**3 – A prototype library to provide seamless concurrency, data management, and stateless functions**

Links to stuff that inspired my desire to write this:

<http://harmful.cat-v.org/software/OO_programming/why_oo_sucks>

<https://www.yegor256.com/2016/08/15/what-is-wrong-object-oriented-programming.html>

<https://content.pivotal.io/blog/all-evidence-points-to-oop-being-bullshit>

The 3 project is my attempt to attempt to provide a solution to 3 systemic problems in software design by breaking them into their separate pieces:

1. Use of mutable/stateful data is core to event driven programs because it’s the easiest/simplest way to communicate change to many moving parts. As software becomes more advanced one of the greatest struggles to overcome is to limit the effects of exponential complexity growth due to ever increasing numbers of variables. The primary way this has been dealt with is by carefully structuring and modeling data to improve readability, consistency, and correctness.
2. Functions and software design become harder to debug when we have to account for and simulate stateful data modification. IE, functions with mutable internals (pointers to values outside the function) are harder to write and validate than functions that only use immutable input arguments with no other access to exterior state. This is because it breaks basic principles of function abstraction: if you know the function’s API you should be able to reliably count on the function’s output to be as you expect. It also makes the function implementation more brittle, because changing something elsewhere can have a cascading effect throughout the greater program.

Therefore, the easiest functions to validate are those that do not internally modify or internally rely on program state, because there’s only one location to work with to generate behavior: the function api. If you use pointers

1. To improve speed and responsiveness software is increasing its reliance on concurrent programming models. However, most used platform languages were designed in an era before concurrent programming had fully formed and have often been more awkward to use than necessary. The proof of this is how easy it is to write dangerous, error prone asynchronous inter-thread code in most languages. Awkward, cumbersome, and error prone code is the systemic result of the language provided concurrency strategies, which detracts from their primary purpose of improving program efficiency.   
     
   This has problem has been improved in recent years by new language specific techniques, but I’ve typically still found them awkward, wordy, or so abstract they are hard to understand and implement readably.

The modern platform language Go on the other hand has provided a much simpler model: handle the specifics of concurrent management internally as part of a simple language keyword so that the programmer is freer to improve efficiency by designing better algorithms.

In summary:  
1. Data wants to be structured and stateful

2. Functions want to be functional and stateless

3. Computation wants to be fast, efficient, concurrent and seamless

Here are my *opinions* of what current languages solve each issue the best:

1. Structured Data: Essentially all object oriented languages (C++, Java, etc.) are effective at enabling data design in a way that helps to manage stateful data complexity. However, they typically struggle with complexity due mixing state and function design.   
     
   However, in my opinion while OOP methodology is good for designing data, it struggles to design functionality through modeling. IE, much more information is communicated through the model about the data being processed than the processing itself, to the point it’s not always clear what objects actually \*do\*.
2. Stateless Functions: Conversely, functional languages are better at writing code that communicates what a function does, but worse at writing code that illustrates what’s happening with the data involved, or how to handle data between widely separate parts of a program, like large event driven systems.  
     
   Functional paradigm programming languages (Haskell, Lisp) limit function design complexity by removing unnecessary state.
3. Seamless concurrency: Go and Qt framework both have strategies I like for addressing asynchronous computation, but they primarily emphasize different styles: Go is user function driven while Qt is event driven.

**3 Features:**

3 attempts to provide all 3 of these features in a simple, non-intrusive way while keeping each feature completely separate from the others. This separation is my attempt to help developers break down problems into smaller pieces, limiting complexity in design as well as making design more flexible.

My belief is that 3 can be converted to other languages and subsequently used alongside other frameworks. Usage of the library is designed to not be overbearing, ie, there is nothing stopping you from mixing functions and state, this library merely makes it possible not to do so. The features are:

**Datapools:** Hash tables of data and/or data objects accessible to the other features. This allows the data to be both mutable and separate from functions while still being accessible to said functions. They provide a place where carefully designed data can be stored and subsequently used drive the functionality of connected code.

**Stateless Functions and Testing:** Input/output redirection handling from/to Datapools allowing functions to be written as pure and stateless, removing a large element of complexity from the design of objects. A small testing suite is provided to validate said functions.

