Oscar

Functional, Actor-based Programming Language

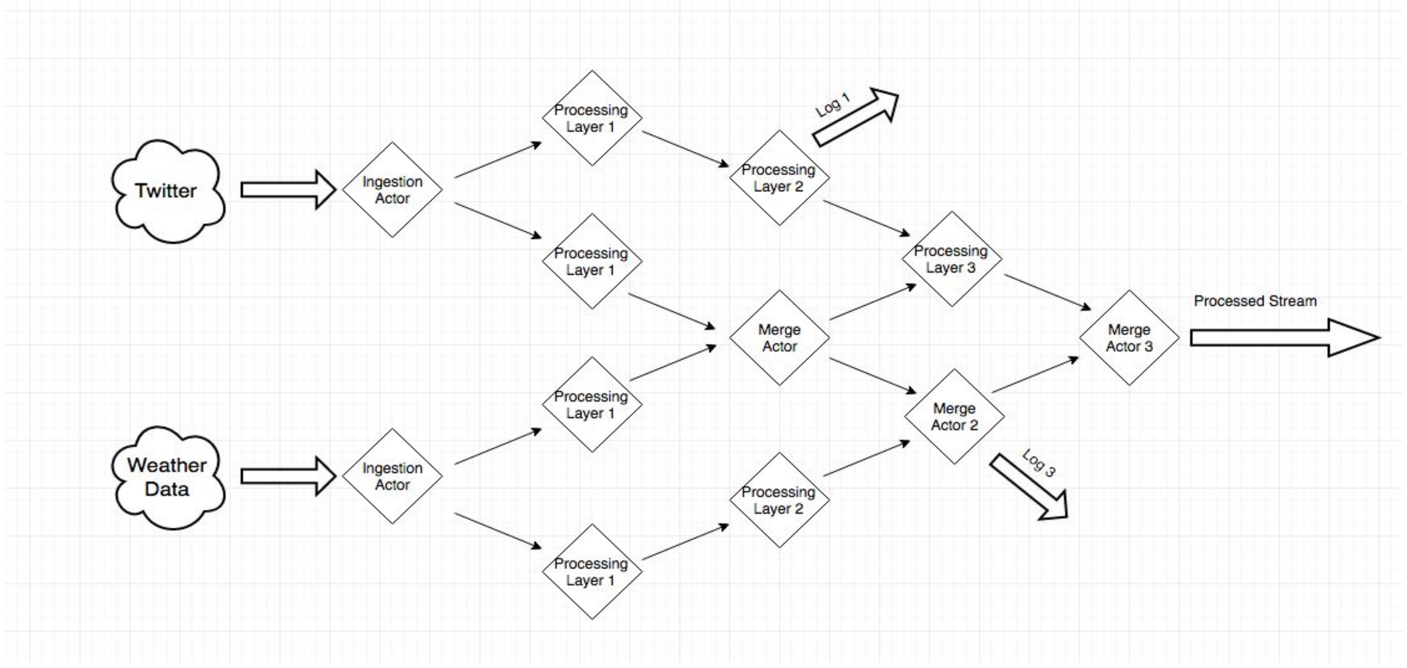
Manager: Ethan Adams EA2678
Language Guru: Howon Byun HB2458
System Architect: Jibben Lee Grabscheid Hillen JLH2218
System Architect: Vladislav Sergeevich Scherbich VSS2113
Tester: Anthony James Holley AJH2186

Goal

We will be implementing an Actor-Oriented Language. We all know there are difficulties associated with parallel computation, namely data collision when multiple threads access the same data. Most often, this requires explicitly defining locks and mutexes to control data access to resolve this issue, which adds significant mental and code overhead. The Actor model is a different approach to this problem that relies on the concept of message-passing between processes. The model simplifies the aforementioned difficulties of multiprocessing by abstracting explicit method calls, and relying on each “actor” to handle tasks and encapsulate mutable data. We would like to construct our language based off Erlang and Akka framework available on JVM to make it easy and safe to write code for parallel computing.

Applications

The main application we aim to solve is that of stream processing. By setting up a network of actors, one can process multiple streams. For example, we can process streams as such.



We can ingest multiple streams of data from different sources, manipulate them in parallel, and pipe them through various transformations. Since our language is pure and functional, we can easily reason about dataflow and trace operations in the case of an unexpected result. Since actors interact through message passing and any side effects from outside actors are prevented, we can attach logging pipes to keep track of various stages without altering the rest of data flow.

Syntax

```
/*
 * C-style multi-line comments
 */
// used for single line comment

// let is used to declare immutable constants
// values can be declared as such with types inferred
let a = 5
a = 7 // ERROR as assignment to an immutable constant
let a = 7 // ERROR as a was declared already
/*
 * all functions are declared using arrow syntax
 * argument type declaration is necessary
 */
let addTwoNums = (a: int, b: int) -> int => a + b
let b = addTwoNums(a, 6) // b == 11

// another example
let applyFunc = (f: (double) -> double, a: double) : double => {
  // arrow syntax to declare function argument f's signature
  return f(a) // return is needed for multi-line functions
} // multi-line functions are surrounded with brackets

// lambda expressions
let d = applyFunc(x => x * 2, 44.5) == 89.0

// no argument functions are declared like this
let sayHi = () -> () => print("hi!") // void function
sayHi() // prints "hi!"

// tuples are first class objects just like fun ctions and actors
// tuples are declared with parentheses
// a single tuple can contain multiple types
let tup = (1, 2, "what")
let tup4 = (1, 2, “what”, 4.5)
// tuples can also be destructured into a series of values
let a, b, c = (1, 2, “what”) // a == 1, b == 2, c == “what”
let printTriple = ((x: int, y: int, z: int)) -> () => {
  print(x + y + z)
}

/*
 * lists are first class object. There are  no arrays
 * These are immutable. To guarantee performance,
 * lists are implemented as Hash-Array-Mapped-Trie [1] in the backend
 * lists can only hold a single type
 */
let intList = list<int>() // empty list of type int
let listSizeTen = list (10, 0.0)
let intList = [1, 2, 3, 4, 5]// used to declare list literal
let intList = [1, 2.0, 3, 4, “5”]// ERROR

// assignment returns a new copy of the list
let changedList = (initList[3] = -4)
changedList == [1, 2, 3, -4, 5]
initList == [1, 2, 3, 4, 5]

// lists support map, reduce/fold and filter
initList.filter(x => x >= 3) // == {3, 4, 5}
initList.map(x => x + 3) // == {4, 5, 6, 7, 8}

// use list comprehension for looping
[i <- start to end].foreach(i => print(i))

[i <- 0 to 5 by 2].map(i => {
  return = i + 4
}).reduce((x, y) => x + y) // == 18
```

