1. Introduction

21st century is coined to be as the era of computer revolution. Almost every work that is done using man power can be performed using computers, thus saving time, improving efficiency and maintaining accuracy. All these are achieved using computer programs. The concept of computer programming can be explained as a set of procedures that needs to be followed to achieve the desired functionality or output using executable computer programs [1]. These computer programs are written in wide variety of programming languages. An official language in which the computer programs are written following a programming paradigm is referred to as the Programming language. These programming paradigms are mainly categorized into Imperative paradigm, Object-Oriented paradigm, Functional paradigm and Logic Paradigm [2]. Among these Object-Oriented programming paradigm is the one that gained more popularity and is widely used in the software industry.

The idea of Object-Oriented programming arouses from the notion of the real-world entities called Objects. The salient features of this programming paradigm are Encapsulation, Polymorphism, Inheritance, and Abstraction. Encapsulation is bounding properties (data members) and behavior(functionality) of an object. Polymorphism is referred to as the ability of an object to exhibit different behavior at different situations. When an object acquires the properties and behavior of the other object, it is Inheritance. Abstraction is the approach where the implementation details are hidden and allowing only necessary features, visible to the user. The reason for using this object-oriented programming paradigm is code reusability. Redundant code can be eliminated thus saving memory. Also, this style of programming provides modularity enhancing better understanding of code. Some programming languages which use this fashion of programming are Simula, Ada, SmallTalk, C++, Java, C#, Python etc.

2. Problem Statement

The main objective of the project is to understand the theory of Polymorphism and different ways to achieve this from the point of type theory. For this we have chosen two different languages, namely Ada and Python to implement, analyze and then comparison is made in both the languages. We discuss the concept of polymorphism in the following part of the report.

3. Polymorphism

As we have said above, if the same object can showcase varying functionality, based on the environment, then the object is said to exhibit polymorphic behavior. To illustrate this with a real-life example, a person when at work is an employee, when in class is a student, when playing Basketball is a player. He will perform different operations according to the place he is at. So, the person now is said to exhibit polymorphic behavior.

Now when discussing this concept connecting with programming languages, the object/function which needs to display this property is written in such a way that it differs in number of arguments and their types, the sub-classes where the functionality is written, etc. The task regarding which functionality an object should exhibit is left to the compiler. This is determined either at the compile time or run time based on the type of polymorphism and the language in which it is executed.

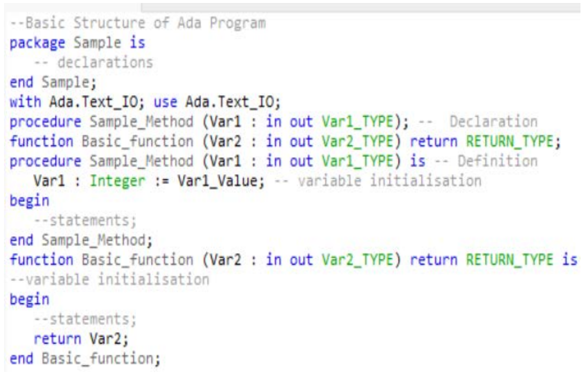
Luca Cardelli in his paper [3] has broadly categorized polymorphism mainly into two categories: Universal Polymorphism and Ad-hoc Polymorphism. When the same functionality can be applied irrespective to the type of arguments, it is Universal Polymorphism. Ad-hoc polymorphism is the polymorphism in which the functionality varies with respect to type and number of arguments that are applied. Considering these polymorphisms in detail, they are furthermore subdivided.

Universal Polymorphism can be achieved in two different ways: - Parametric Polymorphism and Inclusion Polymorphism. Parametric polymorphism can be achieved by using the concept of generics. Even when types of argument that needs to be passed as parameters are differed, the function behaves the same way as expected, then that function is said to achieve the property of parametric polymorphism. To illustrate with a basic example, the functionality of the stack should show same behavior even when the type of elements (i.e. integer, float, double, character etc.) on which it needs to perform are varied. On the other side, specific to object oriented programming there is another kind of polymorphism which can be obtained through inheritance and sub typing called inclusion polymorphism.

