An Analysis on Programming Paradigms

Functional and Object-Oriented Programming

By: Tyler Conley

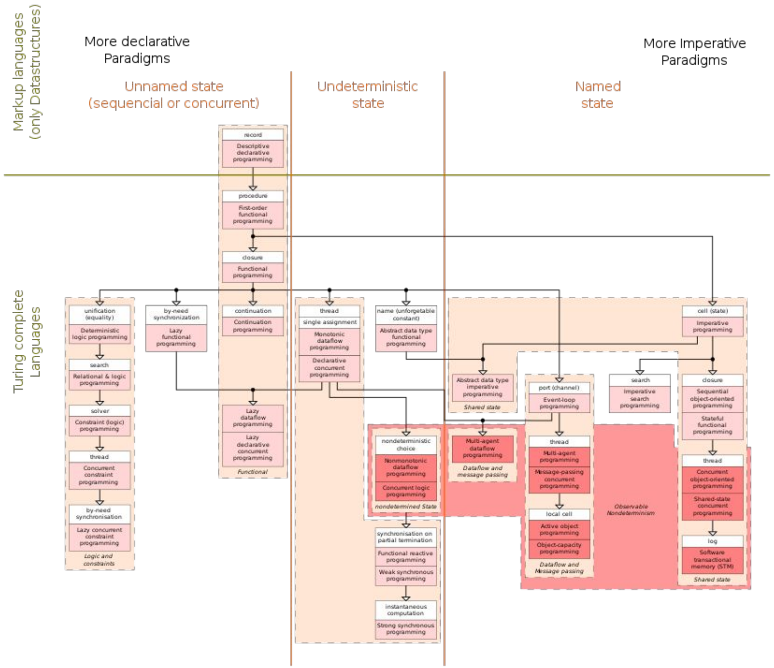
You can look at the code used in this presentation at https://github.com/tylerTaerak/CS4700_Final/

Programming Paradigms

- There are a huge amount of programming paradigms
  - Procedural
  - Imperative
  - Declarative
  - Functional
  - Object-Oriented

- This presentation will focus on Functional and Object-Oriented Programming

- The code examples will use Python, as it works well for both paradigms



Functional Programming

- A subset of declarative programming, revolving around developing functions to map values rather than changing variables
- Commonly uses lambda functions, functions as parameters, and recursion to make calculations
- The closest it gets to a class-like structure is through the use of simple data structures like dictionaries and arrays

Object-Oriented Programming

- Programming based entirely around objects, which consist of data and code
- Follows 4 key elements:
    - Data Abstraction
    - Encapsulation
    - Inheritance
    - Polymorphism

2 Examples

I have written up two simple examples to illustrate the strengths and weaknesses of each of these paradigms.

- The first is a simple RPG-style game consisting of heroes, monsters, and boss monsters, which will show strengths of OOP and fallbacks of FP
- The second is a reiteration of the merge sort algorithm that we worked on in Lisp, which will show strengths of FP and fallbacks of OOP

1. A Simple RPG: OOP

- Code found in ***ex1_RPG_OOP.py***
- The game consists of a Hero, a Monster, and a BossMonster, all of which inherit from a common Creature class
- The Hero and BossMonster classes have special abilities that only entities of that type can have

```python
class Creature:
    def __init__(self, name, damage, hp):
        self.name = name
        self.damage = damage
        self.hp = hp

    def logMessage(self, message):
        print(message)

    def attack(self, enemy):
        pass

class Hero (Creature):
    def __init__(self, name, damage, hp, heal):
        super().__init__(name, damage, hp)
        self.heal = heal
        self.canHeal = True

    def attack(self, monster):
        monster.hp -= self.damage
        self.logMessage(f"Hero {self.name} attacks {monster.name}: {self.damage} damage dealt!")

    def healSelf(self):
        if self.canHeal:
            self.hp += self.heal
            self.logMessage(f"Hero {self.name} heals: {self.heal} hp regained!")
        else:
            self.logMessage(f"Hero {self.name}'s healing is blocked!")
            self.canHeal = True

class Monster(Creature):
    def __init__(self, name, damage, hp):
        super().__init__(name, damage, hp)

    def attack(self, hero):
        hero.hp -= self.damage
        self.logMessage(f"{self.name} attacks Hero {hero.name}: {self.damage} damage dealt!")

class BossMonster(Monster):
    def __init__(self, name, damage, hp):
        super().__init__(name, damage, hp)

    def blockHeal(self, hero):
        hero.canHeal = False
        self.logMessage(f"{self.name} blocks Hero {hero.name}'s healing!")
```

