Installing Trampolines in Kiama's Pretty-Printer

Anthony M. Sloane

Programming Languages Research Group
Department of Computing
Macquarie University

Anthony.Sloane@mq.edu.au
inkytonik@gmail.com
@inkytonik

June 26, 2013

Kiama pretty-printing

```
import org.kiama.output.PrettyPrinter
import PrettyPrinter._

def show (n : Int) : Doc = hsep ((1 to n) map value)

val n = 3
val data = List.fill (n) (4)

pretty (vsep (data map show))
```

- So far, so good:

```
1 2 3 4
1 2 3 4
1 2 3 4
```

Not so far...

```
val n = 1000

java.lang.StackOverflowError
    at scala.collection.immutable.Queue.isEmpty(Queue.scala
    at org.kiama.output.PrettyPrinter$$anonfun$org$kiama$out
    at org.kiama.output.PrettyPrinter$$anonfun$org$kiama$out
    at org.kiama.output.PrettyPrinter$$anonfun$org$kiama$out
    ...
```

Stack overflow

```
def even[A] (ns : List[A]) : Boolean =
  ns match {
    case Nil     => true
    case x :: xs => odd (xs)
  }

def odd[A] (ns : List[A]) : Boolean =
  ns match {
    case Nil     => false
    case x :: xs => even (xs)
  }
```

- Scala does no indirect tail call optimisation.

Trampolines

- A data representation of a computation that produces a value of type `A`.

```
sealed abstract class Trampoline[+A]
```

- We're done and we produce the value `a`.

```
case class Done[+A] (a : A) extends Trampoline[A]
```

- We're continuing with the computation produced by `k`.

```
case class More[+A] (k : () => Trampoline[A])
    extends Trampoline[A]
```

Avoiding stack overflow

```
def even[A] (ns : List[A]) : Trampoline[Boolean] =
  ns match {
    case Nil     => Done (true)
    case x :: xs => More (() => odd (xs))
  }

def odd[A] (ns : List[A]) : Trampoline[Boolean] =
  ns match {
    case Nil     => Done (false)
    case x :: xs => More (() => even (xs))
  }
```

runT

- Build the computation and run it

```
(even (List (7, 4, 6, 2))).runT
```

- All the recursion is in `runT` and it's a direct tail recursive call:

```
sealed abstract class Trampoline[+A] {

  final def runT : A =
    this match {
      case More (k) => k ().runT
      case Done (v) => v
    }

}
```

It's a loop!

```
public final A runT();
  Code:
     0: aload_0
     1: astore_2
     2: aload_2
     3: instanceof    #10          // Trampolines$More
     6: ifeq          34
        ...
    15: invokevirtual #14          // Trampolines$More.k
    18: astore        4
    20: aload         4
    22: invokeinterface #19,  1    // Function0.apply()
    27: checkcast     #2
    30: astore_0
    31: goto          0
    34: ...
```

Original pretty-printing types

```
def pretty (d : Doc, w : Width) : Layout = {
  val initBuffer = new ListBuffer[String] ()
  val cend = (p, dq) => (r) => initBuffer
  val finalBuffer = d (0, w) (cend) (0, emptyDq) (w)
  finalBuffer.mkString
}
```

```
  type Remaining  = Int
  type Horizontal = Boolean
  type Buffer     = ListBuffer[String]
  type Out        = Remaining => Buffer
  type OutGroup   = Horizontal => Out => Out
  type PPosition  = Int
  type Dq         = Queue[(PPosition,OutGroup)]
  type TreeCont   = (PPosition,Dq) => Out
  type IW         = (Indent,Width)
  type DocCont    = IW => TreeCont => TreeCont
```

New pretty-printer types: three levels of trampoline...

```
type Remaining  = Int
type Horizontal = Boolean
type Buffer     = ListBuffer[String]
type Out        = Remaining => Trampoline[Buffer]
type OutGroup   = Horizontal => Out => Trampoline[Out]
type PPosition  = Int
type Dq         = Queue[(PPosition,OutGroup)]
type TreeCont   = (PPosition,Dq) => Trampoline[Out]
type IW         = (Indent,Width)
type DocCont    = IW => TreeCont => Trampoline[TreeCont]
```

Code transformation: final continuation

```
val cend = (p, dq) => (r) => initBuffer
```

- becomes

```
val cend = (p, dq) => Done (r => Done (initBuffer))
```

Code transformation: computing the final buffer

```
val finalBuffer = d (0, w) (cend) (0, emptyDq) (w)
```

- becomes

```
val finalBufferComputation =
    for {
        c <- d (0, w) (cend)
        o <- c (0, emptyDq)
        buffer <- o (w)
    } yield buffer

val finalBuffer = finalBufferComputation.runT
```

Transformation of library functions (before)

```
def text (t : String) : Doc =
  if (t == "") empty else
    new Doc (
      (iw : IW) => {
        val l = t.length
        val outText =
          (_ : Horizontal) => (o : Out) =>
            (r : Remaining) =>
              t + o (r - l)
        scan (l, outText)
      }
    )
```

Transformation of library functions (after)

```
def text (t : String) : Doc =
  if (t == "") empty else
    new Doc (
      (iw : IW) => {
        val l = t.length
        val outText =
          (_ : Horizontal) => (o : Out) =>
            Done (
              (r : Remaining) =>
                More (() =>
                  for {
                     buffer <- o (r - l)
                  } yield t +=: buffer
                ))
          scan (l, outText)
      })
```

Victory!

```
val n = 100000

1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
...
```

Conclusion

- A relatively easy and very successful application of trampolines to a non-trivial application area.

- More information
  - "Stackless Scala With Free Monads" by Rúnar Óli Bjarnason

`http://apocalisp.wordpress.com/2012/05/15/stackless-scala-with-free-monads-2`