Ad-hoc polymorphism again have 2 sub-branches namely Overloading and Coercion. Overloading is done by altering the number of parameters, the type of parameters while maintaining the same variable name of the function always. For instance, we can implement the application of a calculator on any number of variables and data types changing the process within a function keeping the same function name say ‘Calculator’. When a variable of one type is implicitly converted to higher type to coordinate with type of other variable, then the notion is referred to as Coercion. Explaining with an example, addition of integer variable and floating-point variable is done by upcasting data type of integer variable to float implicitly by the compiler.

4. Polymorphism in ADA

To start with Ada is one of the oldest object-oriented programming language which is strongly typed. Fig 1 gives the basic structure of Ada program containing importing libraries, creation of packages, declarations of procedure and function, their definitions and variable initialization.

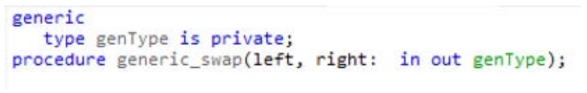


```
--Basic Structure of Ada Program
package Sample is
   -- declarations
end Sample;
with Ada.Text_IO; use Ada.Text_IO;
procedure Sample_Method (Var1 : in out Var1_TYPE); --  Declaration
function Basic_function (Var2 : in out Var2_TYPE) return RETURN_TYPE;
procedure Sample_Method (Var1 : in out Var1_TYPE) is -- Definition
   Var1 : Integer := Var1_Value; -- variable initialisation
begin
   --statements;
end Sample_Method;
function Basic_function (Var2 : in out Var2_TYPE) return RETURN_TYPE is
--variable initialisation
begin
   --statements;
   return Var2;
end Basic_function;
```

Fig 1. Basic Structure of Ada

4.1 Parametric polymorphism

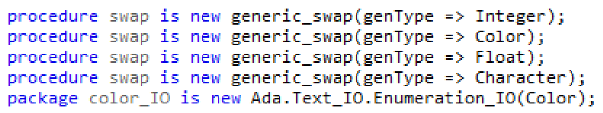
In Ada Parametric polymorphism is attained using the concept of generics. The keyword ‘generic’ is used before the parameters that are expected to exhibit the generic behavior. The parameter in Fig 2 represents the generic formal type parameter, which can be applied to almost all the pre-defined classes or datatypes in Ada. This formal parameter along with the keyword generic is called generic unit. These generic formal type parameters are then passed as arguments to the function or procedure that needs to exhibit the generic behavior which in turn are defined using these parameters.



```
generic
   type genType is private;
procedure generic_swap(left, right:  in out genType);
```

Fig 2. Generic Unit and procedure

The type of this generic formal type parameter that is to be assigned to the function is determined only after an object is created for it. Fig 3 represents the objects that are instantiated for Integer type, Float type, Color type and Character type. Now these functions are ready to showcase same functionality when the values of above created types are send as arguments. This is how parametric polymorphism is achieved.

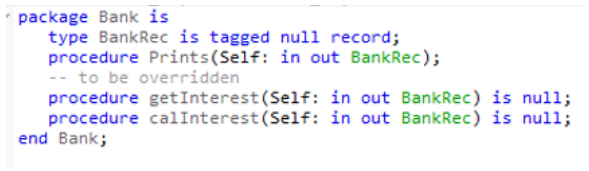


```
procedure swap is new generic_swap(genType => Integer);
procedure swap is new generic_swap(genType => Color);
procedure swap is new generic_swap(genType => Float);
procedure swap is new generic_swap(genType => Character);
package color_IO is new Ada.Text_IO.Enumeration_IO(Color);
```

