Design patterns in functional programming

It's not about monads.

Logary Analytics

Me me me

- Currently founding a Behavioural Analytics firm: Logary Analytics
- It will model user segments and user behaviours with causative analytics;
  - Highlighting what you can do more of to improve your key metrics
  - Highlighting what you should do less of to improve your key metrics

Expecto

An advanced testing library for F#

build passing | build passing | nuget v9.0.0 368.0k | f# 99.0% | License Apache 2.0 | Sponsor

Unit-, fuzz-stress and generative, parallel, async testing

tradera

On-prem to Google Cloud migration

SUAVE.io

F# Web server, co-author

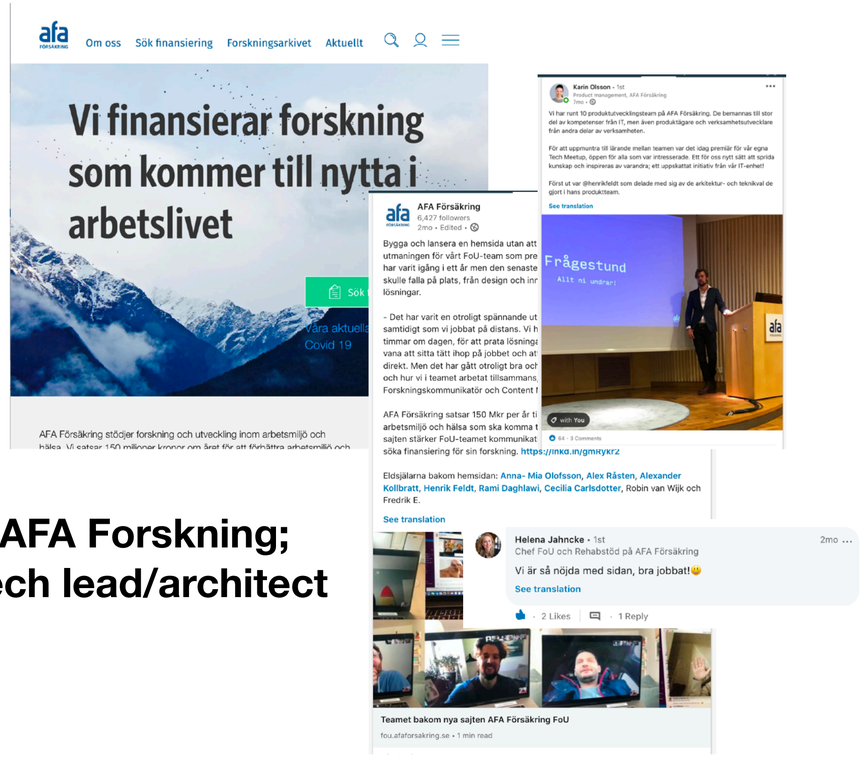
Qvitoo_

Founder, CEO, architect, etc...

Logary

Logging, metrics, tracing — used at Tradera in production in hot paths

FuncProgSTHLM founder



AFA Forskning; Tech lead/architect

RaySearch Laboratories

Built an HPC cluster for cancer treatment

voi.

Bootstrap engineering dept

Logary / Archivas

What's a design pattern?

A construction that repeats itself because it's useful and apt for the situation

— me

Categories of patterns

- Parsers and validation ("Parse don't validate")
- State mutation, handling ("Moore", "Mealy" and the monoids)
- Control flow ("await", "async" and the monads)
- Higher order functions

Some interesting patterns

- Delegator / hole-in-the-middle
- Simple state machines with mutually recursive functions
- Currying
- Dynamic configuration via closure
- Continuation passing style
- Scans and reductions / folds
- Cold observables / Alt/Job for retries

Hole in the middle

When you want to reuse a complex **function implementation**

Delegator / hole-in-the-middle

```
// currying and hole-in-the-middle
function colourise(configure: (config: RequestInit) ⇒ RequestInit):
  // returning a configured function
  (item: Item) ⇒ Promise<ColourisedItem> {

  // async-await (aka monadic control flow)
  return async (item: Item) ⇒ {
    const res = await fetch('/api/colours', configure({
      method: 'GET',
    }))

    const json = await res.json()
    return {
      ...item,
      colour: json.colour as string
    }
  }
}
```

Colourise is "advanced" in that it has many implementation details you don't want to leak.

Delegator / hole-in-the-middle usage

localToPage is some state only the caller knows

```
47  // currying usage
48  const localToPage = '0123456789ABCDEF'
49  const getColour = colourise(init ⇒ ({
50    ...init,
51    headers: {
52      ...init.headers,
53      'x-vanity-header': localToPage
54    }}))
55
```

Varying the usage of the complex implementation is easy now.

