DIY Continuations

Paolo Picci

"Say you're in the kitchen in front of the refrigerator, thinking about a sandwich. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwich, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwich. But fortunately, there's a sandwich on the counter, and all the materials used to make it are gone. So you eat it."

- Palmer, Luke


What is exactly a continuation?

What is a continuation?

A continuation is a portion of the context surrounding an expression.

```
(10 - ((4) * 2)) + 1
```

What is a continuation?

A continuation is a portion of the context surrounding an expression.

```
(10 - ((4) * 2)) + 1
```

```
val a  = 4
val k1 = (x: Int) => x * 2
```

What is a continuation?

A continuation is a portion of the context surrounding an expression.

```
(10 - (k1(a))) + 1
```

```
val a  = 4
val k1 = (x: Int) => x * 2

val k2 = (x: Int) => 10 - x
```

What is a continuation?

A continuation is a portion of the context surrounding an expression.

```
(k2(k1(a))) + 1
```

```
val a  = 4
val k1 = (x: Int) => x * 2

val k2 = (x: Int) => 10 - x

val k3 = (x: Int) => x + 1
```

What is a continuation?

A continuation is a portion of the context surrounding an expression.

$k(a)$

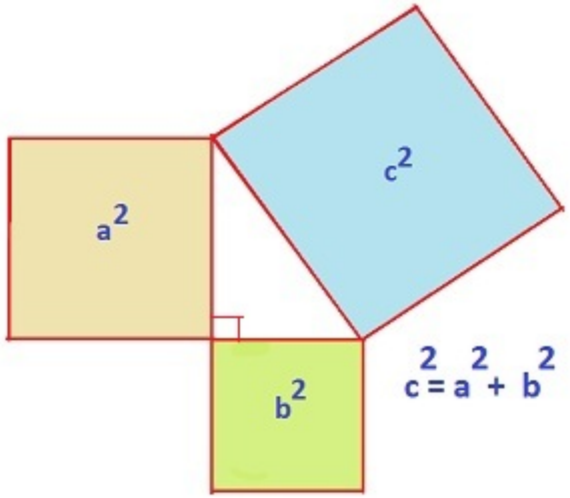
```
val a  = 4
val k1 = (x: Int) => x * 2

val k2 = (x: Int) => 10 - x

val k3 = (x: Int) => x + 1

val k = k3 compose k2 compose k1
```

Direct style

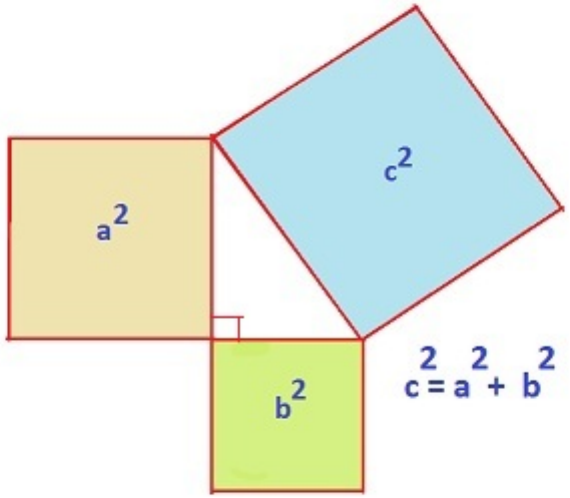



```
def square(x: Int): Int =
  x * x

def sum(x: Int, y: Int): Int =
  x + y

def pitagoras(a: Int, b: Int): Int = {
  val aa = square(a)
  val bb = square(b)
  sum(aa, bb)
}
```

Continuation Passing Style (CPS)




```scala
def square(x: Int, k: Int => Int): Int =
  k(x * x)

def sum(x: Int, y: Int, k: Int => Int): Int =
  k(x + y)

def pitagoras(a: Int, b: Int, k: Int => Int): Int =
  square(a, aa =>
    square(b, bb =>
      sum(aa, bb, k)
    )
  )
```

CPS 101

A real world example

A Recursive Data Type

```scala
sealed trait Tree[T] extends Product with Serializable

final case class Leaf[T](value: T) extends Tree[T]
final case class Node[T](left: Tree[T], right: Tree[T]) extends Tree[T]



val tree1: Tree[Int] = Node(Leaf(1), Leaf(2))
val tree2: Tree[Int] = Node(Leaf(1), Leaf(3))

val sameTrees: Boolean = tree1 == tree2
```