Fig 3. Creating instances for generic objects

4.2 Inclusion Polymorphism

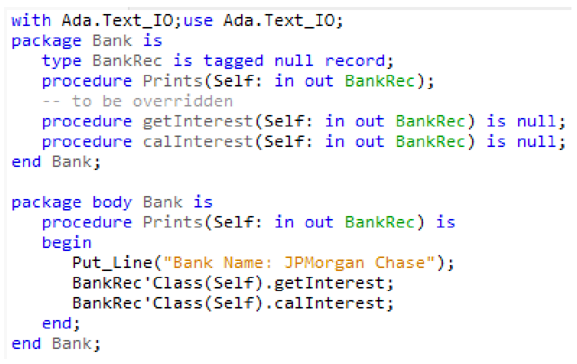
As discussed above inclusion polymorphism involves the notion of classes and inheritance. In Ada, class is defined using packages along with tagged types and primitive operations, as represented in Fig 4. Several different procedures or functions are bound in a single unit called Package. Tagged types provide a mechanism where an object can be inherited and extended with new set of operations along with the old ones. The keyword ‘tagged’ is used to declare a tagged type. Thus, using these tagged types, the polymorphic behavior is achieved at run-time. [4]



```
package Bank is
   type BankRec is tagged null record;
   procedure Prints(Self: in out BankRec);
   -- to be overridden
   procedure getInterest(Self: in out BankRec) is null;
   procedure calInterest(Self: in out BankRec) is null;
end Bank;
```

Fig 4: Class in Ada

A primitive operation can have arguments with no more than one tagged type in a package. There is a Class-wide type associated with each tagged type which is a combination of all the types that are inherited from the class and the class itself. Class-wide type is denoted as A ’Class where A represents object created from tagged type. But as the size of the class-wide type is not known until the runtime, it needs to be initialized with an “*access value*”. The theory of dynamic dispatching is attained using these class-wide types. When the method in the child class needs to provide different implementation as that of the base class using the same name, it is overriding. To override a method of child class in Ada, the method should be preceded using the keyword ‘overriding’.



```
with Ada.Text_IO;use Ada.Text_IO;
package Bank is
   type BankRec is tagged null record;
   procedure Prints(Self: in out BankRec);
   -- to be overridden
   procedure getInterest(Self: in out BankRec) is null;
   procedure calInterest(Self: in out BankRec) is null;
end Bank;

package body Bank is
   procedure Prints(Self: in out BankRec) is
   begin
      Put_Line("Bank Name: JPMorgan Chase");
      BankRec'Class(Self).getInterest;
      BankRec'Class(Self).calInterest;
   end;
end Bank;
```

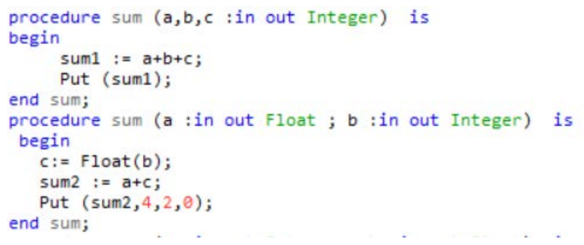
Fig 5. Inclusion Polymorphism in Ada

In the above example, getInterest() method can be redefined using the keyword ‘overriding’ in the sub-class. Both the overridden method and non-inherited methods can be included using this syntax:

accessBankRec : access Bank.BankRec'Class;
accessBankRec:= new CheckingAcc.CheckType;

4.3 Method Overloading

Ada supports both Method Overloading and Operator Overloading. Method Overloading can be done by varying the number and type of arguments passed to the procedure or function. In Fig 6, there are three parameters of same type in first procedure whereas there are two different type of parameters in second procedure. Both procedures have the same name and the procedure that needs to be invoked is determined by the compiler based on the arguments passed.

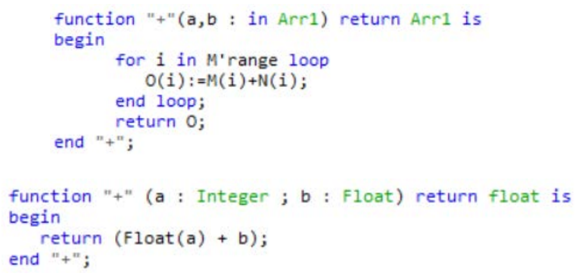


```
procedure sum (a,b,c :in out Integer)  is
begin
     sum1 := a+b+c;
     Put (sum1);
end sum;
procedure sum (a :in out Float ; b :in out Integer)  is
 begin
    c:= Float(b);
    sum2 := a+c;
    Put (sum2,4,2,0);
end sum;
```