Logary Analytics

Currying

When you want to reuse a complex **setup**

Currying

```
// currying and hole-in-the-middle
function colourise(configure: (config: RequestInit) ⇒ RequestInit):
  // returning a configured function
  (item: Item) ⇒ Promise<ColourisedItem> {

  // async-await (aka monadic control flow)
  return async (item: Item) ⇒ {
    const res = await fetch('/api/colours', configure({
      method: 'GET',
    }))

    const json = await res.json()
    return {
      ...item,
      colour: json.colour as string
    }
  }
}
```

Colourise is pre-configured (can pre-compute CPU-intensive stuff or fetch from the network before the second red line)

Curring usage

getColour is a configured function

```
47    // currying usage
48    const localToPage = '0123456789ABCDEF'
49    const getColour = colourise(init ⇒ ({
50      ...init,
51      headers: {
52        ...init.headers,
53        'x-vanity-header': localToPage
54      }}))
55
```

So part of the cost of calling `colourise` is taken when you configure it, and then it's quick for subsequent callers.

Dynamic reconfiguration

E.g. your set of processors/ servers is changing continuously as your nodes go up and down

Dynamic reconfiguration via composition

Config

```
18      /// Null object pattern; will only return loggers that don't log.
        T  Henrik Feldt
19      let defaultConfig =
20        { getLogger = fun _ -> NullLogger.instance
21          getTimestamp = MonotonicClock.getTimestamp
22          consoleLock = Lock()
23          metrics = MetricRegistry() }
24
25      /// This is the "Global Variable" containing the last configured Logary
26      /// instance. If you configure more than one logary instance this will be
27      /// replaced.
        DVar<T>  Henrik Feldt
28      let internal configD = DVar.create defaultConfig
29
        DVar<IClock>  Henrik Feldt
30      let internal clockD =
          Henrik Feldt
31        let createClock config = { new IClock with member x.GetCurrentInstant() = Instant.ofEpoch (config.getTimestamp()) }
32        configD |> DVar.map createClock
```

Dynamic reconfiguration via composition Impl

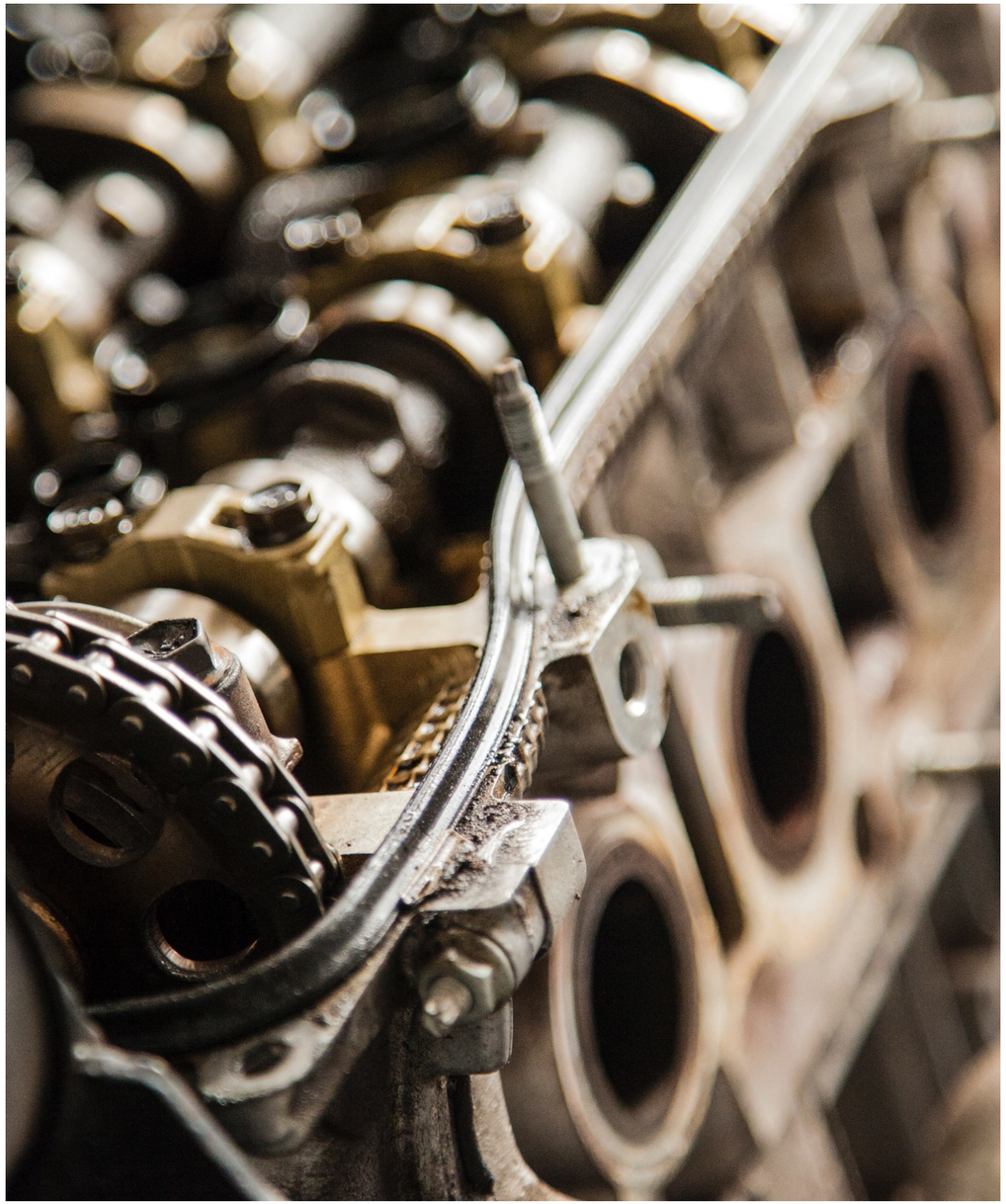
```fsharp
namespace Logary.Internals

open System
open System.Threading

type DVar<'a> = private { mutable cell : 'a; event : Event<'a> }

[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
module DVar =

  'a -> DVar<'a>
  let create (a:'a): DVar<'a> =
    { cell = a ; event = Event<'a>() }

  DVar<'a> -> 'a
  let get (d:DVar<'a>) : 'a =
    d.cell

  'a -> DVar<'a> -> unit
  let put (a:'a) (d:DVar<'a>) : unit =
    Interlocked.Exchange (&d.cell, a) |> ignore
    d.event.Trigger a

  DVar<'a> -> 'a -> unit
  let inline set d a =
    put a d
```