1. A Simple RPG: FP

- Code found in ***ex1_RPG_FP.py***
- Still needs dictionaries to keep track of data (hp, attack, etc.)
- All functions are effectively agnostic of what is using them

```python
def logMessage(message):
    print(message)

def healSelf(hero):
    if 'heal' in hero and 'canHeal' in hero and hero['canHeal']:
        hero['hp'] += hero['heal']
        logMessage(f"Hero {hero['name']} heals: {hero['heal']} hp regained!")
    else:
        logMessage(f"{hero['name']} unable to heal!")
        if 'canHeal' in hero:
            hero['canHeal'] = True

def attack(entity1, entity2):
    entity2['hp'] -= entity1['attack']
    logMessage(f"{entity1['name']} attacks {entity2['name']}: {entity1['attack']} damage dealt")

def blockHeal(boss, hero):
    if hero['canHeal']:
        hero['canHeal'] = False
        logMessage(f"{boss['name']} blocks {hero['name']}'s healing!")
    else:
        logMessage(f"{boss['name']} cannot block {hero['name']}'s healing!")
```

Strengths of OOP

- Modular, reusable code
- Good methods of data encapsulation
- Types can have methods attached to them
- Polymorphism allows different types to act similarly but do different things

Fallbacks of FP

- Being type-agnostic means work needs to be done to support all types
- Data can’t really be kept very well
- Difficult to organize well, and sometimes difficult to read

2. Merge Sort: FP

- Code found in ***ex2_MergeSort_FP.py***
- It’s the same algorithm I used for the Lisp assignment
- A lot of the confusing code comes from lists of uneven lengths being merged
- Note that the code [left] + [right] appends the two lists together in Python

```python
@profile
def mergeSort(ls):
    if not isinstance(ls, list):
        return ls
    if len(ls) == 1:
        return ls
    return sortHelp(mergeSort(ls[:math.ceil(len(ls)/2)]), mergeSort(ls[math.ceil(len(ls)/2):]))

@profile
def sortHelp(left, right):
    if not left:
        if not right:
            return None
        return right
    if not isinstance(left, list):
        if not isinstance(right, list):
            if left < right:
                return [left] + [right]
            return [left] + [right]
        if left < right[0]:
            return [left] + right
        return [right[0]] + [left] + [right[1:]]
    if not right:
        return left
    if not isinstance(right, list):
        if left[0] < right:
            return [left[0]] + [right] + [left[1:]]
        return [right] + left
    if left[0] < right[0]:
        return [left[0]] + sortHelp(left[1:], right)
    return [right[0]] + sortHelp(left, right[1:])
```

