Futzing with actors (etc.)

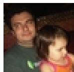
Christopher League

Long Island University

New York Scala Enthusiasts
A Pattern Language of Concurrency

27 June 2011

Analogous advice

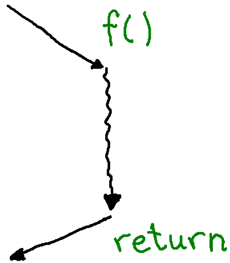


max4f Max Afonov

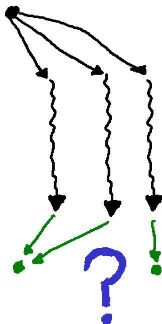
@chrisleague Remember how you forced yourself to be 100% immutable when getting into FP? Force yourself to be 100% non-blocking now.

24 Feb

Call graph



Asynchronicity



Scala actor asynchronicity

```
scala> import scala.actors.Actor._

scala> actor{println("TICK")}; println("TOCK")
TOCK
TICK

scala> actor{println("TICK")}; println("TOCK")
TICK
TOCK
```

Scala actors

- Actors are objects that send/receive messages.
- `a ! m` sends message `m` to actor `a`, and returns immediately (fire and forget).
- System serializes message receives within actor.
- `react` does not block thread, but also does not return.
- Can arrange computations to follow `react` using `loop`, `andThen`.

Scala actor messaging

```
import scala.actors.Actor._
case object Incr
val counter = actor {
  var n = 0
  loop {    // repeatedly wait for a message
    react { // (but don't block thread)
      case Incr => n += 1; println(n)
    }
  }
}

counter ! Incr // fire and forget; eventually
counter ! Incr // prints '1' then '2'
```

Sending replies

```
import scala.actors.Actor._
case object Incr
case object Get
val counter = actor {
  var n = 0
  loop {
    react {
      case Incr => n += 1
      case Get => sender ! n
    }
  }
}
```

Awaiting replies

```
scala> counter.getState
res0: scala.actors.Actor.State.Value = Runnable

scala> counter ! Incr
scala> counter.getState
res2: scala.actors.Actor.State.Value = Suspended

scala> counter ! Incr
scala> val f = counter !! Get
f: counter.Future[Any] = <function0>

scala> f()
res5: Any = 2
```

Return to sender

```
scala> counter ! Incr

scala> val a = actor{
  counter ! Get
  react { case x:Int => println(x) }
}
3
a: scala.actors.Actor = Actor-anon1-@1b17b38

scala> a.getState
res8: scala.actors.Actor.State.Value = Terminated
```

Does sender know best?

- Sometimes awkward for sender to make sense of response.
- Instead, allow reply to another arbitrary actor — we can always specify `self`.

‘Actor-passing style’

```
import scala.actors.Actor
import Actor._
case object Incr
case class Get(k: Actor)
val counter = actor {
  var n = 0
  loop {
    react {
      case Incr => n += 1
      case Get(k) => k ! n
    }
  }
}
```

‘Actor-passing style’

```
scala> counter ! Incr

scala> counter ! Incr

scala> counter ! Get(actor{
  react{
    case x:Int => println(x)
  }
})

scala>
2
```

- Haven’t we seen something like this before?

Continuation-passing style

```scala
def factRecur(n: Int): Int =
  if(n > 0) n * factRecur(n-1)
  else 1

def factCPS[A](n: Int, k: Int => A): A =
  if(n > 0) factCPS(n-1, (x:Int) => k(n*x))
  else k(1)
```

```
scala> factCPS(10, println)
3628800
```

Actor-passing factorial

```
def factAPS(n: Int, k: Actor): Unit =
  if(n > 0) factAPS(n-1, actor{
    react{ case x:Int => k ! (x*n) }
  })
  else k ! 1
```

```
scala> val printer = actor{loop{react{
    case x:Any => println(x)
  }}}
scala> factAPS(7, printer)
5040
scala> factAPS(10, printer)
3628800
```

Tree recursion: Fibonacci

```
def fibRecur(n: Int): Int =
  if(n < 2) 1
  else fibRecur(n-1) + fibRecur(n-2)

def fibCPS[A](n: Int, k: Int => A): A =
  if(n < 2) k(1)
  else fibCPS(n-1, (x:Int) =>
         fibCPS(n-2, (y:Int) =>
           k(x+y)))
```

Actor-passing Fibonacci

```
def fibAPS(n: Int, k: Actor): Unit =
  if(n < 2) k ! 1
  else {
    actor{fibAPS(n-1, ???)}
    fibAPS(n-2, ???)
  }
```

- How to join the results?