**Seamless Concurrency:** Computepools of managed threads and coroutine tasks. Executing a new task is as simple as using the provided (go) function. Also provides Qt-esque messaging and message handlers (similar to Signals and Slots)

**3 Library Structure Diagrams**













Efficient multithreading through a combination of traditional threads and coroutines:

A coroutine is a special kind of function that can pause while running, save its current state, and resume running later. Normally this is done with a (yield) call. By using coroutines and intelligently yielding it’s possible to have an asynchronous program running on a single thread of execution.

Computepools use a combination of threads and coroutines. My reason for starting with threads is hardware enhanced thread efficiency. However, as stated above coroutines are useful because they allow asynchronous behavior without blocking threads on tasks that evaluate for a long time.

Currently 3 only supports cooperative coroutine (yield)s. This means the programmer is in charge of knowing when to yield the current task. However, a future improvement might include moving to Engines, which yield after a timer timeout.

Each computepool thread has a task queue it can pull work from. When one thread runs out of tasks in their queue they pull from the fullest queue. Threads put themselves to sleep when there are no tasks and are woken up when given a new task.

A task can be added to a Computepool task queue by invoking the (go) command. The go command takes a reference to a Datapool, a suspended coroutine, and a list of instructions for how to handle function output.

The function (define-coroutine) will create a generator function that generates suspended coroutines. Take the below:

$ racket

Welcome to Racket v7.0.

> (require "3.rkt")

> (define-coroutine

(example-coroutine-generator x)

(printf "you gave me ~a" x))

> (define suspended-coroutine (example-coroutine-generator 5))

> (suspended-coroutine)

you gave me 5

>

Using the (go) function you can run it as an asynchronous task:

$ racket

Welcome to Racket v7.0.

> (require "3.rkt")

> (let ([datapool (make-datapool (make-computepool 2))])

(define-coroutine

(example-go-coroutine x)

(printf "you gave me ~a\n" x))

(go datapool (example-go-coroutine 27))

(wait-len datapool #f) ;this is a convenience function for making sure all tasks are completed before killing the threads

(close-dp dp))

> you gave me 27

Additionally 3 allows you to write message handlers. Messages and message handlers are roughly equivalent to Qt signals and slots.

$ racket

Welcome to Racket v7.0.

> (require "3.rkt")

> (let\* ([datapool (make-datapool (make-computepool 2))]

[example-message-type 'example-type]

[example-message (make-message example-message-type 6)])

(define-coroutine

(example-handler-coroutine msg)

(printf "this time you gave me ~a\n" (message-content msg)))

(register-message-handler

datapool

example-handler-coroutine ;use the generator instead of a suspended coroutine

example-message-type) ;handler is called for incoming messages of this type)

(send-message datapool example-message)

(wait-len datapool #f)

(close-dp dp))

#t

> this time you gave me 6

Qt primarily supports event driven callbacks while Go primarily supports programmer driven multitasking. I wrote 3 to have both because I think both are helpful for different tasks. Qt style callbacks are most useful for handling information that arrives at an unspecified time (IE, event driven programming). Go style multitasking is more useful when you’re trying to get performance improvements by creating asynchronous algorithms.

As stated before, co-routines can yield to allow execution of other tasks. Here’s an example with a computepool with only 1 worker thread showing how this works:

$ racket

Welcome to Racket v7.0.

> (require "3.rkt")

> (let ([dp (make-datapool (make-computepool 1))]

[ch (channel)])

(define-coroutine

(example-yield ch)

(printf "pre-yield\n")

(yield 0) ;yield value is arbitrary

(printf "~a\n" (ch-get ch)))

(define-coroutine

(another-coroutine ch)

(ch-put ch "hello-world!\n"))

(go dp (example-yield ch))

(go dp (another-coroutine ch))

(wait-len)

(close-dp dp)

(void))

> pre-yield

hello-world!

Problems with 3’s design

The primary problem with 3’s current design is that it creates a 4th realm of complexity, where connecting datapools to functions can be quite wordy and cumbersome (a complicated message handler can be a paragraph of text!). While I would argue that other asynchronous methodologies (outside of Go’s simplicity) may be just as cumbersome, it is a fact that this could be potentially improved. However, I don’t know of any other asynchronous strategy that attempts to clearly separate the different problems in computing into clear responsibilities as 3 does, making design easier by breaking the problem into smaller pieces.

It is relevant to note that 3 code can be written using \*just\* (make-datapool), (go), and (channels), making the code similarly clean to standard Go while still being a library with scoped usage instead of an entire language.