A Recursive Data Type

```scala
sealed trait Tree[T] extends Product with Serializable

final case class Leaf[T](value: T) extends Tree[T]
final case class Node[T](left: Tree[T], right: Tree[T]) extends Tree[T]



val tree3: Tree[Int] = treeFromInput
val tree4: Tree[Int] = treeFromInput

val sameInput: Boolean = tree3 == tree4
```

A Recursive Data Type

```scala
sealed trait Tree[T] extends Product with Serializable

final case class Leaf[T](value: T) extends Tree[T]
final case class Node[T](left: Tree[T], right: Tree[T]) extends Tree[T]



def map[A, B](t: Tree[A])(f: A => B): Tree[B] = t match {
  case Leaf(a)    => Leaf(f(a))
  case Node(l, r) => Node(map(l)(f), map(r)(f))
}
```

Using CPS

```
def map[A, B](t: Tree[A])(f: A => B): Tree[B] = {

  def mapping(tt: Tree[A]                                ): Tree[B] = tt match {
    case Leaf(a)       => ???
    case Node(la, ra) => ???






  }

  mapping(t      )
}
```

Using CPS

```
def map[A, B](t: Tree[A])(f: A => B): Tree[B] = {

  def mapping(tt: Tree[A], k: Tree[B] => Tree[B]): Tree[B] = tt match {
    case Leaf(a)      => ???
    case Node(la, ra) => ???






  }

  mapping(t, ???)
}
```

Using CPS

```scala
def map[A, B](t: Tree[A])(f: A => B): Tree[B] = {

  def mapping(tt: Tree[A], k: Tree[B] => Tree[B]): Tree[B] = tt match {
    case Leaf(a)       => ???
    case Node(la, ra) => ???






  }

  mapping(t, x => x)
}
```

Using CPS

```scala
def map[A, B](t: Tree[A])(f: A => B): Tree[B] = {

  def mapping(tt: Tree[A], k: Tree[B] => Tree[B]): Tree[B] = tt match {
    case Leaf(a)       => k(Leaf(f(a)))
    case Node(la, ra) => ???





  }

  mapping(t, x => x)
}
```

Using CPS

```
def map[A, B](t: Tree[A])(f: A => B): Tree[B] = {

  def mapping(tt: Tree[A], k: Tree[B] => Tree[B]): Tree[B] = tt match {
    case Leaf(a)      => k(Leaf(f(a)))
    case Node(la, ra) =>
      mapping(la, lb =>
        mapping(ra, rb =>
          k(Node(lb, rb))
        )
      )
  }

  mapping(t, x => x)
}
```

Converting CPS in a data type

```
def mapping(tt: Tree[A], k: Tree[B] => Tree[B]): Tree[B]

def mapping(tt: Tree[A]): (Tree[B] => Tree[B]) => Tree[B]

def mapping(tt: Tree[A]): (C => D) => D

def mapping(tt: Tree[A]):         Cont
```

```
Cont[A, R](k: (A => R) => R)
```

TDD: Continuation data type

```
case class Cont[A, R](k: (A => R) => R) {

  def compose[B](f: A => Cont[B, R]): Cont[B, R] =
    Cont[B, R] {
      def type_0:     (B => R) => R          = ???





    }
}
```

TDD: Continuation data type

```
case class Cont[A, R](k: (A => R) => R) {

  def compose[B](f: A => Cont[B, R]): Cont[B, R] =
    Cont[B, R] {
      def type_0:      (B => R) => R          = ???
      val type_1:      (A => R) => R          = k
    }
}
```

TDD: Continuation data type

```
case class Cont[A, R](k: (A => R) => R) {

  def compose[B](f: A => Cont[B, R]): Cont[B, R] =
    Cont[B, R] {
      def type_0:       (B => R) => R         = ???
      val type_1:       (A => R) => R         = k
      val type_2:       A => Cont[B, R]       = f



    }
}
```

