COMP 311
Functional Programming

Coroutines

Shams Imam, Rice PhD
Two Sigma Investments, LLC
Review: Subroutines (aka Functions)

- A block of executable code
- Exactly one point of entry
- Once a subroutine exits, it is done
Review: Subroutines relationships

• A subroutine may call another subroutine
• Starts a caller-callee relationship
  – Control transferred to the entry point of callee
  – Callee local data created from scratch
  – Callee runs to completion and returns
  – Caller resume computation from call site
Review: Subroutines control flow
Review: Subroutines Example

```scala
object ProducerConsumerSubroutine {
  def main(args: Array[String]) {
    var (itemsConsumed, consumerResult) = (0, 0L)
    val numItems: Int = 10
    val queue = new util.LinkedList[Long]()
    for (i <- 1 to numItems) {
      producer(numItems, i, queue)
      val (a, b) = consumer(queue)
      itemsConsumed = a
      result = b
    }
    println("Items Consumed = " + itemsConsumed)
    println("Sum = " + consumerResult)
  }
}
```
Review: Subroutines Example

object ProducerConsumerSubroutine {

  ...

  def producer(numItems, itemIndex, queue) = {
    if (itemIndex >= numItems)
      queue.offer(-1)
    val item = 1L * itemIndex
    queue.offer(item)
  }

  def consumer(queue) = {
    val item = queue.poll()
    var (itemsConsumed, itemsSum) = (0, 0L)
    if (item != -1) {
      itemsConsumed += 1
      itemsSum += item
    }
    (itemsConsumed, itemsSum)
  }

  ...
}
Review: Subroutines relationships

• Caller-callee relationship
  – Control transferred to the entry point of callee
  – Callee local data created from scratch
  – Callee runs to completion and returns
  – Callee local data is destroyed
  – Caller resume computation from call site

• If Caller calls Callee again, whole process is repeated
object ProducerConsumerSubroutine {

  ... 
  private var (itemsConsumed, itemsSum) = (0, 0L)
  def producer(numItems, itemIndex, queue) = {
    if (itemIndex >= numItems)
      queue.offer(-1)
    val item = 1L * itemIndex
    queue.offer(item)
  }
  def consumer(queue) = {
    val item = queue.poll()
    if (item != -1) {
      itemsConsumed += 1
      itemsSum += item
    }
    (itemsConsumed, itemsSum)
  }
}
Imagine a procedure that "remembers" its state across calls
Example control flow
Coroutines

• A block of executable code
• Exactly one point of entry
• Coroutines can exit by calling other coroutines
  – Typically using the **yield** statement
  – Yield indicates that the routine is done executing for now
  – Coroutine may be resumed from the yield point
• **One or more** points of re-entry
Coroutines (contd)

• Allow for **suspending** and **resuming** execution at yield points

• Coroutines hold state between invocations
  – parameters and local variables are preserved between invocations
  – Nested call chains
Coroutines Example

```scala
object ProducerConsumerCoroutine {
  def main(args: Array[String]) {
    val numItems: Int = 10
    val queue = new util.LinkedList[Long]()

    runCoroutines("producer", () => {
      coroutine("producer", () => producer(numItems, queue))
      coroutine("consumer", () => consumer(queue))
    })
    // wait for one of the registered coroutines to return
    val (itemsConsumed, result) = coroutineResult("consumer")

    println("Items Consumed = " + itemsConsumed)
    println("Sum = " + result)
  }
  ...
}
```
Coroutines Example

```scala
object ProducerConsumerCoroutine {
...
  def producer(numItems: Int, queue: util.Queue[Long]) = {
    var itemIndex = 1
    while (itemIndex <= numItems) {
      queue.offer(itemIndex); yieldTo("consumer")
      itemIndex += 1
    }
    queue.offer(-1); yieldTo("consumer")
  }
  def consumer(queue: util.Queue[Long]): (Int, Long) = {
    var (itemsConsumed, itemsSum) = (0, 0L)
    var item = queue.poll()
    while (item != -1) {
      itemsConsumed += 1; itemsSum += item
      yieldTo("producer")
      item = queue.poll()
    }
    (itemsConsumed, itemsSum)
  }
}
```
Class Exercise:
Write Code for Example control flow

A() → 1 → 6
B() → 2 → 3 → 8
C() → 4 → 5
Iterator Example

```scala
object FibonacciGeneratorCoroutine {
  def printFib(numItems: Int) = {
    var itemIndex = 1
    while (itemIndex <= numItems) {
      yieldTo("fib")
      itemIndex += 1
    }
  }
  def fib() = {
    var f1 = 1; println(f1); yieldToCaller()
    var f2 = 1; println(f2); yieldToCaller()
    while (true) {
      val f3 = f1 + f2; f1 = f2; f2 = f3
      println(f3); yieldToCaller()
    }
  }
  ...
}
```
Observation

- Any subroutine can be translated to a coroutine which does not call `yield`.
- Coroutines are more general than subroutines!
Implementation Details

• Rely on Scala’s support for Delimited Continuations using shift/reset (http://infoscience.epfl.ch/record/149136/files/icfp113-rompf.pdf)

• Taught in COMP 411: Continuations and Continuation-passing style transforms
Current motivations for Coroutines

• Mainly in the Concurrency/Parallelism world
  – Use coroutines to build efficient runtimes
• Overcoming the limitations of a single-threaded process
• Achieve better computational performance
Concurrent/Parallel Programming

- Most current runtimes rely on O/S-level threads to execute work in parallel
- Ideally execute one-thread (worker) per core
- No overheads from thread context switches
Issues with OS Threads Blocking Operations

• When worker encounters blocking operation =>
  – Spawn another worker to maintain parallelism
• E.g. One thread each for the producer and consumer
• Not scalable when we have hundreds of interacting producers and consumers!
Concurrent Programming

• Coroutines as user-level threads
  – Another level of abstraction
  – Process => OS Threads => User-level threads
  – Context switch of coroutines is much cheaper

• Concurrent Scheduler
  – Manages interactions between coroutines
  – Determines when to resume coroutines
Learn more about use of Coroutines in COMP 322

• Habanero-Java library uses Coroutines to implement its Cooperative runtime
• Users write programs unaware of presence of Coroutines
  – Compiler and runtime uses Coroutines behind the scenes
Coroutines Performance Gains

Benchmark Name with HJCOOP time in secs below

JUCBLK  HJBLK  HJCOOP
Acknowledgments

- [https://en.wikipedia.org/wiki/Coroutine#Comparison_with_subroutines](https://en.wikipedia.org/wiki/Coroutine#Comparison_with_subroutines)