java, C <: Object.

Subtype Polymorphism:

Subtype polymorphism: The ability to use a subclass where a superclass is expected

Thus, dynamic method binding

class A { void m() { … } }

class B extends A { void m() { … } }

class C extends A { void m() { … } }

Client: A a; … a.m(); // Call a.m() can bind to any of A.m, B.m or C.m at runtime based on assignment!

Parametric polymorphism:

Parametric polymorphism is a programming language technique that enables the generic definition of functions and types, without a great deal of concern for type-based errors.

```
fun length xs =
    if null xs
    then 0
    else 1 + length ( tl xs )
```

Unicode:

-> A worldwide character encoding standard

-> Its main objective is to enable single unique character set that is capable of supporting all characters from all scripts as well as symbols.

-> It is capable of encoding atleast 1,110,000 characters

-> It is a superset of ASCII

Various unicode encodings:

Name	UTF-8	UTF-16	UTF-16BE	UTF-16LE	UTF-32	UTF-32BE	UTF-32LE
Smallest code point	0000	0000	0000	0000	0000	0000	0000
Largest code point	10FFFF	10FFFF	10FFFF	10FFFF	10FFFF	10FFFF	10FFFF
Code unit size	8 bits	16 bits	16 bits	16 bits	32 bits	32 bits	32 bits
Byte order	N/A	<BOM>	big-endian	little-endian	<BOM>	big-endian	little-endian
Fewest bytes per character	1	2	2	2	4	4	4
Most bytes per character	4	4	4	4	4	4	4

Unicode Transformation Format(UTF)

An algorithmic mapping from virtually every code point to a unique byte sequence

Each UTF is reversible thus every UTF supports lossless round tripping; mapping from any unicode coded character sequence to a sequence of bytes and back will produce S again

The conversions between all UTF encodings are algorithmically based, fast and lossless

Ex: UTF-7, UTF-8, YTF-16, UTF-32