# Continuations

Christophe Scholliers

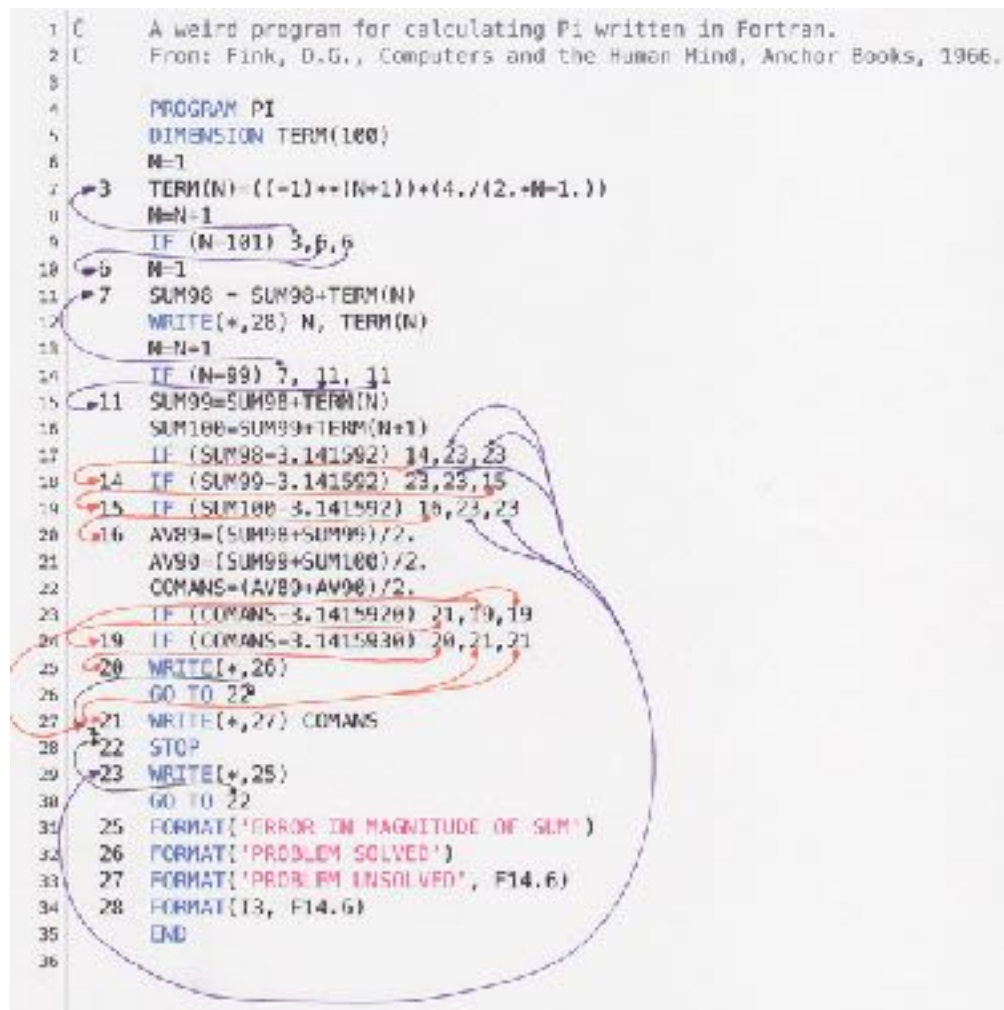
https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

# (Un)Structured Programming

Knuth

Go-to is (usually) bad !

