Week 2 – Sep 16 * 19

Lazy evaluation

Expression, methods and types

- **A need for expressions?**
  - Function vs method > difference? -> no
  - When it comes to push down computing, values
  - Source code of Scala library
    - X +: xs
    - xs map f: map f to evey elements that xs in the low level. They rather do integration than recursion, when it comes down to low-level, performance is prioritized
  - Scala came out when Java was 6 or 7, that solved most of Java inability to solve current problem at the time
    - At that time, Java could only loop over an array and iteratively updates its element through each iteration
- **Others should be parameterizable (where values depends on unknown expressions at the programming time)**
  - The ideas of parametric type
    - int x = 0, x = x + 1 - > x is an int
    - parametric type > type value > compile time
    - List[T], Array[T], Tree[T] map., we don't know T at the time of building the expression
    - Method(x) -> x is parametric value, and type[T] T is parametric type
    - List[T] does not become a concrete type until being declared as
      - val xs = List(1, 2, 3)
    - Most languages don't take parametric type T, in Java, List<T> where T might be integer
    - How about, List[Int] where Int is totally new to the compiler, not built-in ones
    - When run-time, JVM has no clue its integers, it's a bug not a feature for it not knowing the type
- **A need for types**
  - For example, when it comes to Complex numbers, they have 2 parts, imaginary, real
  - Tuple can have any number of components,
    - Let's say, (Boolean, Boolean)
    - Table of possible valus for this current type, where X represents row, Y represent columns

TT	TF
FT	FF

    - That's why they call Cartesian product, taking each vakue from each products
    - How about Wrapper? What if we don't have actual Boolean?
      - Fuzzy Logics or Kleenean Algebra or Three-value logics

	Y	N	Maybe
Y	Y	N	M
N	N	N	N
Maybe	M	N	M

- Type Comparison = Option[Boolean]
  - Some(true), Some(false), None
- We call it trait Option with parametric T: Option[T]

```
trait Option[T] {
        def get T
        def empty: Boolean
}
```

Examples:

- Case class Some[T](t: T) extends Option[T]

```
{
        def empty = false
        def get T = t
        }
```

Case object None extends Optional[None]

```
{
        def empty = true
        def get T throws Exception
        }
```

- **Expression continued..**
  - Alonzo Clumich 1936 in “Lambda Calculus”

$$lambda.x\ x * x$$

  - Parallelism and concurrency

- Parallel processing	- Concurrency
- Divive and conquer -> unlimited recursion - Map/recude -> need to join threads tgt - Dynamic programming -> need efficient lazy evaluation	- Pub/sub (publish/ subcribe) -> message passing and queuing, in generals, for independent topis; - Actor model - Mutex, locks, synchronization -

  - Talking about parallelism, In 2000, Google made Map Reduce concept later took up by Hadoop

- For concurrency, we might have some limited set of stuffs like tickets, which are unique instance, make sure don't sell same ticket to another people, where parallel processing, data in one partition might not affect other in other partition.
- That's why it leads to Actor model (basically pub/sub with more roles)
- Mutex, locks, synch works at low level dealing with threads (really hard to get it right)

- **Communication**
  - How such threads communicate (through sockets, aka primary mode of inter-process communication). But sockets only know bits and bytes, don't know any other types
  - => a need for serialization and deserialization => a need for type
    - Encoding and decoing of type. Ex: List[String] => thiss a new type, List[String] is not something JVM knows, the other ends who receive also need to understand these
      - => a need for serializing type too
  - How we can achieve this?
- **Pushdown computing**
  - What does this mean?
    -
    - Have two functions that want to apply

      f1: multiply by two, f2: add one => at some points, these will become costly, that takes twice as long as just applying 1 function, which combines two function into one

Lambda calculus	**Function**
lambdax. x*2	f1: x=> x*2
lambda x. x+1	f2: x=> x+1
	val y = f1(x) val z = f2(y)

  - **In Scala,** combined function can be achieved using
    - val g = f1 andthen f2
    - val h = f1 compose f2

Q&A session

Q: Which one, Java or Scala?

- Purely function language like Haskell, and Java, what happens if you combine the two. Need to decide based on the tradeoffs between the twos (or more)
  - Example: m(x,y)

Java	**Scala**
m(x, y) what if x is a class, has a class method m ➔ x.m(y)	m(x, y) x m y ➔ this m y