TDD: Continuation data type

```
case class Cont[A, R](k: (A => R) => R) {

  def compose[B](f: A => Cont[B, R]): Cont[B, R] =
    Cont[B, R] {
      def type_0:       (B => R) => R           = ???
      val type_1:       (A => R) => R           = k
      val type_2:       A => Cont[B, R]         = f
      val type_3:       A => (B => R) => R      = a => f(a).k




    }
}
```

TDD: Continuation data type

```
case class Cont[A, R](k: (A => R) => R) {

  def compose[B](f: A => Cont[B, R]): Cont[B, R] =
    Cont[B, R] { k1 =>  (???: R)
      def type_0:       (B => R) => R          = ???
      val type_1:       (A => R) => R          = k
      val type_2:       A => Cont[B, R]        = f
      val type_3:       A => (B => R) => R     = a => f(a).k
      val type_4:       B => R                 = k1





    }
}
```

TDD: Continuation data type

```scala
case class Cont[A, R](k: (A => R) => R) {

  def compose[B](f: A => Cont[B, R]): Cont[B, R] =
    Cont[B, R] { k1 =>  (???: R)
      def type_0:       (B => R) => R          = ???
      val type_1:       (A => R) => R          = k
      val type_2:       A => Cont[B, R]        = f
      val type_3:       A => (B => R) => R     = a => f(a).k
      val type_4:       B => R                 = k1
      val type_5:       A => R                 = a => f(a).k(k1)



    }
}
```

TDD: Continuation data type

```
case class Cont[A, R](k: (A => R) => R) {

  def compose[B](f: A => Cont[B, R]): Cont[B, R] =
    Cont[B, R] { k1 =>  (???: R)
      def type_0:       (B => R) => R          = ???
      val type_1:       (A => R) => R          = k
      val type_2:       A => Cont[B, R]        = f
      val type_3:       A => (B => R) => R     = a => f(a).k
      val type_4:       B => R                 = k1
      val type_5:       A => R                 = a => f(a).k(k1)
      val type_6:       R                      = k( a => f(a).k(k1) )


    }
}
```

TDD: Continuation data type

```
case class Cont[A, R](k: (A => R) => R) {

  def compose[B](f: A => Cont[B, R]): Cont[B, R] =
    Cont[B, R] {
      def type_0:     (B => R) => R          = ???
      val type_1:     (A => R) => R          = k
      val type_2:     A => Cont[B, R]        = f
      val type_3:     A => (B => R) => R     = a => f(a).k
      val type_4:     B => R                 = k1
      val type_5:     A => R                 = a => f(a).k(k1)
      val type_6:     R                      = k( a => f(a).k(k1) )

      k1  =>  k( a => f(a).k(k1) )
    }
}
```

TDD: Continuation data type

```scala
case class Cont[A, R](k: (A => R) => R) {

  def compose[B](f: A => Cont[B, R]): Cont[B, R] =
    Cont[B, R] {
      def type_0:       (B => R) => R          = ???
      val type_1:       (A => R) => R          = k
      val type_2:       A => Cont[B, R]        = f
      val type_3:       A => (B => R) => R     = a => f(a).k
      val type_4:       B => R                 = k1
      val type_5:       A => R                 = a => f(a).k(k1)
      val type_6:       R                      = k( a => f(a).k(k1) )

      k1  =>  k( a => f(a).k(k1) )
    }
}

def pure[A, R](a: A) = Cont[A,R]( k => k(a) )
```

Continuation data type

```scala
final case class Continuation[A, R](k: (A => R) => R) { self =>
  def flatMap[B](f: A => Continuation[B, R]): Continuation[B, R] =
    Continuation(   br  =>  k( a => f(a).k(br) )    )

  def map[B](f: A => B): Continuation[B, R] =
    Continuation(   br  =>  k( a => br(f(a)) )    )

  def run(f: A => R): R = k( a => f(a) )
}

object Continuation {
  def pure[A, R](value: A): Continuation[A, R] =
    Continuation( k => k(value) )
}
```

Using CPS DataType

```scala
def map[A, B](t: Tree[A])(f: A => B): Tree[B] = {

  def mapping(tt: Tree[A], k: Tree[B] => Tree[B]): Tree[B] = tt match {
    case Leaf(a)      => k(Leaf(f(a)))
    case Node(la, ra) =>
      mapping(la, lb =>
        mapping(ra, rb =>
          k(Node(lb, rb))
        )
      )
  }

  mapping(t, x => x)
}
```

Using CPS DataType

```scala
def map[A, B](t: Tree[A])(f: A => B): Tree[B] = {

  def mapping(tt: Tree[A]): Continuation[Tree[B], Tree[B]] = tt match {
    case Leaf(a)       => pure(Leaf(f(a)))
    case Node(la, ra) =>
      for {
        lb <- mapping(la)
        rb <- mapping(ra)
      } yield Node(lb, rb)

  }

  mapping(t).run(identity)
}
```

Continuation what are they good for?

