Structuring Applications

…with the Capability Pattern

andrew.condon@gmail.com

A Problem of Complexity

Not easily summarized!

- It's really about all the stuff *around* the programming of your application
- How to test, how to partition work as a team, how to evolve code over time, how to deploy it, how to evolve how you deploy it etc etc
- In short, all the real world stuff that enters the picture when you have an application of almost any size that *other people rely on* in some way.

Preliminary remarks

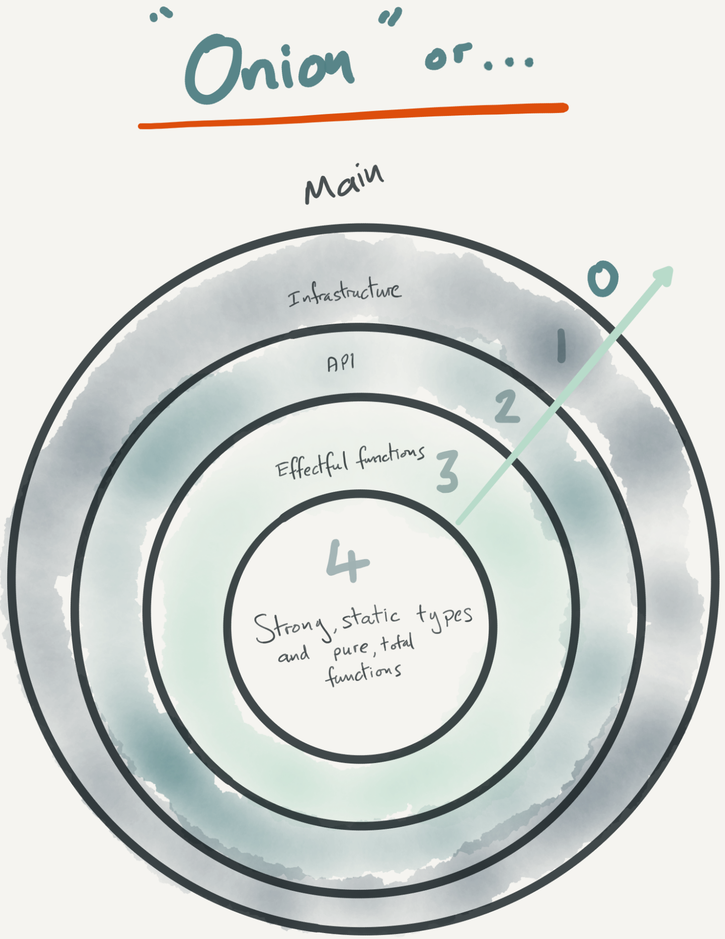
1. Assume it's preferable to take an "off the shelf" structure for your new app than design application structures from scratch, especially if you are new to the language
    - gives rise to Design Patterns, which are obviously very useful.
    - it can also be that "Design Patterns are signs of weakness in programming languages" (probably not the case here?)
2. Documenting design pattern is only first part of the work
    - second (more important?) part is defining its applicability