Actor-passing Fibonacci

```
def fibAPS(n: Int, k: Actor): Unit =
  if(n < 2) k ! 1
  else {
    val join = actor{
      react{case x:Int =>
        react{ case y:Int => k ! (x+y) }}}

    actor{fibAPS(n-1, join)}
    fibAPS(n-2, join)
  }
```

- Pass the same actor, that receives both results using nested react.

Ordering results with nested react

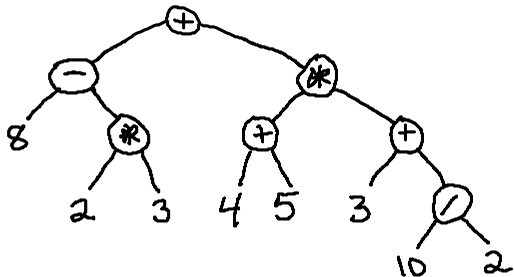
- What if order matters?
- `react` uses a partial function
  - first matching message is used
  - any other messages remain in mailbox

Ordering results with nested react

```
val orderedJoin = actor {
  react{ case (1, x) =>
    react{ case (2, y) => println(x,y) }}}
```

```
scala> orderedJoin ! (1,"Hello")
scala> orderedJoin ! (2,"world")
(Hello,world)

scala> orderedJoin.getState
res3: scala.actors.Actor.State.Value = Terminated
scala> orderedJoin.restart
scala> orderedJoin ! (2,"hacking")
scala> orderedJoin ! (1,"Happy")
(Happy,hacking)
```

An expression tree



Interpreting operators

```
sealed trait Operator
case object Add extends Operator
case object Sub extends Operator
case object Mul extends Operator
case object Div extends Operator

def interpOp(op: Operator, v1: Int, v2: Int): Int =
  op match {
    case Add => v1 + v2
    case Sub => v1 - v2
    case Mul => v1 * v2
    case Div => v1 / v2
  }
```

Building an expression tree

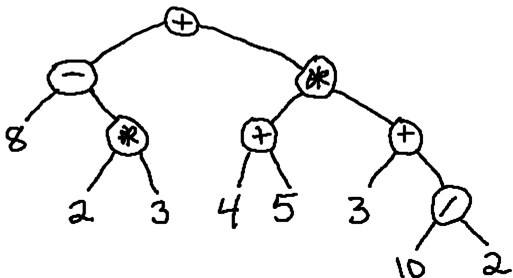
```
sealed trait Expr
case class Const(value: Int) extends Expr
case class BinOp(op: Operator, e1: Expr, e2: Expr)
    extends Expr

val eg1 =
  BinOp(Add,
        BinOp(Sub, Const(8),
              BinOp(Mul, Const(2), Const(3))),
        BinOp(Mul,
              BinOp(Add, Const(4), Const(5)),
              BinOp(Add, Const(3),
                    BinOp(Div, Const(10), Const(2)))))
```

Concurrent tree interpretation

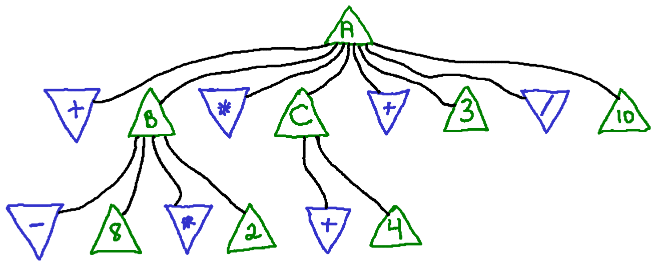
```
def interp(e: Expr, k: Int => Unit): Unit =
  e match {
    case Const(value) => k(value)
    case BinOp(op, e1, e2) => {
      val join = actor{
        react{ case (1, v1:Int) =>
          react{ case (2, v2:Int) =>
            k(interpOp(op,v1,v2)) }}}
      actor{
        interp(e1, (v1:Int) => join ! (1,v1))
      }
      interp(e2, (v2:Int) => join ! (2,v2))
    }
  }
```

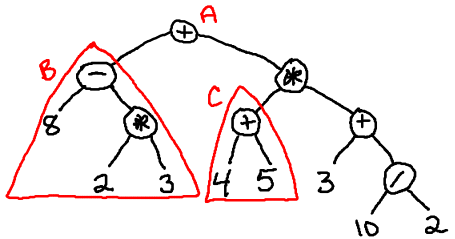
Concurrent tree interpretation

```
scala> interp(eg1, println)
scala>
74
```