- Write tail recursive functions.
- Abstract over the flow of your program. Every function is in control of what comes next.
- Can be used to implement many useful control structures
  eg: coroutines, async calls with call backs, backtracking, …
- Representing Monads

```
counter = 0
game = 0


LABEL("START")
counter = 1
game += 1
PRINT(s"-- Round $game --")
LABEL("INCREMENT")
PRINT(s" counter = $counter")
counter += 1
if (game > 2 && counter > 3)
  CONTINUE
else if (counter <= 3)
  GOTO("INCREMENT")
else
  GOTO("START")
PRINT("-- Bye Bye --")
```

OUTPUT

-- Round 1 --
counter = 1
counter = 2
counter = 3
-- Round 2 --
counter = 1
counter = 2
counter = 3
-- Round 3 --
counter = 1
counter = 2
counter = 3
-- Bye Bye --

```scala
var counter = 0
var game = 0

val program = for {
  _ <- LABEL("START")
  _ =  counter = 1
  _ =  game += 1
  _ =  println(s"-- Round $game --")
  _ <- LABEL("INCREMENT")
  _ =  println(s" counter = $counter")
  _ =  counter += 1
  _ <- if (game > 2 && counter > 3)
         CONTINUE
       else if (counter <= 3)
         GOTO("INCREMENT")
       else
         GOTO("START")
  _ =  println("-- Bye Bye --")
} yield ()

program.run(x => x)
```

```scala
def LABEL(value: String): Continuation[Unit, Unit]
def GOTO(value: String): Continuation[Unit, Unit]
def CONTINUE: Continuation[Unit, Unit]
```

```scala
var counter = 0
var game = 0

val program = for {
  _ <- LABEL("START")
  _ =  counter = 1
  _ =  game += 1
  _ =  println(s"-- Round $game --")
  _ <- LABEL("INCREMENT")
  _ =  println(s" counter = $counter")
  _ =  counter += 1
  _ <- if (game > 2 && counter > 3)
         CONTINUE
       else if (counter <= 3)
         GOTO("INCREMENT")
       else
         GOTO("START")
  _ =  println("-- Bye Bye --")
} yield ()

program.run(x => x)
```

Performances and Problems.