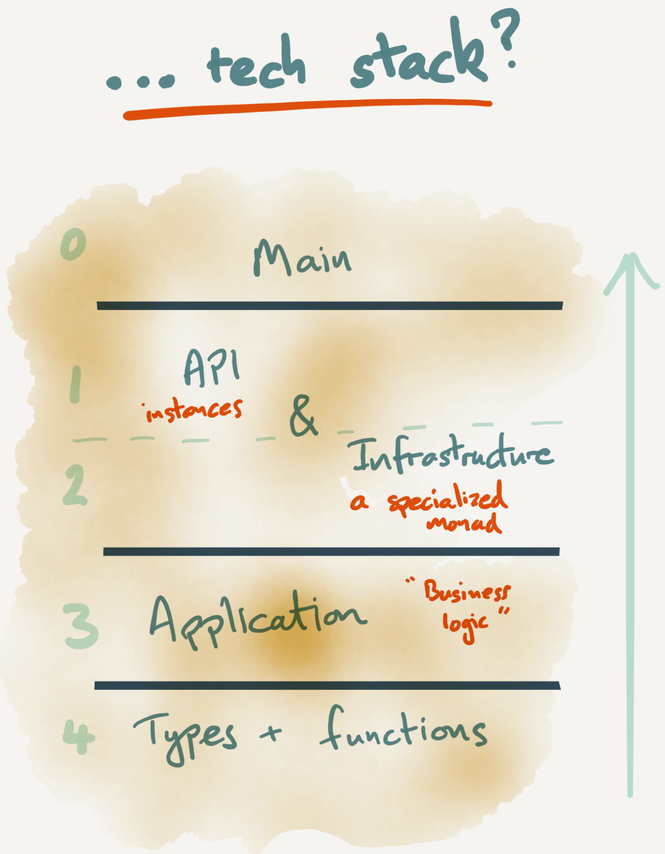
recipe + talk

discussion

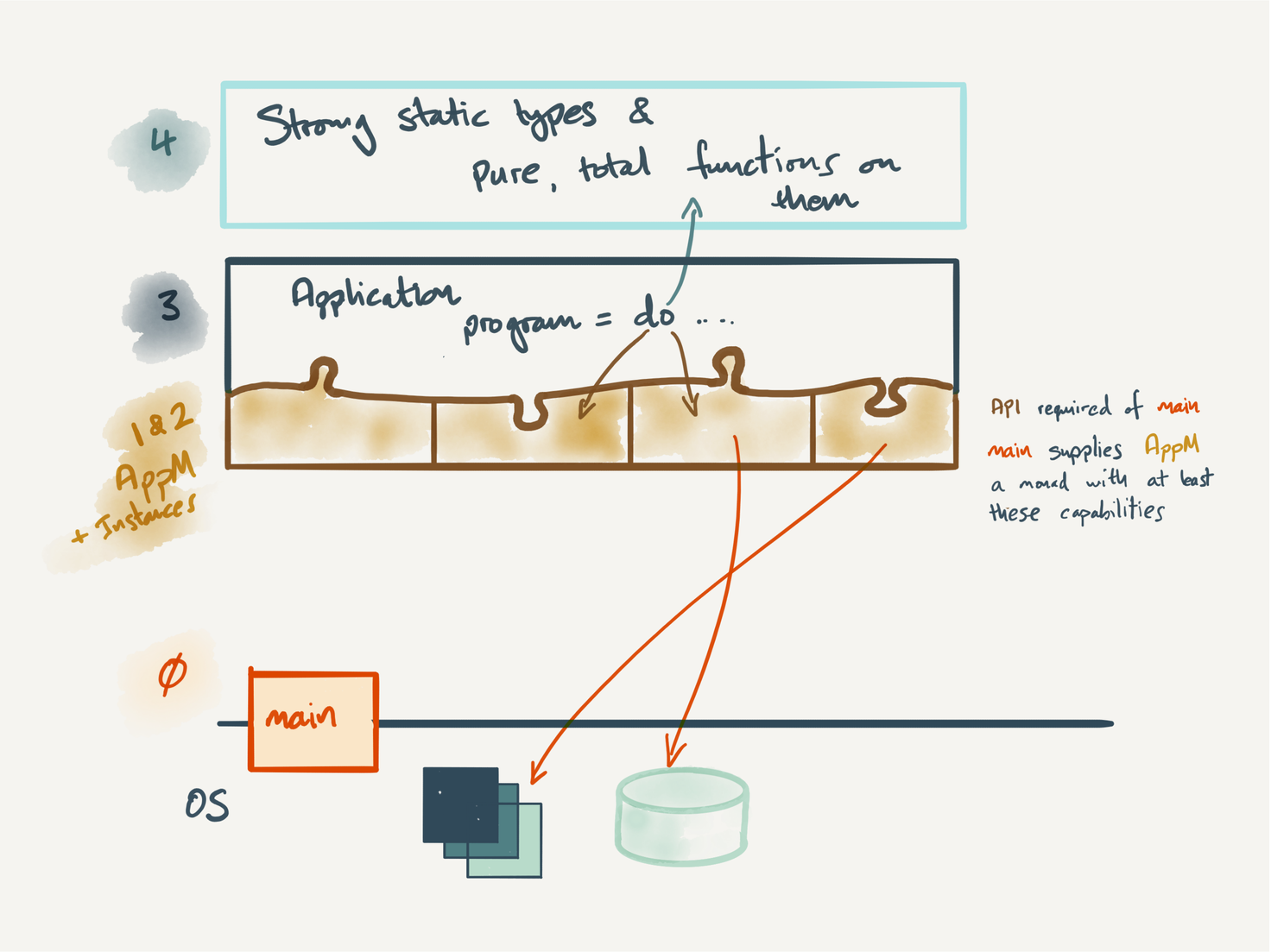
Visualizing the Onion

You can think of the layers however you prefer, it's the fact of separation that matters





I prefer to think of it something like this…



Start with the program you want to write

“layer 3”, part 1

```haskell
-- | a program that will run in _any_ monad that can fulfill the
-- | requirements (Logger and GetUserName)
program :: forall m.
    Logger m ⇒
    GetUserName m ⇒
    m String
program = do
    log "what is your name?"
    name ← getUserName
    log $ "Your name is " <> getName name
    pure $ getName name
```

Define some capabilities that it needs…

“layer 3”, part 2

```haskell
-- | Monads to define each capability required by the program
class (Monad m) <= Logger m where
    log :: String -> m Unit

class (Monad m) <= GetUserName m where
    getUserName :: m Name
```

…write pure code

“layer 4”

```haskell
-- | Layer 4
-- | Strong types & pure, total functions on those types
newtype Name = Name String

getName :: Name → String
getName (Name s) = s
```

NB: Ideally, there’s actually much, much more of this than the other layers but not in our simple example

…define (at least one) `AppM` to run application in

“layer 2”

```purescript
-- | Layer 2 Define our "Production" Monad...
type Environment = { productionEnv :: String }
newtype AppM a = AppM (ReaderT Environment Effect a)

-- | ...and the means to run computations in it
runApp :: forall a. AppM a → Environment → Effect a
runApp (AppM reader_T) env = runReaderT reader_T env
```

…provide capabilities for this Monad

“layer 1”

```purescript
-- | Layer 1 Provide instances for all capabilities needed
