Scala Coroutines

Aleksandar Prokopec Fengyun Liu

Wait, but why?

With the rise of asynchronous computing, programmers need a more powerful, language-level coroutine model.

So far, JVM, .NET or V8 did not provide a runtime-level solution.

Yet another model?

Scala Coroutines are simpler to understand and use compared to alternatives such as delimited continuations or algebraic effects, but they have a similar expressive power.

But, when is it useful in practice?

Scala Coroutines unify many other control constructs for suspension.

A few examples: iterators for arbitrary data structures, Erlang-style actors, Rx-style event streams without callbacks, Oz-style single-assignment variables, Esterel-style pause statement, QuickCheck-style testing without monads, backtracking, continuations… list goes on!

Example Use Cases

Iterators

Given a higher-order foreach statement, an iterator is for free.

```
val ply = coroutine { (t:Tree)=>
  if (t.fst != null) ply(t.fst)
  yieldval(t.element)
  if (t.snd != null) ply(t.snd)
}
```

Actors

No need for top-level event loops and finite state machines.

```
val pass = receive()
assert(thouShallNot(pass))
while (true) receive() match {
  case Get(url) => serve(url)
  case Logout() => stop()
}
```

Event streams

Direct-style event stream handling is simpler than using combinators.

```
var e = mouse.get
while (!e.isDown)
  e = mouse.get
val c = new Curve(e.x, e.y)
while (e.isDown) {
  e = mouse.get
  c.add(e.x, e.y)
}
```

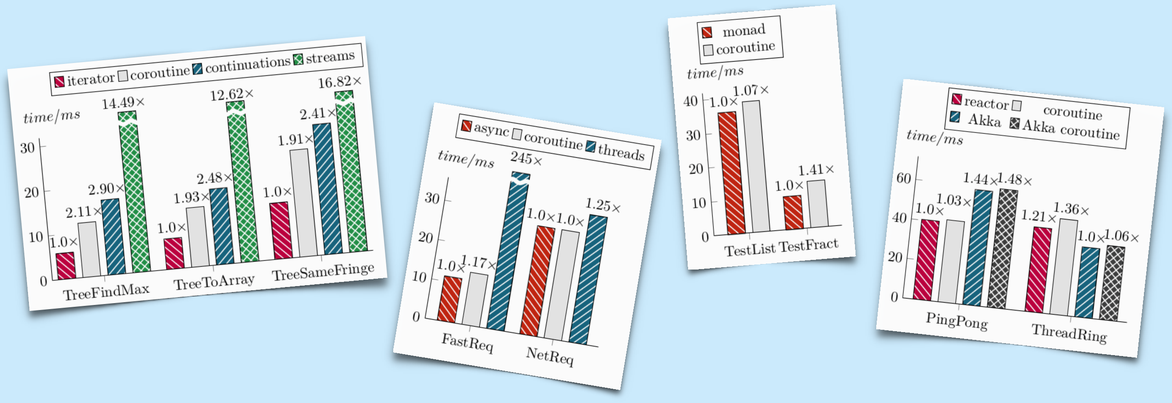
Backtracking

Snapshots duplicate the execution state, and allow backtracking.

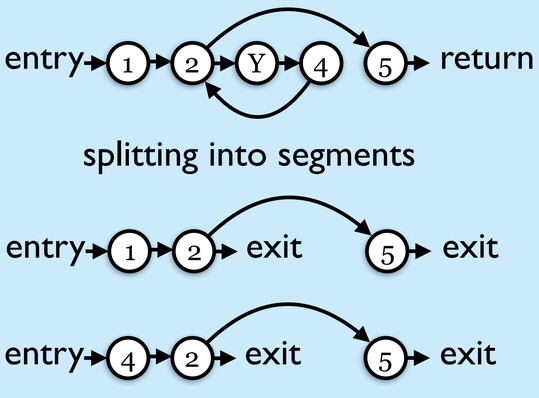
Property-based testing no longer needs monadic generators.

```
test {
  val a = integer(0 until MAX_INT)
  val b = integer(0 until MAX_INT)
  assert(a * b == b * a)
}
```

Performance



Transformation and optimizations



At each yield-point, local state must be saved. The saving overheads can be reduced.

Must-load rule: a variable must be loaded only if the segment uses it.

Was-changed rule: a variable must be saved only if the exit sees a write.

Is-needed rule: a variable must be saved only if there is a reachable segment that consumes it.

On the TreeSameFringe[1] benchmark, these optimizations improve performance by ~2x.

