INTRODUCTION TO

FUNCTIONAL PROGRAMMING

OUTLINE

- Tail Recursion
- Continuation-Passing Style
- CPS Monad

THE PROBLEM WITH RECURSION

- Each time we call a function, some information about it is stored on the stack
- A badly written recursive function stores stuff on the stack until it reaches the base case
- We can run out of memory before we reach the base case: stack overflow.
- Solution: tail recursion, meaning that the recursive call is the last thing the function does

```
fact 1 = 1
fact n = n * fact (n - 1)

fact 4 = 4 * (fact 3) =
       = 4 * (3 * (fact 2)) =
       = 4 * (3 * (2 * (fact 1))) =
       = 4 * (3 * (2 * 1)) =
       = 4 * (3 * 2) =
       = 4 * 6 =
       = 24
```

ACCUMULATOR PATTERN

- Rewrite the function with an accumulator
- Make sure to call the function as the last step
- Nothing is needed to be stored on stack, call stack frame can be reused

```
fact n =
    go 1 n
  where
    go acc 1 = acc
    go acc n = go (n * acc) (n - 1)



fact 4 = go 1 4 =
       = go 4 3 =
       = go 12 2 =
       = go 24 1 =
       = 24
```

ACCUMULATOR PATTERN: LIST REVERSAL

- What's the complexity of this reverse function?

```
reverse [] = []
reverse (h:t) = reverse t ++ [h]
```

ACCUMULATOR PATTERN: LIST REVERSAL

- What's the complexity of this reverse function?
    - It's quadratic
- Let's use accumulator pattern
    - Note that the function is now tail-recursive

```
reverse [] = []
reverse (h:t) = reverse t ++ [h]

reverse =
    go []
  where
    go acc [] = acc
    go acc (h:t) = go (h:acc) t
```

EXERCISE

- Write a tail recursive implementation of:
  - Sum of list of numbers
  - GCD

OUTLINE

- Tail Recursion
- Continuation-Passing Style
- CPS Monad

WAIT, I HAVE TWO CALLS

- Can the problem be solved with two accumulators?

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

WAIT, I HAVE TWO CALLS

- Can the problem be solved with two accumulators?

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fib n =
    if n == 0 then 0 else go n 1 0
  where
    go 0 res _ = res
    go n res prev = go (n-1) (res+prev) res
```

CONTINUATION PASSING STYLE

- Continuation – a function that is going to be called once the current evaluation is finished
- When we rewrite a function in CPS, we add a new argument
- This argument is a continuation, its type – function
- Instead of returning a result, we call continuation on it
- Make sure that functions pass their results to the appropriate continuation

```haskell
add :: Int → Int → Int
add x y = x + y

addCont :: Int → Int → (Int → r) → r
addCont x y k = k (x + y)

main :: IO ()
main = do
  print (add 13 42)
  addCont 13 42 print
  addCont 13 42 (\x → addCont x 777 print)
```

CPS FOR TAIL RECURSION

- Using CPS allows for easy transformation into a tail recursion
- Exercise:
  - Rewrite fibonacci function into CPS

```haskell
factorialCPS :: Int -> (Int -> r) -> r
factorialCPS 0 k = k 1
factorialCPS n k =
  factorialCPS (n - 1) (\r -> k (n * r))

main :: IO ()
main =
  factorial 6 print
```

CPS FOR CONTROL FLOW

- Explicit control flow
- Non-local returns

```
compute :: Int
        → (Int → r)
        → (String → r)
        → r
compute x success failure =
  if x > 0
  then success (x * 2)
  else failure ("Input must be positive")

main = print $
  compute 42 (\x → "Success: " ++ show x)
             (\err → "Failure: " ++ err)
```

- Tail Recursion
- Continuation-Passing Style
- CPS Monad

FIBONACCI IN CPS

```haskell
fibCPS :: Int → (Int → r) → r
fibCPS 0 k = k 0
fibCPS 1 k = k 1
fibCPS n k =
  fibCPS (n-1) $ \a →
  fibCPS (n-2) $ \b →
  k (a + b)

fib :: Int → Int
fib n = fibCPS n id
```

FIBONACCI IN CPS

```haskell
fibCPS :: Int → (Int → r) → r
fibCPS 0 k = k 0
fibCPS 1 k = k 1
fibCPS n k =
  fibCPS (n-1) $ \a →
  fibCPS (n-2) $ \b →
  k (a + b)

fib :: Int → Int
fib n = fibCPS n id
```

```haskell
fibMonad :: Int → Cont r Int
fibMonad 0 = return 0
fibMonad 1 = return 1
fibMonad n = do
  a ← fibMonad (n-1)
  b ← fibMonad (n-2)
  return (a + b)

runFib :: Int → Int
runFib = evalCont . fibMonad
```

CPS MONAD

- Encapsulation of continuations
- Composable control flow
- Automation of the CPS transformation
- Abstraction over control flow

```haskell
newtype MyCont r a
  = MyCont { runMyCont :: (a -> r) -> r }

evalMyCont :: MyCont r r -> r
evalMyCont m = runMyCont m id
```

MONAD TYPE CLASS

- >>= is a way to compose two arrows
- return is a way to create an arrow
- + Applicative

```haskell
class Applicative m => Monad m where
  return :: a -> m a
  return = pure

  (>>=) :: m a -> (a -> m b) -> m b -- bind
  -- ...

{- Laws
Left identity
  return a >>= k == k a

Right identity
  m >>= return == m

Associativity
  m >>= (\x -> k x >>= h) == (m >>= k) >>= h
-}
```