Practical Concurrent and Parallel Programming X

Coroutines I

IT University of Copenhagen

Friday 2020-10-29

Kasper Østerbye

Starts at 8:00

Plan for today

Follow up on last week

What is a coroutine and why are they useful

Ultra brief intro to kotlin

- the parts we need to concentrate on coroutines

Coroutines

Follow up and loose ends

Exam

Written exam, take home from

Different kinds of concurrency

- **Parallelism**: multiple threads running at the same time at different cores
- **Concurrency**: multiple threads - perhaps through interleaving - interleaving necessary if more threads than cores
    - **preemptive concurrency** - the scheduler stops the thread and and resumes an other (thread states running vs. runnable)
    - The word *preemptive* means to take action to prevent something bad from happening - here - that one thread monopolize a core
    - cooperative/non-preepmtive concurrency - it is the responcibility of the thread to change state from running to runnable

Cooperative concurrency - coroutines

Usage scenarios:

- generators in streams
- suspending while waiting for something else (e.g. react style waiting)
- message based concurrency (Comunicating Sequential Processes)
- simulation (original usage), custom scheduling, …

Focus this week

- coroutine implementation
- coroutines in yield

Section - kotlin

Why kotlin?

- It is becoming popular - driven by Android development, but also for other applications
- It supports coroutines in a general form (not just yield and async)
- It is a full modern programming language
- I (Kasper) think it is well designed - good balance between usefulness and sound language design

To fully get the examples about coroutines, we need to look at a few kotlin of kotlins language constructs

Hello world

var vs. val

null safety

higher order functions

Lambda

Extension functions

See in particular the page on functions and lambdas

Sequences (somewhat like Java streams)

```
fun main() {
    val seq : Sequence<Double> = sequenceOf(10.3, 10.9, 11.2, 9.3, 9.7, 10.4)
    println("Sequence: $seq")
    println("Sequence as list ${seq.toList()}")
    println("Length: ${seq.count()}")
    println("Sum: ${seq.sum()}")
    println("Sum of squares: ${ seq.map{it*it}.reduce{a,b->a+b} }")
}
```

[Example in kotlin playground]

Notice that the sequence is immutable - unlike Java streams.

That is, each expression using the sequence gets the whole sequence, unlike in java, where you have to create the stream from the underlying data again.

Walking through a sequence

The mutable version of a sequence is an iterator.

```
fun main() {
    val seq : Sequence<Double> = sequenceOf(10.3, 10.9, 11.2, 9.3, 9.7, 10.4)
    val iterator = seq.iterator()
    while (iterator.hasNext()){
        println("Value: ${iterator.next()}")
        if (iterator.hasNext()) iterator.next()
    }
}
```

[Iterators of streams]

Section - yield

```
fun main() {
    val sequence = sequence {
        // yielding a single value
        yield(0)
        // yielding an iterable
        yieldAll(1..5 step 2)
        // yielding an infinite sequence
        yieldAll( generateSequence(8) { it * 3 } )
    }

    println( sequence.take(8).toList() )
}
```

[In playground]()

Notice: the yield must be used withing a `sequence` call.

Remember `sequence {...}` is really just a shortcut for `sequence( {...} )`

The ubiquitous primes

range → filter2 → filter3 → filter5 → filter7 →

```
fun main() {
    (2..20).asSequence().onlyPrimes().forEach(::println)
}

fun Sequence<Int>.onlyPrimes(): Sequence<Int> = sequence {
    if ( any() ){
        yield( first() )
        val morePrimes = filter { i -> i % first() != 0 }.onlyPrimes()
        yieldAll( morePrimes )
    }
}
```

[Playground]()

We are extending `Sequence<Int>` with a method `onlyPrimes()`.

- Inside the extension, one can use member functions from `Sequence` - like `any()`, `first()`,`filter()`, and `onlyPrimes()` itself

Prime factorization

```
fun main() {
    630.factorsOf().forEach(::println)
}
```

```
fun Int.factorsOf(): Sequence<Int> = sequence {
    var n = toInt()
    twoToSqrtOf(n).onlyPrimes().forEach { prime ->
        while (n % prime == 0){
            yield(prime)
            n /= prime
        }
    }
    if ( n != 1 ) yield(n)
}
```

Again, an extension method, this time on integers.