Fig 6. Method Overloading in Ada

4.4 Operator Overloading

Ada does support Operator Overloading. Package Standard contains the functionality of all the operators, which is an Ada built-in library. To overload an operator, we need to place the operator within double quotes (i.e. “operator”) in the overloaded function. Overloaded operator is referred to as Designator. Once an operator is overloaded it can be used both normally (infix notation) and as a function with arguments (prefix notion). Arithmetic operators, relational operators, logical operators and unary operators can be overloaded in Ada.



```
function "+"(a,b : in Arr1) return Arr1 is
begin
        for i in M'range loop
           O(i):=M(i)+N(i);
        end loop;
        return O;
end "+";

function "+" (a : Integer ; b : Float) return float is
begin
   return (Float(a) + b);
end "+";
```

Fig 7. Operator Overloading in Ada

4.5 Coercion

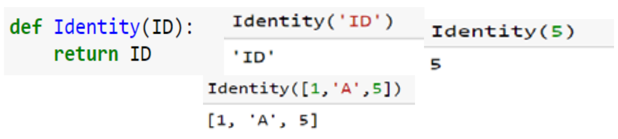
As Ada is strongly typed programming language it doesn’t support Coercion. However, the goal of coercion can be accomplished using type casting as follows: var1:= Float(var2);

5. Polymorphism in Python

Python is a dynamically and strongly typed language i.e., every variable in Python has a type and the type is determined by the content of that variable. Polymorphism in object oriented languages like Python can be mostly implemented using classes and inheritance.

5.1 Parametric Polymorphism

Generally parametric polymorphism is implemented using the concept of generics or templates in programming languages like ADA, JAVA, C++ etc., but Python doesn’t have support for generics. In Python variables can closely relate to the concept of pointers in C, so a Python variable always represents a memory location of the content that it holds. Due to this nature, Python has implicit polymorphic behavior. For example, consider a simple universal function like Identity function which takes an input and gives the same as output independent of the type of variable.



```
def Identity(ID):
    return ID
```

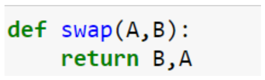
```
Identity('ID')
'ID'
```

```
Identity(5)
5
```

```
Identity([1,'A',5])
[1, 'A', 5]
```

Fig 8. Identity function

A type for the Python variable is never declared so generic behavior of the function in Python always depend upon the functionality defined inside the functions. For example, consider the same swap function example in ADA for explaining generics. In ADA a generic function must be declared, and it is defined for all the types that function should be acted upon. Where as in Python the below code snippet does the same function for all the types or objects.



```
def swap(A,B):
    return B,A
```

Fig 9. Generic swap function in Python

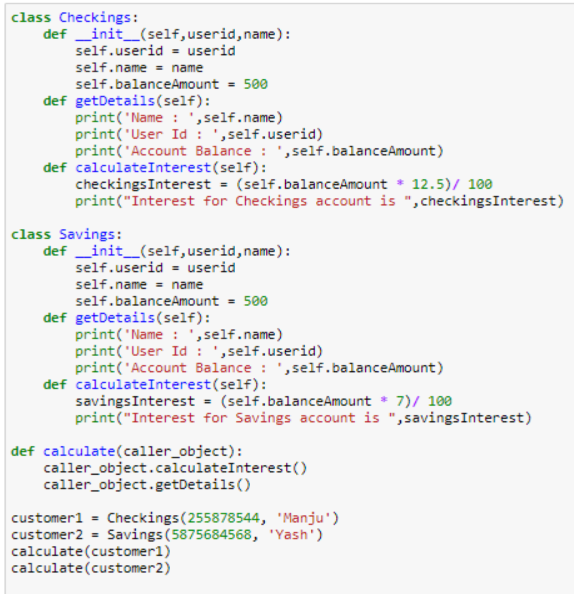
The type independent nature of Python implicitly allows its generic behavior [5]. Many operations such as print, +, * are examples of generic functions in Python.

5.2 Method Overriding

Re-defining the parent class method inside a child class is called Method overriding. In Python, method overriding can be achieved by simply defining the parent class function with the same signature [5] in the inherited-class. General concept of overriding when viewed from the point of type theory comes under inclusion polymorphism. In inclusion polymorphism a base class reference can access both the parent class's function and the overridden child function, but in python when the function is overridden its base class function is lost due to dynamic typing. Although Python has the concept of method overriding, it cannot be considered as inclusion polymorphism.