-- | Many of the instances are provided by deriving from the
-- | underlying ReaderT...
derive newtype instance functorAppM     :: Functor AppM
derive newtype instance applyAppM       :: Apply AppM
derive newtype instance applicativeAppM :: Applicative AppM
derive newtype instance bindAppM        :: Bind AppM
derive newtype instance monadAppM       :: Monad AppM
derive newtype instance monadEffectAppM :: MonadEffect AppM

-- | Reader instance not quite as simple a derivation as "derive newtype",
-- | as it needs TypeEquals for the env
instance monadAskAppM :: TypeEquals e Environment ⇒ MonadAsk e AppM where
  ask = AppM $ asks from

-- | implementing Logger here just to the console, but in real world you'd use
-- | the available Env to determine log levels, output destination, DB handles etc
instance loggerAppM :: Logger AppM where
  log = liftEffect <<< Console.log

-- | a version of getUserName that reads the name from a file
-- | given in the Environment
instance getUserNameAppM :: GetUserName AppM where
  getUserName = do
    env ← ask
    contents ← liftEffect $ Sync.readTextFile UTF8 env.productionEnv
    pure $ Name contents
```

…finally, run the `Application` in `AppM`

“layer 0”

```
main :: Effect Unit
main = do
  result ← runApp program { productionEnv: "Test" }
  pure unit
```

Now, we are able to run this “program” (or “application” / “business logic” / “layer 3”) in different monads which enables us to execute it with potentially radical differences in underlying resource usage, hiding async / call-backs by baking Aff into the AppM etc etc

Part Two

Checked Exceptions

Capability model works for Effect and Aff

…but what about Exceptions?