A Tour of Scala Coroutines

At first glance, a coroutine is but a subroutine enclosed in a special **coroutine** block. For example, the coroutine `const` behaves exactly like a subroutine:

```
val const = coroutine { () => return 42 }
```

However, a coroutine can *yield*, which suspends its execution, and gives a value to the point that resumed the coroutine. For example, `twice` yields twice:

```
val twice = coroutine { (x:Int) => yieldval(x); yieldval(x) }
```

Another difference is that a coroutine must be *started* instead of being invoked:

```
val i = twice.start(7)
```

The *coroutine instance* `i` represents the invocation of the coroutine.

Delimited

Consider a cons-list of integers. The code on the right defines a coroutine that yields the integers in the list.

The coroutine definition is *delimited*, meaning that **yieldval** statements can occur only in the lexical scope of the **coroutine** block.

```
val plyList = coroutine {
  (b: List[Int]) =>
  while (b != Nil) {
    yieldval(b.head)
    b = b.tail
  }
}
```

One advantage is that existing libraries do not need to be changed - the transformation is applied only to the **coroutine** blocks.

Stackful

```
val plyArray = coroutine {
  (t: Array[List[Int]]) =>
  var i = 0
  while (i < t.length) {
    plyList(t(i))
    i += 1
  }
}
```

Consider now a hashtable, which is but an array of lists. To yield its elements, a coroutine must traverse the array, and the list in each array slot.

But, wouldn't it be nice if we could reuse the earlier definition of `plyList`?

Within the **coroutine** block, a coroutine definition can be invoked as if it were a normal function - *stackful* coroutines enable reusability.

Type-Safe

Coroutines are type-safe! `val plyList: List[Int]~>(Int,Unit)`

The ⇝ type constructor forms coroutine types. The `plyList`'s type is shown above. Can you guess what the type of `plyArray` is?

The λ⇝ calculus is a theoretical foundation for coroutines with snapshots, and it satisfies the standard safety properties[2].

```
val iterator: Int<~>Unit =
  plyArray.start(hashTable)
```

A coroutine instance's type encodes the *yield type* and the *return type*.

Snapshots

In a way, a coroutine instance is like most iterators - it can only be used once. However, an instance can be cloned, and effectively reused twice.

For example, computing a standard deviation naively requires traversing the list twice. By *snapshotting* the coroutine instance `i` into `j`, we can compute the deviation in the 2nd pass, after computing the average.

```
def stdev(i: Int<~>Unit) = {
  val j = i.snapshot
  var n = 0, sum = 0.0
  while (i.resume) {
    sum += i.value
    n += 1
  }
  var stdev = 0.0
  while (j.resume)
    stdev += sq(j.value-sum/n)
  stdev
}
```

First-Class

```
type Program =
  List[()=>Int]<~>Int

def bt(p: Program) =
  if (p.resume) {
    for (env <- p.value) {
      env(); bt(p.snapshot)
    }
  } else {
    println(p.result)
  }
```

Coroutines are *first-class* entities, so they can be passed around as values. This allows implementing backtracking, for example.

On the left, a `Program` is a coroutine instance that yields a list of environments. The program is resumed recursively for each of the environments. If the program ends, its value is printed.

Comparison with Alternative Constructs

Construct	Type-Safe	First-Class	Stackful	Heap-Free	Scope	Snapshots
C# Enumerators	Yes	No	No	N/A	Delimited	No
Python Generators	No	Yes	Yes	No	Delimited	No
C# Async-Await	Yes	Yes	Yes	No	Delimited	No
Cilk Spawn-Sync	Yes	No	Yes	Yes	Whole program	No
Lua Coroutines	No	Yes	Yes	Yes	Just-In-Time	No
Scala Continuations	Yes	Yes	Yes	No	Delimited	Yes
Scala Coroutines	Yes	Yes	Yes	Yes	Delimited	Yes

[1] A. Prokopec, F. Liu, Theory and Practice of Coroutines with Snapshots, ECOOP 2018
[2] A. Prokopec, F. Liu, On the Soundness of Coroutines with Snapshots, arXiv.org 2018
[3] A. Prokopec, F. Liu, Scala Coroutines Website, https://scala-coroutines.github.io/coroutines

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Aleksandar Prokopec is a principal researcher at Oracle Labs, in the Programming Languages and Runtimes group.

Fengyun Liu is a doctoral assistant at École Polytechnique Fédérale de Lausanne, in the Programming Methods laboratory lead by Prof. Martin Odersky.