5.3 Duck Typing

Duck-typing can also be referred as behavioral sub-typing in Python. It comes from the basic idea that if something quacks then it's a duck, irrespective of its original properties [12]. Duck typing is a feature only implemented in dynamic typed languages. In simple terms, if one or more objects have the same method then a duck function can be written which can act as a polymorphic function on those objects. This duck function is independent of the objects and it can perform any action if the object that has those actions (functions and properties) defined for that object. Duck-typing is independent of inheritance, it mainly looks for the common behavior among the types. The following example is a simple implementation of duck-typing. Checking and Savings class in the below example doesn't have any relationship through inheritance. But both the classes have getDetails() and calculateInterest() function in common. So a duck function calculate() can be used to implement polymorphism through functions.



```
class Checkings:
    def __init__(self,userid,name):
        self.userid = userid
        self.name = name
        self.balanceAmount = 500
    def getDetails(self):
        print('Name : ',self.name)
        print('User Id : ',self.userid)
        print('Account Balance : ',self.balanceAmount)
    def calculateInterest(self):
        checkingsInterest = (self.balanceAmount * 12.5)/ 100
        print("Interest for Checkings account is ",checkingsInterest)

class Savings:
    def __init__(self,userid,name):
        self.userid = userid
        self.name = name
        self.balanceAmount = 500
    def getDetails(self):
        print('Name : ',self.name)
        print('User Id : ',self.userid)
        print('Account Balance : ',self.balanceAmount)
    def calculateInterest(self):
        savingsInterest = (self.balanceAmount * 7)/ 100
        print("Interest for Savings account is ",savingsInterest)

def calculate(caller_object):
    caller_object.calculateInterest()
    caller_object.getDetails()

customer1 = Checkings(255878544, 'Manju')
customer2 = Savings(5875684568, 'Yash')
calculate(customer1)
calculate(customer2)
```

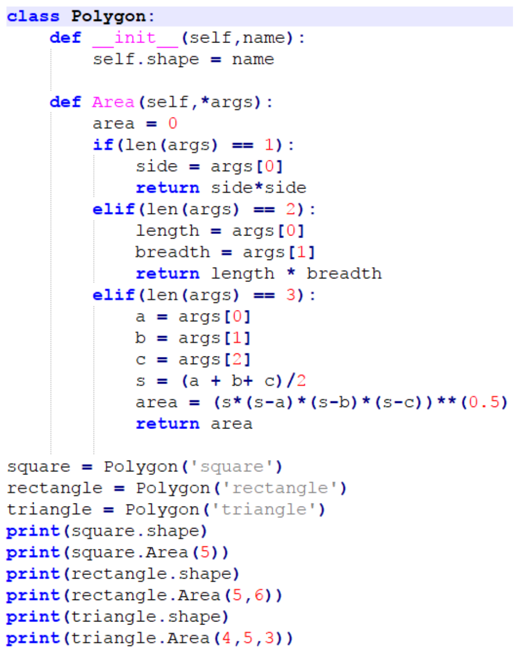
Fig 10. Duck typing function

5.4 Function Overloading

As per the definition of function overloading, the ability to have more than one function with the same function name but differs in the type of arguments or number of arguments [7]. Python doesn't have support for traditional function overloading. In Python functions are treated in the same way as variables, So the most recently defined function is only stored in the memory. But we can implement the behavior of overloading using the following methods:

5.4.1 *args (magic variables)

Python allows to send variable number of arguments using the special syntax *(variable_name) [8]. The functionality of overloaded functions can be implemented in the function depending upon the number of parameters or type of parameters passed to the function. The below code snippet shows the implementation of area function using the *args. In other programming languages like ADA, C++ or JAVA, this can be achieved by different implementations of area function.