[Playground]

```
fun twoToSqrtOf(n:Int): Sequence<Int> = sequence {
    var i = 2
    while (i*i <= n) yield( i++ )
}
```

Yield in other (popular) languages

The good

- C# (and all the .net languages I think) - has yield in one form or the other
- Python - since 3.7.X

The bad

- Java 14 - where it does something completely different from any other programming language (switch expression)
- ruby - where is means calling an implicitly passed lambda - called `apply` in other languages

The rest

Programming languages with native support [edit]

Coroutines originated as an assembly language method, but are supported in some high-level programming languages. Early examples include Simula,[8] Smalltalk, and Modula-2. More recent examples are Ruby, Lua, Julia, and Go.

- Aikido
- AngelScript
- Ballerina
- BCPL
- Pascal (Borland Turbo Pascal 7.0 with uThreads module)
- BETA
- BLISS
- C++ (Since C++20)
- C# (Since 2.0)
- ChucK
- CLU
- D
- Dynamic C
- Erlang
- F#
- Factor
- GameMonkey Script
- GDScript (Godot's scripting language)
- Go
- Haskell[9][10]
- High Level Assembly[11]
- Icon
- Io
- JavaScript (since 1.7, standardized in ECMAScript 6)[12] ECMAScript 2017 also includes await support.
- Julia[13]
- Kotlin (since 1.1)[14]
- Limbo
- Lua[15]
- Lucid
- µC++
- MiniD
- Modula-2
- Nemerle
- Perl 5 (using the Coro module)
- PHP (with HipHop, native since PHP 5.5)
- Picolisp
- Prolog
- Python (since 2.5,[16] with improved support since 3.3 and with explicit syntax since 3.5[17])
- Raku[18]
- Ruby
- Rust (since version 1.39 based on async_std or tokio)
- Sather
- Scheme
- Self
- Simula 67
- Smalltalk
- Squirrel
- Stackless Python
- SuperCollider[19]
- Tcl (since 8.6)
- urbiscript

Since continuations can be used to implement coroutines, programming languages that support them can also quite easily support coroutines.

From wikipedia - which for this particular topic seems a bit wrong in my oppinion

Other non-preemptive constructs

Goroutines

Lightweight threads which are not mapped to operating system threads

Fibers

Seems to be adopted as a concept for "threads not implemented using operating system threads"

Actors - (briefly) next week

Simula 1968 - a brief historical aside

```
begin

    class point(x, y); real x, y;
    begin
        real r;
        r := sqrt(x**2 + y**2);
    end***point***;

    class Foo;
    begin
        outtext("Pre foo"); outimage;
        inner;
        outtext("Post foo"); outimage;
    end***Foo***;

    ref(point) p;
    p :- new point(4,4);

    Foo begin
        outtext("Hello world: ");
        outfix(p.r, 2    , 8);
        outimage;
    end

end
```

Simula coroutines

```
Demos begin
    ref(resource) tugs, jetties;

    entity class boat;
        begin
            jetties.acquire(1);
                tugs.acquire(2);
                    hold(2.0);
                tugs.release(2);
                hold(14.0);
                tugs.acquire(1);
                    hold(2.0);
                tugs.release(1);
            jetties.release(1);
        end***boat***;

    jetties :- new resource("jetties", 2);
    tugs    :- new resource("tugs"   , 3);
    new boat("boat").schedule(0.0);
    new boat("boat").schedule(1.0);
    new boat("boat").schedule(15.0);
    hold(36.0);
end;
```

Section - Implementation

Normal stack based method calls

Demo

Coroutines

One way of understanding a coroutine is as "a reference to a stack-frame".

Plus two operations:

- suspend - called from **within** the coroutine to return to its caller/attacher
- call/attach - called to reactivate a suspended coroutine

In Simula constructors were coroutines, and member methods/functions were normal functions

