FP-SYD 2017

DEPENDENT TYPES, NOT JUST FOR VECTORS?

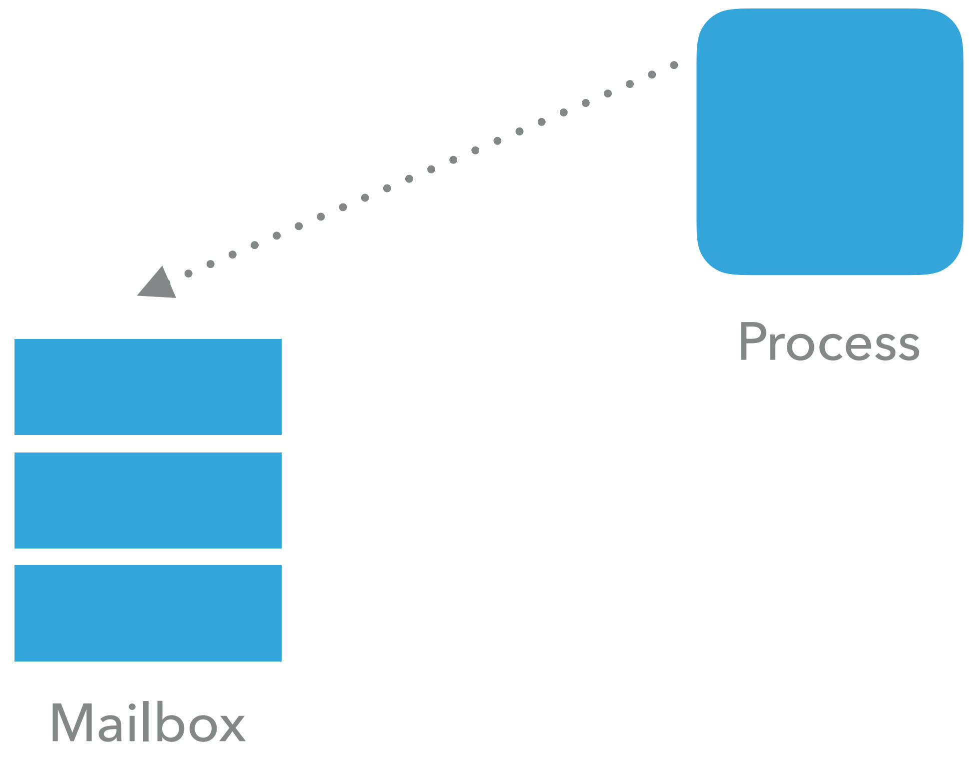
WHO AM I?

- Tim McGilchrist @lambda_foo
- Haskell programmer at Ambiata
- Curious about Distributed Systems
- Curious about Types

HOW DID I GET HERE?

BACKGROUND

ACTORS AND ERLANG




```
server_loop (state) ->
    receive
        {From, {update_balance, Amount, AccountId}} ->
            new_state = update_account (AccountId, Amount),
            From ! {self(), ok},
            server_loop (new_state)
    end.
```

SESSION TYPES

- Describe communication protocols
- Session types codify the structure of communication
- Data types codify the structures communicated


PROBLEMS

EFFECT SYSTEMS

- Available in Idris and Purescript

```
Effect : Type
Effect = (x : Type) -> Type -> (x -> Type) -> Type

data EFFECT : Type where
  MkEff : Type -> Effect -> EFFECT
```

- Use effects to model state machines.

EFFECT PROBLEMS

- "it was not possible to implement one effectful API in terms of others" E Brady
- "difficult to describe the relationship between separate resources" E Brady
- Composing problems?

IDRIS IS A PACMAN COMPLETE LANGUAGE

Edwin Brady

```
λΠ>:doc Vect

Data type Data.Vect.Vect : (len : Nat) -> (elem : Type) -> Type

   Vectors: Generic lists with explicit length in the type

   Arguments:

      len : Nat  -- the length of the list

      elem : Type  -- the type of elements

Constructors:

   Nil : Vect 0 elem

      Empty vector

   (::) : (x : elem) -> (xs : Vect len elem) -> Vect (S len) elem

      A non-empty vector of length S len, consisting of a head element and the rest of
      the list, of length len.
```

STATES ALL THE WAY DOWN

- "A useful pattern in dependently typed programming is to define a state transition system”
- “an architecture for dependently typed applications”
- “How to implement a state transition system as a dependent type “
- "How to combine state transition systems into a larger system"