- when programming in the large could we make some sort of similar layering / separation of concerns for the exceptions raised?
- Nate Faubion's impressive `checked-exceptions` library can be combined with capability pattern
- i think this is very interesting but its utility is untested beyond this prototype / recipe style implementation

Extending our previous example

(which makes this kind of contrived but minimises the delta)

- We provide another AppM
  - One which is using resources with distinct failure modes represented by distinct exceptions
  - Borrowing the examples from the `checked-exceptions` README
    - an http service with `httpServerError`, `httpNotFound`, `httpOther` exceptions
    - a file system service with `fsPermissionDenied` & `fsFileNotFound` exceptions

Using these exception-raising services

Without re-engineering the “layer 3” code we wrote before

```
-- the type that we expect to be in the ReaderT as our environment
type Environment = { url :: String }

-- | Aff wrapped in ExceptV wrapped in ReaderT
newtype AppExcVM var a = AppExcVM (ReaderT Environment (ExceptV var Aff) a)
```

The AppExcVM Monad

Significantly more complicated but…remember application isn't seeing this

```
-- | ...and the means to run computations in it
runApp :: forall a. AppExcVM () a → Environment → Aff a
runApp = runAppExcVM (RProxy :: _ ())
  where
    runAppExcVM :: forall var rl.
      RowToList var rl ⇒
      VariantTags rl ⇒
      VariantShows rl ⇒
      RProxy var →
      AppExcVM var a →
      Environment →
      Aff a
    runAppExcVM _ (AppExcVM appExcVM) env = do
      ran ← runExceptT $ runReaderT appExcVM env
      case ran of
        Right result → pure result
        Left err     → throwError $ error $ show err
```

Here’s the very contrived bit

(not actually very likely to implement “getName” using http and fs like this)

```
instance getUserNameAppExcVM :: GetUserName (AppExcVM var) where
  getUserName = do
    env ← ask

    name ← safe $ (getPureScript env.url) #
                  handleError (errorHandlersWithDefault "there was an error!")

    pure $ Name name
```

A function that can raise exceptions of two types

(checked-exceptions is providing this unification of error types)

```
getPureScript :: forall m r
   .  Monad m
   ⇒ MonadHttp m
   ⇒ MonadFs m
   ⇒ String → ExceptV  (HttpError + FsError + r) m String
getPureScript url = do
  HTTP.get url >>= FS.write "~/purescript.html"
  pure "some result"
```

handlerErrors errorHandlersWithDefault

Illustrates how if all anticipated exceptions can be removed…

```
errorHandlersWithDefault :: forall m a.
  MonadEffect m ⇒
  a →
  { fsFileNotFound      :: String → m a
  , fsPermissionDenied :: Unit → m a
  , httpNotFound        :: Unit → m a
  , httpOther           :: { body :: String, status :: Int} → m a
  , httpServerError     :: String → m a
  }
```

…then we can use `safe` to remove the ExceptT

Thus isolating all of these exceptions from the two libraries

```
instance getUserNameAppExcVM :: GetUserName (AppExcVM var) where
  getUserName = do
    env ← ask

    name ← safe $ (getPureScript env.url) #
                  handleError (errorHandlersWithDefault "there was an error!")

    pure $ Name name
```

Running the program looks the same as before

Importantly, program hasn't changed, other AppM's haven't changed

```
-- | Layer 0 - Running the `program` in this context
main :: Effect Unit
main = launchAff_ do
  result ← AppExcVM.runApp program { url: "http://www.purescript.org"}
  pure unit
```

Discussion?