Dynamic reconfiguration via composition
Usage

```
type Flyweight(name: PointName) =
  DVar<Logger> Henrik Feldt
  let loggerD = configD |> DVar.map (fun cfg -> cfg.getLogger name)
  interface Logger with // flyweight
    PointName Henrik Feldt
    member x.name = name
    LogLevel Henrik Feldt
    member x.level =
      let logger = DVar.get loggerD in logger.level
    bool * LogaryMessage -> LogResult Henrik Feldt
    member x.logWithAck(putBufferTimeOut, message) =
      let logger = DVar.get loggerD
      logger.logWithAck(putBufferTimeOut, message)
```

Mutually recursive functions as a state machine

When you want to separate runtime states from each other



State machine w/ functions

```
FileConf -> Will<TargetMessage [] * uint16> -> TargetAPI -> Job<uni  Henrik Feldt
let loop (conf: FileConf) (will: Will<TargetMessage[] * uint16>) (api: TargetAPI) =
  let ri, rotateCh = api.runtime, Ch ()

  let shutdownState state = ...

  // In this state the File target opens its file stream and checks if it
  // progress to the recovering state.
  let rec init () = ...

  // In this state we try to write as many log messages as possible.
  and running (state: State): Job<unit> = ...

  // In this state we verify the state of the file.
  and checking (state: State) = ...

  // In this state we do a single rotation.
  and rotating (state: State) = ...

  and recovering (state: State) (lastBatch: TargetMessage[]) = function ...

  init ()
```

State machine w/ functions

```
65      // In this state we try to write as many log messages as possible.
66      and running (state: State): Job<unit> =
67        Alt.choose [
68          api.shutdownCh ^=> fun ack ->
69            shutdownState state >>=.
70            ack *<= ()
71
72          // TODO: handle scheduled rotation
73          rotateCh ^=> fun () -> rotating state
74
75          RingBuffer.takeBatch (conf.batchSize) api.requests ^=> fun reqs ->
76            writeRequestsSafe ri.logger conf state reqs >>= function
77              | Choice1Of2 () ->
78                checking state
79              | Choice2Of2 err ->
80                ri.logger.errorAck("IO Exception while writing to file. Batch size is {batchSize}.", fun m ->
81                  m.setField("batchSize", conf.batchSize)
82                  m.addExn err)
83                >>=. Will.update will (reqs, 1us)
84                >>=. Job.raises err
85        ] :> Job<_>
```

Logary Analytics

Finding more patterns

- Functional Pearls — haskell.org
- Read my code-bases; Expecto, LogaryJS, Logary, HttpFs — they all use these patterns
- Read Parse don't validate by Alexis King

- More interesting keywords:
  - Process algebra, pi-calculus, Kahn process networks, Mealy machines, Stateful functions (Flink), Stream processing, the Elm community

Questions?

🙏 for listening

Logary Analytics