1974

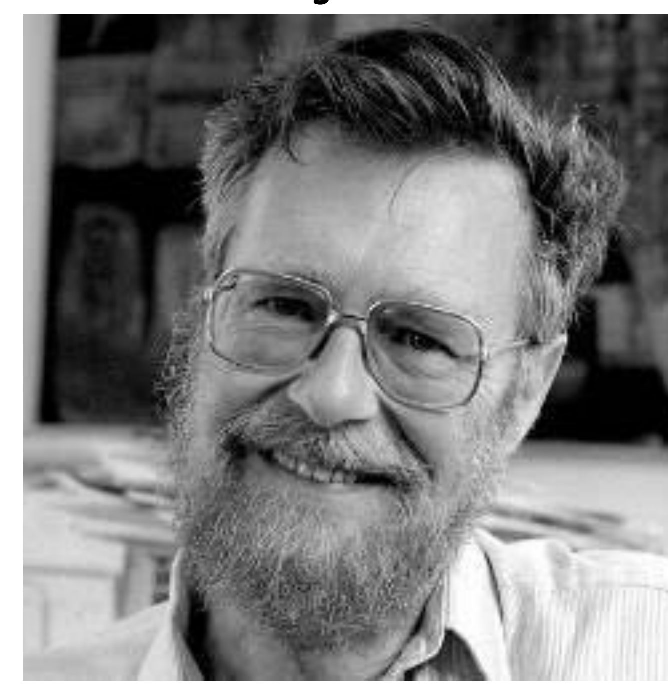


```
C     A weird program for calculating Pi written in Fortran.
C     From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.

      PROGRAM PI
      DIMENSION TERM(100)
      N=1
    3 TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
      N=N+1
      IF (N-101) 3,6,6
    6 N=1
    7 SUM98 = SUM98+TERM(N)
      WRITE(*,28) N, TERM(N)
      N=N+1
      IF (N-99) 7, 11, 11
   11 SUM99=SUM98+TERM(N)
      SUM100=SUM99+TERM(N+1)
      IF (SUM98-3.141592) 14,23,23
   14 IF (SUM99-3.141592) 23,23,15
   15 IF (SUM100-3.141592) 16,23,23
   16 AV89=(SUM98+SUM99)/2.
      AV90=(SUM99+SUM100)/2.
      COMANS=(AV89+AV90)/2.
      IF (COMANS-3.1415920) 21,19,19
   19 IF (COMANS-3.1415930) 20,21,21
   20 WRITE(*,26)
      GO TO 22
   21 WRITE(*,27) COMANS
   22 STOP
   23 WRITE(*,25)
      GO TO 22
   25 FORMAT('ERROR IN MAGNITUDE OF SUM')
   26 FORMAT('PROBLEM SOLVED')
   27 FORMAT('PROBLEM UNSOLVED', F14.6)
   28 FORMAT(I3, F14.6)
      END
```

GOTO

Dijkstra



Go-to considered harmful!

1968

# Structured Control Flow

```
if (x>10) {
    ...
} else {
    ...
}
```

```
while(x>0) {
    ...
}
```

```
for(int i=0;i<10;i++) {
    ...
}
```

```
try {
    int result = divide(2,1);
    System.out.println(result);
} catch (BadNumberException e) {
   //do something clever with the exception
   System.out.println(e.getMessage());
}
```

*Continuations*

# Continuations

*“A continuation represents the computational **process at a given point** in the process execution”*

*“Calling a continuation **aborts** the current control flow and **restores** the computational process it represents”*

# Continuations $\lambda_{Num}$


Continuation

`(1 + (1 + (1 + 0)))`  `(1 + (` ● `))`  `(1 + 2)`  `3`

Continuation

`(+ 1 (+ 1 (+ 1 0)))` $\xrightarrow{+}$ `(+ 1 (+ 1 1))` $\xrightarrow{+}$ `(+ 1 ●)`

*"A continuation represents the computational **process at a given point** in the process execution"*

# How can we represent continuations ?

```
(+ 1 (+ 1 ● ))
```

# Suspended computations

"suspended"

```
map ($ 2) [(2*), (4*), (8*)]
```

"continuations"

# Suspended computations

“suspended”

```
map ($ 2) [(2*), (4*), (8*)]
>[4,8,16]
```

“continuations”

```
Prelude> :t ($)
($) :: (a -> b) -> a -> b
```

# Continuation Passing Style

**In CPS, each procedure takes an extra argument representing what should be done with the result the function is calculating.**

https://en.wikipedia.org/wiki/Continuation-passing_style

# Continuation Passing Style

```
add      :: Int -> Int -> Int
add x y = x + y

square :: Int -> Int
square x = x*x

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

# Continuation Passing Style

```
add_cps :: Int->Int->(Int->b)->b
add_cps    x y k =  k (add x y)

square_cps :: Int->(Int->b)->b
square_cps x k =  k (square x)
```

# Continuation Passing Style

```
add_cps :: Int->Int->(Int->b)->b
add_cps    x y k =  k (add x y)

square_cps :: Int->(Int->b)->b
square_cps x k =  k (square x)


pythagoras_cps :: Int->Int->(Int->b)->b
pythagoras_cps x y k = …
```

# Continuation Passing Style

