Functional Programming (In an oops culture)

What is functional programming (FP)

- It is a programming paradigm and not a programming technique
- Broadly speaking it’s the philosophy of programming which dictates that the building blocks of code should be “pure functions”
- The idea is to model computational problems in terms of mathematical formulation.
- This usually means writing code with functions which excel at doing one thing and doing it great.
- The data structures in functional programming are usually immutable

Pure functions (Functions with no side-effects)

- T square(T x) { return x*x; }
- bool find(Container<T> items, T item) { for (auto x : items) if (x == item) return true; return false; }
- void WillDoSomething(T *value) { *value = DidSomething(); }
- std::sort(T start, T end);
- T WillDoSomething() { return DidSomething(); }

(Functions in black font are "pure" while those in orange are not pure. Why?)

- Data structures in FP based code are mostly immutable.
- Immutable data structures + pure functions = Highly parallelizable code

Making containers parallelizable? Iterators

- Expose an iterator interface.

For eg all typical stl containers support iterator pattern.

```
bool Find(const Container<Type> &list,  const Type &t) noexcept
{
      for(const auto &x : list) { if (x==t) return true;}
      return false;
}
```

Notice this find is actually a “pure” function.

☺


1500 STARTING WATTS
VATIOS DE ARRANQUE
1200 RATED WATTS
VATIOS CONTINUOS
ENGINE SWITCH
ON
OFF
CHAMPION
GLOBAL POWER EQUIPMENT
VOLTMETER
120-VOLT AC
CIRCUIT BREAKER

Generators

- Generators are like iterators except that the items are generated on demand.
- (Python demo)
- A relatively good example in c++ of generators would be input iterators

```
while(cin >> value)
{
    // Do something
}
```

Although this does explains what generators are supposed to do, it doesn’t do justice to the power of generators.

The true power of generators

Problem : write a function that returns the sum of all primes <= N.

A trivial implementation would be:

```
int sumOfPrime(int N){
  if ( N <= 1) return 0;
   int sm = 2;
  for (int i = 3;i <= N;i+=2) {
        if (isPrime(i)){
                sm += i;
        }
   }
  return sm;
}
```

The Problem with this approach is that for each number i s.t. 0 <=i <= N one will necessarily have to make sqrt(i) iterations taking the worst case to O(N*sqrt(N))

```
void primeSieve(int N, function doOnPrimes){
    vector<bool> sieve(N + 1, 1);
    sieve[2] = true;
    doOnPrimes(2);
    for (int i = 3; i <= N; i+=2){
        if(sieve[i]){
                doOnPrimes(i);
            for(j=3*i; j <= N; j+=2*i) sieive[j] = false;
        }
    }
}
int sumOfPrimes(int N){
    if (N <=1 ) return 1;
     int sm = 0;
     primeSieve(N, [&sm](int prime){ sm+= prime; });
     return sm;
}
```

Using the sieve the asymptotic complexity remains the same but fewer iterations are made

But….

- Still not a good example of generator
- sumOfPrime needs to keep a state which sort of violates the principles of FP
- That is an important limitation for scalability

Yield keyword

```
generator<int> primeSieve(int N){
    yield 2;
    vector<bool> sieve(N + 1, 1);
     for (int i = 3; i <= N; i+=2){
          if(sieve[i]){
                    yield i;
                   for(j=3*i; j <= N; j+=2*i) sieve[j] = false;
           }
     }
    yield break;
}
```

```
int sumOfPrimes(int N)
{
    int sm = 0;
    for (auto var : primeSieve(N)) sm += var;
    return sm;
}
```

(Code shared here: https://github.com/bashrc-real/Codearchive/tree/master/YeildTry)

STAR WARS
RETURN OF THE FUNCTIONS

Lambdas and closures

- Lambdas are anonymous functions
- Used for returning from higher order functions* or to be passed to higher order functions
- Closures are lambdas with state

auto square = [](int x) { return x*x; }   -> Lambda

auto squareAndSum = [value](int x) { return (x*x) + value; } -> Closure

MapReduce

- Map: Given a list map each value to another value. The mapping function must not depend on the state of the list
- reduce : given a list transform the value to a single value
- For eg

Problem : Find the sum of all squares of [0,b]

List -> {0, 1, 2, 3…….b} [equivalent of vector<int> l(b); iota(l.begin(), l.end(), 0); }

Map-> [](int x) { return x*x; }

Reduce -> [](list values) { return accumulate(values.begin(), values.end(), 0); }

Combining the two:

transform(input.begin(), input.end(), output.begin(), [](int x) { return x*x; })
accumulate(output.begin(), output.end(), 0)

MapReduce(contd.)

- Notice that the steps of map reduce can be concatenated

Problem: Find whether a word exists among all words on the internet

Word: “Chewbacca”

MapReduce solution:
1. Define a map function such that [](string x) { int sm = 0; for(auto var:x) sm+=var; return {x, sm%MOD;} }

2. After running the map on all strings the result will look something like

   [DarthVader,7] [Yoda, 0] [Luke,5] [anakinSkyWalker,0]…..

Now we define a reduce function which returns a list of list of the form:

[0->{AnakinSkyWalker, Yoda}, 1->{HanSolo, …}, 2->{…},….]

3. We know the value of Chewbacca = 1 from the map function

4. We take the list with “key value” as 1

[0->{AnakinSkyWalker, Yoda}, 1->{HanSolo, …}, 2->{…},….]

5. We can now apply steps 1-4 each time with probably a new hash function and at each step after reduce the size of the list will keep getting smaller

6. Eventually the list will be so small that we can iterate through the list and match the strings character by character to our search string

7. Chewbacca does exist in star wars universe ☺

Conclusion

- Small stateless functions = testability, easier debugging, less bugs, such amaze much wow.
- Easy to scale and expand the program to take advantage of multi-cores without changing the logic
- More power to the compiler

References

- https://wiki.haskell.org/Functional_programming
- http://stackoverflow.com/questions/715758/coroutine-vs-continuation-vs-generator
- https://en.wikipedia.org/wiki/Higher-order_function