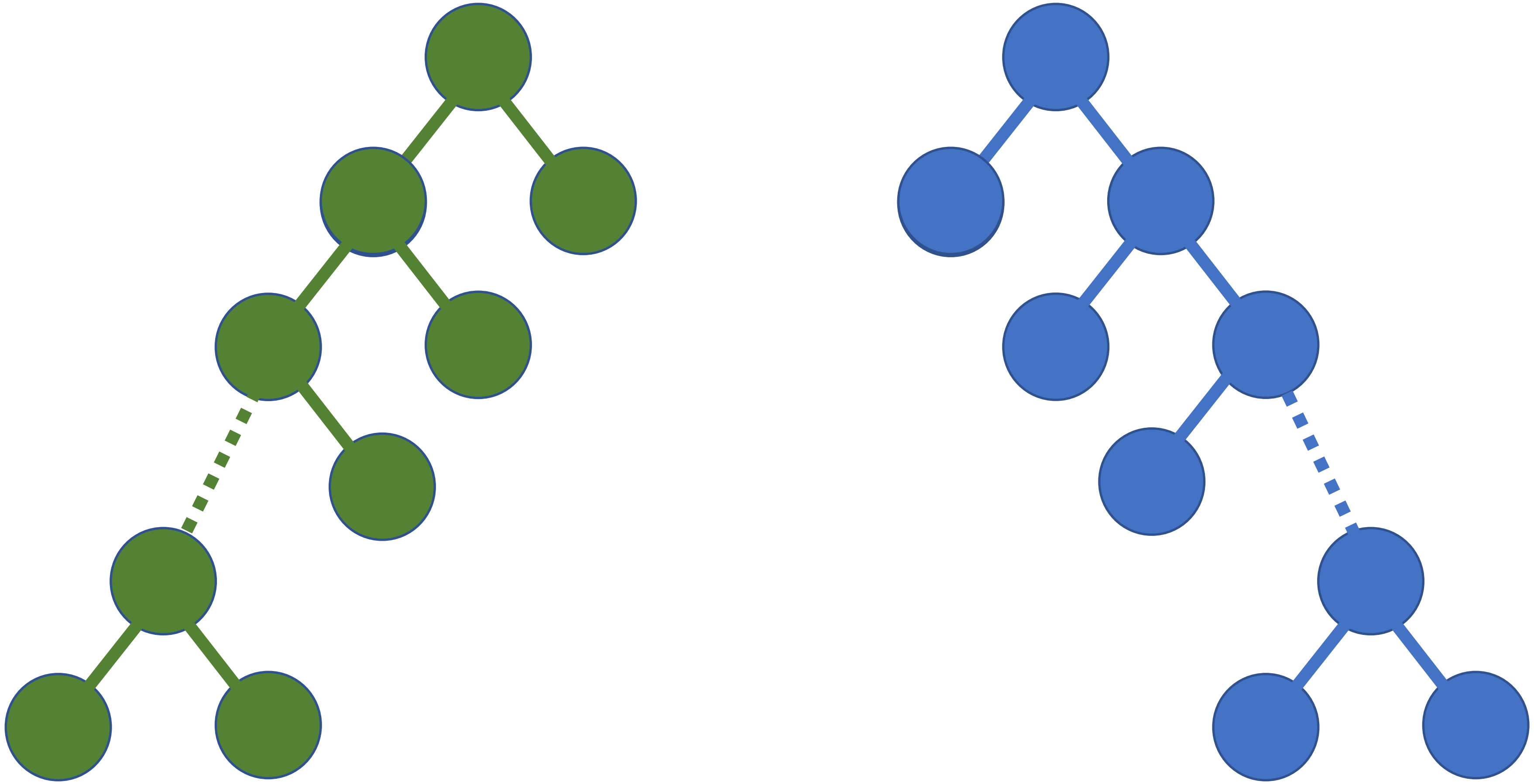
Is my CPS function not tailrec?

```scala
@tailrec
def g(k: Unit => Unit): Unit = g(x => g(k))
```

```
[info] Compiling 1 Scala source …
[error] /…/src/main/scala/example/Var.scala:12:42:
could not optimize @tailrec annotated method g: it contains a
recursive call not in tail position
[error]   def g(k: Unit => Unit): Unit = g(x => g(k))
[error]                                         ^
[error] one error found
```

Big Trees

```
def mapping(t: Tree[A]): Continuation[Tree[B], Tree[B]]
```



Big Trees

```
def mapping(tt: Tree[A], k: Tree[B] => Tree[B]): Tree[B] = tt match {
  case Leaf(a)       => k(Leaf(f(a)))
  case Node(la, ra) =>
    mapping(la, lb =>
      mapping(ra, rb =>
        k(Node(lb, rb))
      )
    )
}

mapping(t, x => x)
```

```
k1 compose k2 compose k3 compose ... compose kn
```

Big Trees

```scala
import scala.util.control.TailCalls._

def map[A, B](t: Tree[A])(f: A => B): Tree[B] = {

  def mapping(tt: Tree[A], k: Tree[B] => TailRec[Tree[B]]): TailRec[Tree[B]] = tt match {
    case Leaf(a)      => k(Leaf(f(a)))
    case Node(la, ra) =>
      mapping(la, lb => tailcall(
        mapping(ra, rb => tailcall(
          k(Node(lb, rb))
        ))
      ))
  }

  mapping(t, x => done(x)).result
}
```

Other resources

Scala compiler plugin

https://github.com/scala/scala-continuations

Cats

https://github.com/typelevel/cats/blob/master/core/src/main/scala/cats/data/ContT.scala

Questions?

Slides and code at

https://github.com/qqupp/continuations-playground/