[1] http://hypirion.com/musings/understanding-persistent-vector-pt-1

```
// message construct dedicated for passing messages to actors
// unlike tuples, these hold reference to  sender actor
message message(msg: string, payload: double)

// actors are first class data types in our lan guage
// mutables are declared with  mut keyword inside actors.
actor Actor(initVa lue: int) { // takes in a value
  mut x = 10 // allowed within actors
  x = 14 // x == 14 now since x was declared to be mutable
}

mut y = 5 // ERROR  as it is declared outside an actor

actor Worker(name:  String) {
  let receive => { // all actors must have  receive method defined
    | messageType1(name: string)  => // do something 1
    | end() => die() // used to kill this actor and
                     // all other workers declared in this actor
    }
}

// creates a new worker with name  worker

let botbot =  spawn Worker(“worker”)

// define a message type that contains two strings
message helloMessa ge(prefix: string, suffix: string)
//  |> is used to send a message. Note that “new ” is not needed
helloMessage(“Hello”, “ World!”)  |> botbot
// sender keeps track of the source actor of a  message
helloMessage(“hi”, “there”)  |> sender

// pool manages a collection of actors. Use  spawn to open a pool.
// Workers do not have to be spawned individually.
let workerPool =
  spawn pool (Worker(“multiple”), 10) // puts ten workers into a pool
let workerPool2 = // another example of spawning pool
  spawn pool (botbot, Worker(“worker-3”), Worker(“worker-4”))

// broadcasts a message to a single worker in a pool using   |>
// messages passed into a pool is distributed in round-robin fashion
// this is useful for cases where values are accumulated at the end
helloMessage("just one", " message")  |> workerPool
// broadcasts this message to all workers in this pool using  |>>
// useful for cases like streams
helloMessage("everyone gets a", " message!")  |>> workerPool
// list of messages can be mass broadcasted
list(msg1, msg1, msg1) |>> workerPool
```

Key words

`let`	Declaration for immutable data
`mut`	Declaration for **mutable** data (within actors only)
`actor`	Actor as a first class object
`die`	Kills the actor and all actors spawned by this actor
`spawn`	Used to spawn actors
`pool`	Construct used to manage a collection of workers
`sender`	Source worker of the message
`message`	Used to communicate between actors and pools
`|>`	Used to send a message to a worker or into a pool in round-robin fashion
`|>>`	Used to broadcast a message or a list of messages into a pool
`return`	Return
`to/by`	For loop range and loop increment/decrementer
int	Integer data type
double	Double data type used for floating point arithmetics
char	Characters
bool	Boolean data type. Lowercase true/false please. (Please no python)
string	String
maybe/none/some	Optional type. "Null" should be wrapped with none in our language
list	List backed with HAMT[2] for persistence with performance. Remember, no arrays

[2] http://lampwww.epfl.ch/papers/idealhashtrees.pdf

Calculation of Pi

Program inspired by http://doc.akka.io/docs/akka/2.0/intro/getting-started-first-scala.html

```
message start() // empty message
message end()
message work(start: int, numElems: int)
message result(value: double)
message piApproximation(pi: double)

// Master worker for pi approximation
actor Master(numWo rkers: int, numMsgs: int, numElems: int) {
  mut pi = 0.0
  mut numResults =  0
  mut workerPool =  spawn pool() // empty pool
  let listener =  spawn Listener(“pi listener”)

  let receive => {
    | start() => {
        workerPool =  spawn pool(Worker, numWorkers)
        let msgList = [i <- 0  to numMsgs].map(i  => {
          work(i * numElems, numElems)
        })
        msgList  |>> workerPool
      }
    | result(value: double)  => {
        pi = pi + value
        numResults = numResults + 1
        if (numResults == numMsgs)  {
          piApproximation(pi)  |>> listener
        }
      }
    | end() => {
        // automatically closes actors/pools within this actor
        die()
      }
  }
}
```

```
// Listener actor listens for final computed value of pi
actor Listener(name: string) {
  let receive => {
    | piApproximation(value: double) => {
        print("value of pi is approximately :" + value)
        end()  |> sender
      }
  }
}

// Pi calculation actor
actor Worker {
  let calcPi = (start: int, numElems: int) : double = {
    return [i <- start to (start + numElems)].map(i => {
      return 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)
    }).reduce((x, y) => x + y) // list operations
  }

  let receive => {
    | work(start: int, numElems: int) => {
        result(calcPi(start, numElems))  |> sender
      }
  }
}

let main = () => {  // main is a keyword
  spawn Master(5, 10000, 10000)
}
```