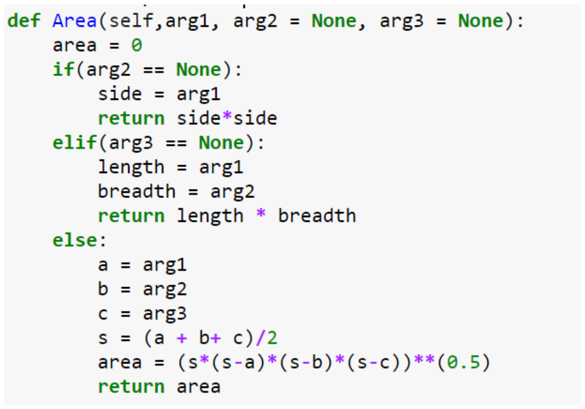


```python
class Polygon:
    def __init__(self,name):
        self.shape = name

    def Area(self,*args):
        area = 0
        if(len(args) == 1):
            side = args[0]
            return side*side
        elif(len(args) == 2):
            length = args[0]
            breadth = args[1]
            return length * breadth
        elif(len(args) == 3):
            a = args[0]
            b = args[1]
            c = args[2]
            s = (a + b+ c)/2
            area = (s*(s-a)*(s-b)*(s-c))**(0.5)
            return area

square = Polygon('square')
rectangle = Polygon('rectangle')
triangle = Polygon('triangle')
print(square.shape)
print(square.Area(5))
print(rectangle.shape)
print(rectangle.Area(5,6))
print(triangle.shape)
print(triangle.Area(4,5,3))
```

Fig 11. Function overloading thru Variable length args

5.4.2 Optional Parameters

For implementing the overloading functionality, optional parameters can also be used in Python [9]. The area function implemented using optional parameters is shown in the below code snippet. If a parameter is set to None or some default in the function declaration it can be treated as an optional parameter.

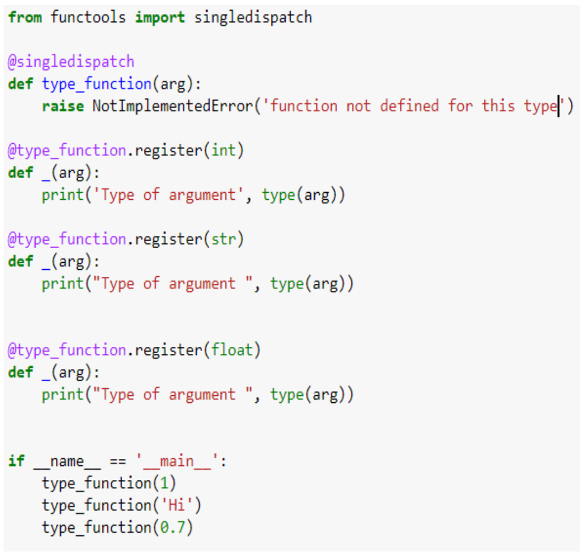


```python
def Area(self,arg1, arg2 = None, arg3 = None):
    area = 0
    if(arg2 == None):
        side = arg1
        return side*side
    elif(arg3 == None):
        length = arg1
        breadth = arg2
        return length * breadth
    else:
        a = arg1
        b = arg2
        c = arg3
        s = (a + b+ c)/2
        area = (s*(s-a)*(s-b)*(s-c))**(0.5)
        return area
```

Fig 12. Optional Parameters

5.4.3 Single dispatch

In Python 3, the overloading can be partially supported using the **singledispatch** decorator which is added to the functools module [10]. Using this decorator, a function can be transformed into single dispatch generic function. The function that must be called is determined based upon the first argument's type. The following code snippet [10] shows the simple usage of single-dispatch.

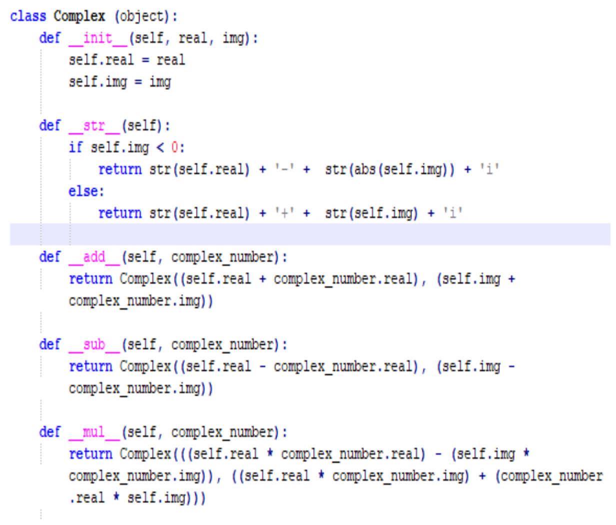


```python
from functools import singledispatch

@singledispatch
def type_function(arg):
    raise NotImplementedError('function not defined for this type')

@type_function.register(int)
def _(arg):
    print('Type of argument', type(arg))

@type_function.register(str)
def _(arg):
    print("Type of argument ", type(arg))


@type_function.register(float)
def _(arg):
    print("Type of argument ", type(arg))


if __name__ == '__main__':
    type_function(1)
    type_function('Hi')
    type_function(0.7)
```