```
add_cps :: Int->Int->(Int->b)->b
add_cps    x y k =  k (add x y)

square_cps :: Int->(Int->b)->b
square_cps x k =  k (square x)


pythagoras_cps :: Int->Int->((Int->r)->r)
pythagoras_cps x y = \k ->
                   square_cps x $ \xr ->
                   square_cps y $ \yr ->
                   add_cps xr yr k
```

# Factorial

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

# Factorial

```
factorial_cps :: Int -> ((Int -> r) -> r)
factorial_cps 0 = \k -> k 1
factorial_cps n =  …
```

# Factorial

```
factorial_cps :: Int -> ((Int -> r) -> r)
factorial_cps 0 = \k -> k 1
factorial_cps n = \k ->
                   minus_cps n 1 $ \n' ->
                   factorial_cps n' $ \res ->
                   k (res * n)
```

# CPS arguments

```
thrice :: (a->a) -> a -> a
thrice f v = f (f (f v))

thrice_cps :: (a->(a->r)->r)->a-> (a->r) ->r
thrice_cps f_cps v = …
```

# CPS arguments

```
thrice :: (a->a) -> a -> a
thrice f v = f (f (f v))

thrice_cps :: (a->(a->b)->b)->a-> (a->b) ->b
thrice_cps f_cps v = \k ->
                     f_cps v   $ \fv ->
                     f_cps fv  $ \ffv ->
                     f_cps ffv $ k
```

# Pattern Matching

# Pattern matching on Bool

```
check :: Bool -> String
check b = case b of
    True  -> "It's True"
    False -> "It's False"
```

# PM BoolCPS

```
type BoolCPS r = r -> r -> r

true :: BoolCPS r
true x _ = x

false :: BoolCPS r
false _ x = x

check :: BoolCPS String -> String
check b = b "It's True" "It's False"
```

# Elaborate example

```
data Expr = Zero | Inc Int | Add Int Int

compute :: Expr -> Int
compute x = case x of
              Zero -> 0
              Inc a -> a +1
              Add a b -> a + b
```

# Elaborate example

```
type ExprCPS r = (r) -> (Int -> r) -> (Int -> Int -> r) -> r

zero :: ExprCPS r
zero x _ _ = x
inc :: Int -> ExprCPS r
inc a _ f _ =  f a
add :: Int -> Int -> ExprCPS r
add a b _ _ f = f a b

computeCPS e = e 0 (+1) (+)
```

# CPS combinators

# CPS Combinator

```
--                suspended a        a-> suspended b               suspended b
chainCPS :: ((a -> r) -> r) -> (a -> ((b -> r) -> r)) -> ((b -> r) -> r)
chainCPS sa fcps = \k -> …
```

# CPS Combinator

```
--                    suspended a
chainCPS :: ((a -> r) -> r) ->
--                    a-> suspended b
                 (a -> ((b -> r) -> r)) ->
--                     suspended b
                 ((b -> r) -> r)
chainCPS sa f = \k -> sa $ \x-> f x k
```

# Monad

```
--              suspended a       a-> suspended b            suspended b
--              M a               a-> M b                    M b
chainCPS :: ((a -> r) -> r) -> (a -> ((b -> r) -> r)) -> ((b -> r) -> r)
chainCPS sa fcps = \k -> sa $ \x-> fcps x k



data Suspended r a = Suspended { run :: (a->r)->r }


instance Monad (Suspended r) where
 sa >>= f = Suspended $ \k -> (run sa) $ \a -> run (f a) k
 return a = Suspended $ \k -> k a
```

# Example

```haskell
add      :: Int -> Int -> Int
add x y = x + y

add_cont :: Int -> Int -> Suspended r Int
add_cont a b = return $ a + b
```

# Example

```
add_cont :: Int -> Int -> Suspended r Int
add_cont a b = return $ a + b

square_cont :: Int -> Suspended r Int
square_cont a = return $ a ^ 2

pythagoras_cont :: Int -> Int -> Suspended r Int
pythagoras_cont a b = …
```

# Example

```
add_cont :: Int -> Int -> Suspended r Int
add_cont a b = return $ a + b

square_cont :: Int -> Suspended r Int
square_cont a = return $ a ^ 2

pythagoras_cont :: Int -> Int -> Suspended r Int
pythagoras_cont a b = do a1 <- square_cont a
                         b1 <- square_cont b
                         add_cont a1 b1
```

# CallCC