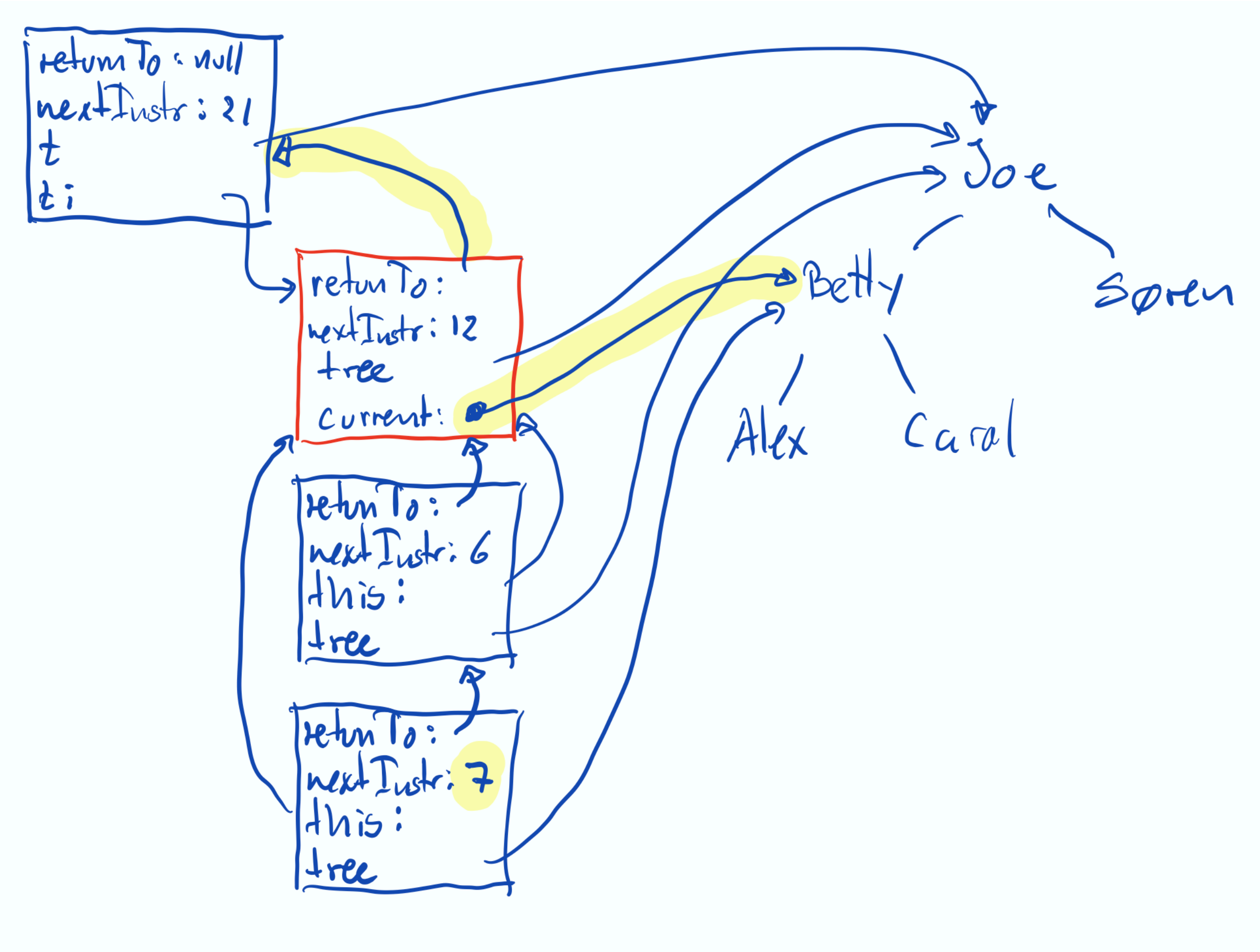
Simula coroutines

```
class TreeIterator(tree) begin
 ref Value current

 procedure traverse(tree) begin
   if tree.left != null then traverse( tree.left )
   current :- tree.value
   detach
   if tree.right != null then traverse ( tree.right )
 end

 travese( tree )
 current :- null
end

begin main class
  ref Tree t :- build wonderful tree
  ref TreeIterator ti :- TreeIterator(t)
  while ti.current != null begin
    print ti.current
    resume (ti)
  end
end
```

Just before first detach

Just after first detach

Just before second detach



Implementation problems

In essence we need one stack per coroutine (if no detach is called when the object is created, no stack is needed).

However, this is the approach (one stack per coroutine) which is being revisited by goroutines and fibers.