Fig 13. Single Dispatch [10]

5.5 Operator Overloading

Operator overloading can be achieved in Python using magic methods. Magic methods are the special functions with syntax __functionname__ as method name [11]. Magic methods are never directely invoked with their functions names. All operators in Python has underlying magic method when the operator is used. Some examples of magic functions are __init__, __add__, __sub__ etc., Operators are overloaded in Python by redefining those magic methods for the respective operators. The below code snippet shows the overloading of "+", "-" and "*" operators to operate on complex numbers.



```
class Complex (object):
    def __init__(self, real, img):
        self.real = real
        self.img = img

    def __str__(self):
        if self.img < 0:
            return str(self.real) + '-' +  str(abs(self.img)) + 'i'
        else:
            return str(self.real) + '+' +  str(self.img) + 'i'

    def __add__(self, complex_number):
        return Complex((self.real + complex_number.real), (self.img +
        complex_number.img))

    def __sub__(self, complex_number):
        return Complex((self.real - complex_number.real), (self.img -
        complex_number.img))

    def __mul__(self, complex_number):
        return Complex(((self.real * complex_number.real) - (self.img *
        complex_number.img)), ((self.real * complex_number.img) + (complex_number
        .real * self.img)))
```

Fig 14. Operator Overloading

5.6 Type Coercion

In Python 3, lower types like Int will be implicitly converted to float or real in an operation involving both these types. If types are not compatible, these can be converted using explicit type conversion.

6. Comparison of Ada and Python

In this section, Ada and Python are compared based on syntactic structures, polymorphic factor and how each one tries to achieve various forms of polymorphism

6.1 Based on Syntax

Like other programming languages Ada and Python have some built in libraries, that needs to be imported to implement certain functionality. In Ada these libraries are imported using 'with' and 'use' keywords followed by library name, where in Python the same can be done by 'import' keyword. Indentation should be followed when programming in Python, which is not required in Ada.

As Ada is static and strongly typed language, we need to explicitly define the type of a variable before its usage in contrast to Python due to its dynamic typed nature.

Procedure statements doesn't return a value whereas Functions return a value in ADA. Keywords 'procedure' and 'function' are used to facilitate their respective purpose in Ada. Execution part of procedures and functions start with 'begin' and 'end'. In Python, 'def' keyword is used to define a function which may or may not return a value. This single keyword serves the purpose of both procedure and function in Ada. The concept of classes in Ada are implemented as a collaboration of Package, tagged type and primitive operations. Whereas class members and member functions are bound together using the keyword 'Class', followed by class name in Python.

To provide different implementations of a function in base class and derived class, we should use keyword 'overriding' before the function in derived class in Ada. Whereas overriding in Python can be implemented directly without any keywords in Python. There are different modes 'in', 'out', 'in out' and 'access' in which a variable can be operated in Ada, which is not provided in Python.

6.2 Polymorphic Factor

There are two different metrics for measuring the quality of polymorphism [13]. The first one is NMO, which determines the total count of the functions or methods, overridden in a single derived class. NMO is applied only to one class at time. And the second one is PF, which stands for Polymorphism Factor, shown in equation 1.

This calculates the quality on whole for the method that is overloaded.

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} M_n(C_i) * DC(C_i)}$$

Equation 1: Polymorphism Factor [13]

Where TC is the no of classes including all base and derived classes, Mo is total methods overridden, C_i is the current class, M_n is new methods in class which are not overridden, DC is the number of classes inherited from the current class.

The reason why we did not use this metric for determining quality is, we did not build a huge hierarchy of classes with overridden methods. All the examples in our work are implemented, in the perspective to understand how polymorphism works in two different languages. Applying this factor to the above-mentioned examples will output only the best values, which is not an optimal parameter to compare.