```haskell
import Control.Monad.Cont

-- Without callCC
square :: Int -> Cont r Int
square n = return (n ^ 2)

-- With callCC
squareCCC :: Int -> Cont r Int
squareCCC n = callCC $ \k -> k (n ^ 2)
```

# CallCC

```
quux :: Cont r Int
quux = callCC $ \k -> do
    let n = 5
    k n
    return 25
```

*“Calling a continuation **aborts** the current control flow and **restores** the computational process it represents”*

# CallCC

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = …
```

# CallCC

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = cont $
           \h ->
           runCont
           (f (\a -> cont $ \_ -> h a))
           h
```

# Exceptions

# Exceptions

```
divExcp :: Int -> Int -> (String -> Cont r Int) -> Cont r Int
divExcp x y handler = callCC $ \ok -> do
        err <- callCC $ \notOk -> do
            when (y==0) $ notOk "Denominator 0"
            ok $ x `div` y
        handler err
```

# Exceptions

```
tryCont :: MonadCont m => ((err->m a)-> m a) ->
                          (err -> m a) ->
                          m a
tryCont c h = callCC $ \ok -> do
     err <- callCC $ \notOk -> do
            x <- c notOk
            ok x
     h err
```

# Exceptions

```
data SqrtException = LessThanZero deriving (Show,Eq)

sqrtIO :: (SqrtException -> ContT r IO()) -> ContT r IO ()
sqrtIO throw = do
   ln <- lift (putStr "Enter a numbert to sqrt:" >> readLn)
   when (ln < 0) (throw LessThanZero)
   lift $ print (sqrt ln)

main = runContT (tryCont sqrtIO (lift . print)) return
```

# Co-routines

# CoroutineT

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype CoroutineT r m a = CoroutineT
    {runCoroutineT' :: ContT r (StateT [CoroutineT r m ()] m) a}
    deriving (Functor,Applicative,Monad,MonadCont,MonadIO)
```

# Put and get the coroutine queue

```
getCCs :: Monad m => CoroutineT r m [CoroutineT r m ()]
getCCs = CoroutineT $ lift get

putCCs :: Monad m => [CoroutineT r m ()] -> CoroutineT r m ()
putCCs = CoroutineT . lift . put
```

# Manipulating the queue

```
dequeue :: Monad m => CoroutineT r m ()
dequeue = do
    current_ccs <- getCCs
    case current_ccs of
        [] -> return ()
        (p:ps) -> do
            putCCs ps
            p
```

# Manipulating the queue

```
queue :: Monad m => CoroutineT r m () -> CoroutineT r m ()
queue p = do
    ccs <- getCCs
    putCCs (ccs++[p])
```

# Yield

```
yield :: Monad m => CoroutineT r m ()
yield = callCC $ \k -> do
    queue (k ())
    dequeue
```

# Fork

```haskell
fork :: Monad m => CoroutineT r m () -> CoroutineT r m ()
fork p = callCC $ \k -> do
    queue (k ())
    p
    dequeue
```

# Exhaust

```
exhaust :: Monad m => CoroutineT r m ()
exhaust = do
    exhausted <- null <$> getCCs
    if not exhausted
        then yield >> exhaust
        else return ()
```

# runCoroutineT

```haskell
runCoroutineT :: Monad m => CoroutineT r m r -> m r
runCoroutineT = flip evalStateT [] .
                flip runContT return .
                runCoroutineT' . (<* exhaust)
```

```haskell
(<*) :: Applicative f => f a -> f b -> f a
```

```
*Main> (<* putStrLn "Hello") $ putStrLn "Foo"
Foo
Hello
```

# runCoroutineT

“simplified”

```
runCoroutineT :: Monad m => CoroutineT r m r -> m r
runCoroutineT =  flip evalStateT [] .
                 flip runContT return .
                 runCoroutineT' . run
                 where run x  =  do res <- x
                                    exhaust
                                    return res
```

# Example

```
printOne :: (Show a) => a -> CoroutineT r IO ()
printOne n = do
    liftIO (print n)
    yield

example =  do fork $ replicateM_ 3 (printOne 3)
              fork $ replicateM_ 4 (printOne 4)
              replicateM_ 2 (printOne 2)
```

```
*Main> runCoroutineT example
3
4
3
2
4
```