State machine implementation

```
val a = a()
val y = foo(a).await() // suspension point #1
b()
val z = bar(a, y).await() // suspension point #2
c(z)
```

There are three states for this block of code:

- initial (before any suspension point)
- after the first suspension point
- after the second suspension point

Pseudo java implementation

```
class <anonymous_for_state_machine> extends SuspendLambda<...> {
    // The current state of the state machine
    int label = 0
    // local variables of the coroutine
    A a = null
    Y y = null

    void resumeWith(Object result) {
        if (label == 0) goto L0
        if (label == 1) goto L1
        if (label == 2) goto L2
        else throw IllegalStateException()
      L0: // result is expected to be `null` at this invocation
        a = a()
        label = 1
        result = foo(a).await(this) // 'this' is passed as a continuation
        if (result == COROUTINE_SUSPENDED) return // return if await had suspended
      L1: // external code has resumed this coroutine passing the result of .await
        y = (Y) result
        b()
        label = 2
        result = bar(a, y).await(this) // 'this' is passed as a continuation
        if (result == COROUTINE_SUSPENDED) return // return if await had suspended
      L2: // external code has resumed this coroutine passing the result of .await
        Z z = (Z) result
        c(z)
        label = -1 // No more steps are allowed
        return
    }
}
```

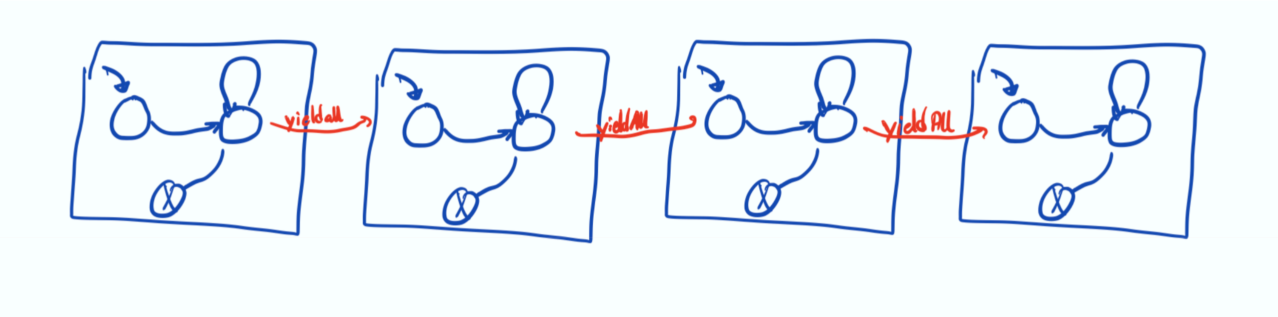
Suspending functions

If a function itself can suspend, it is also translated into a state-machine object.

Our primenumber filter:

```
fun Sequence<Int>.onlyPrimes(): Sequence<Int> = sequence {
    if ( any() ){
        yield( first() )
        val morePrimes = filter { i -> i % first() != 0 }.onlyPrimes()
        yieldAll( morePrimes )
    }
}
```

Which in essence turns into:
(because the `sequence` function is turned into a state-machine):



Sequence builder

```
sequence {
  yield(0)
  yieldAll(1..5 step 2)
  yieldAll( generateSequence(8) { it * 3 } )
}
```

This is translated into (in principle - the compiler does it a bit differently):

```
class MySequence : Iterator<Int> {
  val itr1 = (1..5 step 2).iterator()
  val itr2 = generateSequence(8) { it * 3 }.iterator()
  var state : Int = 0; // state of state machine
  override hasNext(): Boolean { return state != -1 }
  override next() : Int {
    var ret = 0
    when(state){
      0 -> ret = 0; state = 1
      1 -> if (itr1.hasNext() ) ret = itr1.next()
           else { ret = itr.2.next(); state = 2 }
      2 -> if (itr2.hasNext() ) ret = itr1.next()
           if (!itr2.hasNext() ) state = -1
     }
    return ret
  }
}
```

It is all in a library.

Close

Kotlin, like most other languages has labels, used in connection with loops:

```
loop@ for (i in 1..100) {
        for (j in 1..100) {
            if (...) break@loop
        }
}
```

> A `break` qualified with a label jumps to the execution point right after the loop marked with that label. A `continue` proceeds to the next iteration of that loop.

but no cigar

Consider this (*illegal*) program:

```
fun foo() : () -> Unit { // returns a lambda expression
    here@ for( i in 1..10 ){
        println("i is currently $i")
        return { continue@here }
    }
    return {}
}

fun main() {
  var bar : () -> Unit = foo()
  bar(); bar(); bar()
}
```

In most programming languages with labels, this is not allowed. It would in effect turn `foo` into a coroutine.

It is not possible in kotlin either.

Next week

- Async
- React for Kotlin
- Message based concurrency
- Threads and Coroutines