6.3 Comparison of Polymorphism

In Ada Parametric polymorphism is achieved using generics concept, while in Python it is done using "self.args" function. Inclusion Polymorphism is achieved using packages, tagged types, primitive operations and overriding keyword in Ada. Whereas in Python, duck typing is used. Operator overloading in Python is performed using Magic functions. Coercion is not supported in Ada, due to strong typing. However, this is achieved using typecasting. Python supports coercion due to dynamic typing

7. Conclusion

In this research paper, we have learnt various forms of polymorphism and implemented several examples to observe and understand how two specific languages behave with types to achieve the goal of polymorphic behavior

8. Critique

The above survey paper discusses about concept of polymorphism based on type theory and means of achieving this behavior. Also, two different programming languages ADA and Python are taken, for which their own way of implementing polymorphism is discussed. In the later section, a brief comparison on structural behavior, types of polymorphism that ADA and Python can support are addressed. Adding to this few examples are explained which tried to emulate polymorphic behavior in Python. So, what more could be done to make this project better if restarted. Firstly, we have referred resources online and learnt various types of polymorphism that ADA can support. We took time to familiarize the correct syntax and coded several examples considering various types of data to understand how it works. But if we have a chance to start this again, would like to try more complex examples involving multiple packages with support for both pre-defined and user-defined types and observe how ADA can deal with each of the types in a polymorphic fashion. As per Python's perspective, we have tried our best to realize the dynamic typing nature through scripting few examples and tried to emulate the implicit polymorphic behavior for deep understanding. In addition, concept of dynamic dispatching and redispatching can be further explored w.r.t class wide and tagged types in ADA and Python. In comparison section, along with syntax and mechanisms it would be better if we have assessed few extra parameters like run time and performance. This include checking how fast ADA and Python versions can achieve polymorphism either by increasing no of classes (sub type /inclusion polymorphism) or by adding multiple methods (function or method overloading) or by creating multiple generic functions(parametric) etc.,

References

[1] Computer programming. (2017). En.wikipedia.org. Retrieved 12 December 2017, from https://en.wikipedia.org/wiki/Computer_programmin

[2] Cite a Website - Cite This For Me. (2017). Citeseerx.ist.psu.edu. Retrieved 12 December 2017, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.454.6676&rep=rep1&type=pdf

[3] L. Cardelli and P. Wegner; On understanding types, data abstraction, and polymorphism, ACM Computing Surveys, vol. 17, no. 4, pp. 471-523, 1985.

[4] S. C. A. Barbey, "Working with Ada 9X classes," Proceedings of the conference on TRI-Ada 94 - TRI-Ada 94, pp. 129–140, Nov. 1994.

[5] Lutz, M. (2013). *Learning Python: Powerful Object-Oriented Programming*. " O'Reilly Media, Inc.".

[6] Python 3 OOP Part 4 - Polymorphism - The Digital Cat. (2017). Blog.thedigitalcatonline.com. Retrieved 12 December 2017, from http://blog.thedigitalcatonline.com/blog/2014/08/21/python-3-oop-part-4-polymorphism/

[7] Computer Programming/Function overloading - Wikibooks, open books for an open world. (2017). En.wikibooks.org. Retrieved 12 December 2017, from https://en.wikibooks.org/wiki/Computer_Programming/Function_overloading

[8] *args and **kwargs in python explained. (2017). Python Tips. Retrieved 12 December 2017, from https://pythontips.com/2013/08/04/args-and-kwargs-in-python-explained/

[9] Method overloading – Python Tutorial. (2017). Pythonspot.com. Retrieved 12 December 2017, from https://pythonspot.com/method-overloading/

[10] Python 3 – Function Overloading with singledispatch | The Mouse Vs. The Python. (2017).

Blog.pythonlibrary.org. Retrieved 12 December 2017, from https://www.blog.pythonlibrary.org/2016/02/23/python-3-function-overloading-with-singledispatch/

[11] Python3 Tutorial: Magic Methods. (2017). Python-course.eu. Retrieved 12 December 2017, from https://www.python-course.eu/python3_magic_methods.php

[12] Duck typing. (2017). En.wikipedia.org. Retrieved 12 December 2017, from https://en.wikipedia.org/wiki/Duck_typing

[13] A metrics-based comparative study on object-oriented programming languages. (2017). [online] Available at: https://pdfs.semanticscholar.org/5d8d/1dadd6bd5daf3dc9add997504cfacd9482f1.pdf [Accessed 9 Dec. 2017].