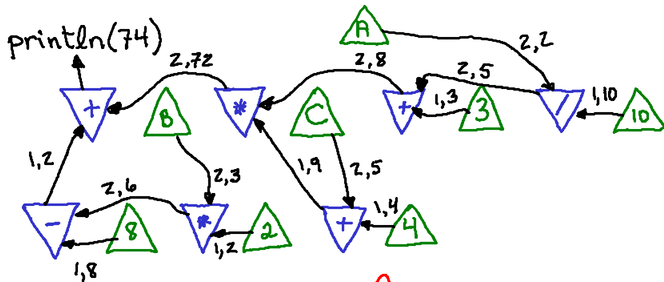


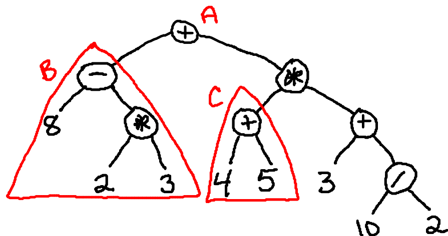
Actors spawned in tree interpreter





Messages sent in tree interpreter





Two actors repeatedly rendezvous

- Next example relies on the flexibility of `react`, `andThen`.
- Can also be solved with lazy streams or coroutines.

Fringe of binary tree

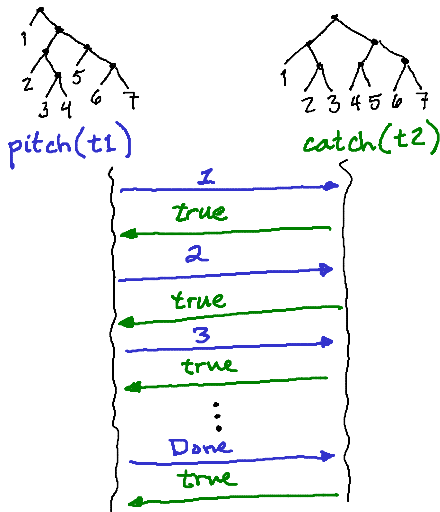
```
sealed trait Tree
case class Leaf(value: Int) extends Tree
case class Branch(left: Tree, right: Tree)
     extends Tree

def fringe(root: Tree): List[Int] = root match {
  case Leaf(value) => List(value)
  case Branch(left, right) =>
    fringe(left) ++ fringe(right)
}
```

Fringe of binary tree

```
val t1 =
  Branch(Leaf(1),
         Branch(Branch(Leaf(2),
                       Branch(Leaf(3),Leaf(4))),
              Branch(Leaf(5),
                     Branch(Leaf(6), Leaf(7)))))
val t2 =
  Branch(Branch(Leaf(1),
              Branch(Leaf(2),Leaf(3))),
         Branch(Branch(Leaf(4),Leaf(5)),
              Branch(Leaf(6),Leaf(7))))
```

```
scala> fringe(t1)
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7)
scala> fringe(t2)
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7)
```

Do two trees have same fringe?



Catcher – traverse and reply true/false

```
def catch_(t: Tree): Unit = t match {
  case Leaf(value) => react {
      case v:Int =>
        if(v == value) sender ! true
        else { sender ! false; exit }
      case Done => sender ! false; exit
  }
  case Branch(left, right) =>
    catch_(left) andThen catch_(right)
}
val catcher = actor {
  catch_(t2) andThen react {
      case Done => sender ! true
      case _ => sender ! false
  }
}
```

Pitcher – traverse, send, await ack

```
def pitch(t: Tree): Unit = t match {
  case Leaf(value) =>
    catcher ! value
    react {
      case true =>
      case false => k(false); exit
    }
  case Branch(left, right) =>
    pitch(left) andThen pitch(right)
}
actor {
  pitch(t1) andThen {
    catcher ! Done
    react {case b: Boolean => k(b)}
  }
}
```

Do two trees have same fringe?

```
def sameFringe(t1: Tree, t2: Tree, k: Boolean => Unit)
{
  def catch_(t: Tree): Unit = ...
  val catcher = actor { ... }
  def pitch(t: Tree): Unit = ...
  actor { ... }
}
```

```
scala> sameFringe(t1, t2, println)
scala>
true

scala> sameFringe(t1, t3, println)
false
scala>
```

Lessons

- Non-blocking actor concurrency
  subverts the call graph, much like CPS
- Actors are stateful, even without using `var`
- State may be represented by nested `react`
- Very cool alternative: `scalaz.concurrent.Promise`
  Ship computations into the **future,** using monads!

Thanks!

`league@contrapunctus.net`
`@chrisleague`

- Code and slides can be made available later; check meetup event page

Bonus: A promising interpreter

```
import scalaz.Scalaz._
import scalaz.concurrent.{Promise, Strategy}
import java.util.concurrent.Executors
implicit val pool = Executors.newFixedThreadPool(5)
implicit val s = Strategy.Executor

def interp(e: Expr): Promise[Int] = e match {
  case Const(value) => promise(value)
  case BinOp(op, e1, e2) =>
    val p1 = promise(interp(e1))
    val p2 = interp(e2)
    for(v1 <- p1.join; v2 <- p2)
    yield interpOp(op, v1, v2)
}
```