INF-3910 Lecture 1

Continuation passing style (CPS)

- Continuation-passing is a technique where functions don't return a value, but instead call a given function (callback) to produce the next result.

process : a → b

vs. process' : a → (cb : (a → b)) → b

or $\boxed{\text{type Cont<r,a> = (a → r) → r}}$

process' : a → Cont<b,a>

- we can think of continuations as suspended computations: they need a further function to complete.

- Another way of looking at the situation is that a value is meaningless until we apply (use) it:

let five f = f 5 (let five = (|>) 5)

do five (printfn "%d")

- If we think of CPS in terms of callbacks:

```
let process (x:a) (onError: Cont<b>) (onSuccess: Cont<b>)
```

- We can even do things like:

```
List.map ((|>) 2) [(*)2; (*)3; (+)42]
```

$\underbrace{((|>)\ 2)}_{\uparrow\ \text{CPS}}$

- Continuations allow us to significantly alter program flow: coroutines, exceptions, parallelism, async, early return...

- Example: pythagoras CPS

```
let add x y = x + y
let square x = x * x
let pythagoras x y = add (square x) (square y)
```

```
let add' x y = fun f -> f (add x y)
let square' x = fun f -> f (square x)
let pythagoras' x y = fun f ->
    square' x (fun x' ->
    square' y (fun y' ->
    add' x' y' f))
```

- Example: thrice

let thrice $(f: a \to a) \to (x : a) \to a$ $= f\ (f\ (f\ x))$

$= x \triangleright f \triangleright f \triangleright f$

$= f \gg f \gg f$

let thrice' $(f' : a \to (a \to r) \to r) \to (x : a) \to ((a \to r) \to r) =$

or let thrice' $(f' : a \to \text{Cont}\langle a \rangle) \to (x : a) \to \text{Cont}\langle a \rangle$ $=$

fun k → f' x
  (fun fx → f' fx
  (fun ffx → f' ffx k))

- We see a pattern! Let's abstract!

let chain $(s : (a \to r) \to r) \to$
$(f : (a \to (b \to r) \to r) \to$
$(b \to r) \to r$ $=$

or let chain $(s : \text{Cont}\langle a \rangle) \to (f : a \to \text{Cont}\langle b \rangle) \to \text{Cont}\langle b \rangle$

= fun k → s (fun x → f x k)

= Cont.bind !!

let thrice' f' x = (|>) x >>= f' >>= f' >>= f'

- Cont<r, a> is also a functor in <u>a</u>:

```
Cont.map : (f: a -> b) -> (x: Cont<r, a>) -> Cont<r, b>
Cont.map f x = fun g -> x (fun a -> g (f a))
               (b->r)
```

```
- Cont.return x = fun f -> f x
                 (a-r) -> r  = Cont<r, a>
```

Now we can create a computation expression:

```
type ContBuilder () =
    member Bind (a, b)     = a >>= b
    member Return x        = Cont.return x
    member ReturnFrom x    = fun k -> x k

let cont = ContBuilder ()

let thrice f x =
  cont {
    let! x' = f x
    let! x'' = f x'
    return! f x''
  }
```

- Continuations are of great practical and theoretical importance
- The continuation monad is the mother of all monads: every monad can be generated from it! (see Dan Piponi (sigfpe))
- There is one type of continuation which is superbly more common than others:

type Async<'a> = (a → ()) → ()

- Since Cont<(),'a> ≡ Async<'a> doesn't return anything useful (it performs IO or mutation) it can be scheduled to run on a separate thread, and return immediately
- Async is fundamental for multi-threading, concurrency, parallelism, and reactive programming.
- The async { } workflow makes it easy and natural to work with asynchronous computations

Async examples:

```
let sleeper x =
    async {
        do! Async.Sleep x
        return x
    }

let serial () =
    async {
        let! a = sleeper 1000
        let! b = sleeper 2000
    } |> Async.RunSyncronously

let prallel () =
    let s1 = sleeper 1000
    let s2 = Sleeper 2000
    [s1; s2] |> Async.Parallel |> Async.Start
```

F# PowerPack has a ParallelSeq library which contains parallel map, fold, sort...

```
data |> PSeq.map exp
data |> PSeq.reduce (+)
```

This works super for pure functions

Managing shared state using MailboxProcessor

- Pure functions go a long way, but sometimes we need communication between threads. If we are careless, this can become a disaster

- Message passing using agents solve this elegantly (using a serial region!)

```
let private mutable state = 0
let agent = MailboxProcessor.Start (fun inbox ->
    let rec loop () = async {
        let! msg = inbox.Receive()
        state <- msg
        return! loop()
    }

let updateState x = agent.Post x
```