2. Merge Sort: OOP

- Code found in ***ex2_MergeSort_OOP.py***
- This was a very difficult implementation
- The sort function takes two MergeSort objects, which are basically linked lists designed to be sorted
- The MergeSort.mergesort() method splits the linked list in half and passes both halves into the static MergeSort.sort(left, right) method

```python
@profile
def mergesort(self):
    if self.length == 0:
        return None
    if self.length == 1:
        return MergeSort([self.first.value])

    secondHalf = self.atindex(math.ceil(self.length/2), self.first)
    prev = secondHalf.left
    prev.right = None
    secondHalf.left = None
    part1 = MergeSort(self.aslist(self.first))
    part2 = MergeSort(self.aslist(secondHalf))
    self = MergeSort.sort(part1.mergesort(), part2.mergesort())
    return self

def aslist(self, node):
    if not node:
        return None
    ls = []
    while True:
        ls.append(node.value)
        node = node.right
        if node is None:
            break

    return ls

def rest(self):
    r = self.first.right
    if r:
        self.first.right = None
        r.left = None
    return r

def atindex(self, index, node):
    if index == 0:
        return node
    return self.atindex(index-1, node.right)

@staticmethod
@profile
def sort(left, right):
    if not left:
        if not right:
            return None
        return right
    if left.length == 1:
        if right.length == 1:
            if left.first < right.first:
                return MergeSort([left.first.value, right.first.value])
            return MergeSort([right.first.value, left.first.value])
        if left.first < right.first:
            return MergeSort([left.first.value] + right.aslist(right.first))
        return MergeSort([right.first.value, left.first.value] + right.aslist(right.rest()))
    if not right:
        return left
    if right.length == 1:
        if left.first < right.first:
            ms = MergeSort.sort(MergeSort(left.aslist(left.rest())), right)
            return MergeSort([left.first.value] + ms.aslist(ms.first))
        return MergeSort([right.first.value] + left.aslist(left.first))
    if left.first < right.first:
        ms = MergeSort.sort(MergeSort(left.aslist(left.rest())), MergeSort(right.aslist(right.first)))
        return MergeSort([left.first.value] + ms.aslist(ms.first))
    ms = MergeSort.sort(left, MergeSort(right.aslist(right.rest())))
    return MergeSort([right.first.value] + ms.aslist(ms.first))
```

Performance of the Merge Sort algorithm

FP:

- 0.212 seconds on average
- 19.75 MiB of memory used

```
Took 212.21485620002932 seconds to complete!

The algorithm took 0.2122148562000293 to complete on average


Filename: ex2_MergeSort_FP.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    13   19.750 MiB   19.059 MiB      199000   @profile
    14                                         def mergeSort(ls):
    15   19.750 MiB    0.000 MiB      199000       if not isinstance(ls, list):
    16                                                 return ls
    17   19.750 MiB    0.000 MiB      199000       if len(ls) == 1:
    18   19.750 MiB    0.000 MiB      100000           return ls
    19   19.750 MiB 1955178.398 MiB       99000       return sortHelp(mergeSort(ls[:math.ceil(len(ls)/2)]), mergeSort(ls[math.ceil(len(ls)/2):]))


Filename: ex2_MergeSort_FP.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    21   19.750 MiB 1955178.398 MiB      640363   @profile
    22                                         def sortHelp(left, right):
    23   19.750 MiB    0.000 MiB      640363       if not left:
    24   19.750 MiB    0.000 MiB       43346           if not right:
    25                                                     return None
    26   19.750 MiB    0.000 MiB       43346           return right
    27   19.750 MiB    0.000 MiB      597017       if not isinstance(left, list):
    28                                                 if not isinstance(right, list):
    29                                                     if left < right:
    30                                                         return [left] + [right]
    31                                                     return [left] + [right]
    32                                                 if left < right[0]:
    33                                                     return [left] + right
    34                                                 return [right[0]] + [left] + [right[1:]]
    35   19.750 MiB    0.000 MiB      597017       if not right:
    36   19.750 MiB    0.000 MiB       55654           return left
    37   19.750 MiB    0.000 MiB      541363       if not isinstance(right, list):
    38                                                 if left[0] < right:
    39                                                     return [left[0]] + [right] + [left[1:]]
    40                                                 return [right] + left
    41   19.750 MiB    0.000 MiB      541363       if left[0] < right[0]:
    42   19.750 MiB    0.000 MiB      278118           return [left[0]] + sortHelp(left[1:], right)
    43   19.750 MiB    0.000 MiB      263245       return [right[0]] + sortHelp(left, right[1:])
```

OOP:

- 0.376 seconds on average
- 21.297 MiB of memory used

```
Took a total of 376.1707735999589 seconds!

The algorithm took 0.37617077359995893 seconds to complete on average (including object creation)


Filename: ex2_MergeSort_OOP.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    67   21.297 MiB   19.547 MiB      199000       @profile
    68                                             def mergesort(self):
    69   21.297 MiB    0.000 MiB      199000           if self.length == 0:
    70                                                     return None
    71   21.297 MiB    0.000 MiB      199000           if self.length == 1:
    72   21.297 MiB    0.000 MiB      100000               return MergeSort([self.first.value])
    73
    74   21.297 MiB    0.000 MiB       99000           secondHalf = self.atindex(math.ceil(self.length/2), self.first)
    75   21.297 MiB    0.000 MiB       99000           prev = secondHalf.left
    76   21.297 MiB    0.000 MiB       99000           prev.right = None
    77   21.297 MiB    0.000 MiB       99000           secondHalf.left = None
    78   21.297 MiB    0.000 MiB       99000           part1 = MergeSort(self.aslist(self.first))
    79   21.297 MiB    0.000 MiB       99000           part2 = MergeSort(self.aslist(secondHalf))
    80   21.297 MiB 2107896.367 MiB       99000           self = MergeSort.sort(part1.mergesort(), part2.mergesort())
    81   21.297 MiB    0.000 MiB       99000           return self


Filename: ex2_MergeSort_OOP.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
   107   21.297 MiB 2107895.852 MiB      521287       @staticmethod
   108                                             @profile
   109                                             def sort(left, right):
   110   21.297 MiB    0.000 MiB      521287           if not left:
   111                                                     if not right:
   112                                                         return None
   113                                                     return right
   114   21.297 MiB    0.000 MiB      521287           if left.length == 1:
   115   21.297 MiB    0.000 MiB       79940               if right.length == 1:
   116   21.297 MiB    0.000 MiB       63783                   if left.first < right.first:
   117   21.297 MiB    0.000 MiB       31952                       return MergeSort([left.first.value, right.first.value])
   118   21.297 MiB    0.000 MiB       31831                   return MergeSort([right.first.value, left.first.value])
   119   21.297 MiB    0.000 MiB       16157               if left.first < right.first:
   120   21.297 MiB    0.000 MiB        5891                   return MergeSort([left.first.value] + right.aslist(right.first))
   121   21.297 MiB    0.000 MiB       10266               return MergeSort([right.first.value, left.first.value] + right.aslist(right.rest()))
   122   21.297 MiB    0.000 MiB      441347           if not right:
   123                                                     return left
   124   21.297 MiB    0.000 MiB      441347           if right.length == 1:
   125   21.297 MiB    0.000 MiB       55663               if left.first < right.first:
   126   21.297 MiB    0.000 MiB       36603                   ms = MergeSort.sort(MergeSort(left.aslist(left.rest())), right)
   127   21.297 MiB    0.000 MiB       36603                   return MergeSort([left.first.value] + ms.aslist(ms.first))
   128   21.297 MiB    0.000 MiB       19060               return MergeSort([right.first.value] + left.aslist(left.first))
   129   21.297 MiB    0.000 MiB      385684           if left.first < right.first:
   130   21.297 MiB    0.000 MiB      195497               ms = MergeSort.sort(MergeSort(left.aslist(left.rest())), MergeSort(right.aslist(right.first)))
   131   21.297 MiB    0.258 MiB      195497               return MergeSort([left.first.value] + ms.aslist(ms.first))
   132   21.297 MiB    0.000 MiB      190187           ms = MergeSort.sort(left, MergeSort(right.aslist(right.rest())))
   133   21.297 MiB    0.258 MiB      190187           return MergeSort([right.first.value] + ms.aslist(ms.first))
```

Strengths of FP

- Algorithms are super great to program functionally
- Functional programs typically avoid bloat (spaghetti code)
- Functional programs are typically simple and very to the point

Fallbacks of OOP

- Object-Oriented code can spaghetti out very easily
- For algorithmic calculations, adding classes/objects to the program often just add complexity rather than simplify through abstraction
- Creation and deletion of objects is pretty performance intensive

Conclusion

- Object-Oriented Programming is good to use when abstraction simplifies the program. So it is useful for when data needs to be kept, accessed, and modified in various ways
- Functional Programming is good to use when a the problem is easy to put into a step-by-step set of instructions. So it is good when working on an algorithm or similar rigid programming problem
- Functional Programming is faster and easier on memory, through I'm sure using a language other than Python would have saved me more performance than changing the paradigm