GENERALISING STATEFUL PROGRAMS

- Types should capture the **states** of resources
- Stateful APIs should **compose**
- Types should be **readable**
- Error messages should be **readable**

ENTER STRANS

```
data STrans_ : (m : Type -> Type) -> (ty : Type) ->
  (in_ctxt : Resource) -> (out_ctxt : ty -> Resource) -> Type
```

- m - underlying monad
- ty - result type of the program
- in_ctxt - input context
- out_ctxt

DATASTORE

```
data Access = LoggedOut | LoggedIn
data LoginResult = OK | BadPassword

interface DataStore (m : Type -> Type) where
  Store : Access -> Type

  connect : STrans m Var [] (\store => [store ::: Store LoggedOut])
  disconnect : (store : Var) -> STrans m () [store ::: Store LoggedOut] (const [])

  login : (store : Var) ->
          STrans m LoginResult
                   [store ::: Store LoggedOut]
          (\res => [store ::: Store (case res of
                                          OK => LoggedIn
                                          BadPassword => LoggedOut)])

  logout : (store : Var) ->
           STrans m () [store ::: Store LoggedIn]
                (const [store ::: Store LoggedOut])
  readSecret : (store : Var) ->
           STrans m String [store ::: Store LoggedIn]
                  (const [store ::: Store LoggedIn])
```

USING DATASTORE

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do
  st <- connect
  OK <- login st
     | BadPassword => do putStrLn "Failure"
                         disconnect st
  secret <- readSecret st
  putStrLn ("Secret is: " ++ show secret)
  logout st
  disconnect st
```

```
- DataStore.idr line 32 col 2:
 When checking right hand side of Main.case block in getData at DataStore.idr:26:9 with expected type
         STrans m
                ()
                [st :::
                 Store (case OK of    OK => LoggedIn BadPassword => LoggedOut)]
                (\result1 => [])
```

TEXT

```
implementation DataStore IO where
  Store x = State String
  connect = do store <- new "Secret Data"
               pure store
  disconnect store = delete store
  readSecret store = read store
  login store = do putStr "Enter password: "
                   p <- getStr
                   if p == "Mornington Crescent"
                     then pure OK
                     else pure BadPassword
  logout store = pure ()
```

```
run {m = IO} getData
```

TYPES OF COMPOSITION

- Horizontally - multiple state machines within a function
- Vertically - implement state machine in terms of another

Examples:

- Application on a Communication Protocol
- Multiple resources, File IO plus State

STATE PLUS DATASTORE

```
getDataCount : (ConsoleIO m, DataStore m) =>
  (failcount : Var) -> ST m () [failcount ::: State Integer]
getDataCount failcount = do
  st <- call connect
  OK <- call $ login st
     | BadPassword => do putStrLn "Failure"
                         fc <- read failcount
                         write failcount (fc + 1)
                         putStrLn ("Number of failures: " ++ show (fc + 1))
                         call $ disconnect st
                         getDataCount failcount
  secret <- call $ readSecret st
  putStrLn ("Secret is: " ++ show secret)
  call $ logout st
  call $ disconnect st
  getDataCount failcount
```

```
call : STrans m t sub new_f ->
       {auto res_prf : SubRes sub old} ->
       STrans m t old (\res => updateWith (new_f res) old res_prf)
```

CLEANING UP THE TYPES

- Type level function ST

```
ST_ : (m : Type -> Type) -> (ty : Type) -> List (Action ty) -> Type
ST_ m ty xs = STrans m ty (in_res xs) (\result : ty => out_res result xs)
```

- List of actions on resources

```
data Action_ : Type -> Type where
  Stable : lbl -> Type -> Action_ ty
  Trans : lbl -> Type -> (ty -> Type) -> Action_ ty
  Remove : lbl -> Type -> Action_ ty
  Add : (ty -> Resources) -> Action_ ty
```

DATASTORE - CLEAN

```
data Access = LoggedOut | LoggedIn
data LoginResult = OK | BadPassword


interface DataStore (m : Type -> Type) where
  Store : Access -> Type

  connect : ST m Var [Add (\store => [store ::: Store LoggedOut])]
  disconnect : (store : Var) -> ST m () [Remove store (Store LoggedOut)]

  login : (store : Var) -> ST m LoginResult [ store ::: Store LoggedOut :->
                                 (\res => Store (case res of
                                                      OK => LoggedIn
                                                      BadPassword => LoggedOut))]

  logout : (store : Var) -> ST m () [store ::: Store LoggedIn :-> Store LoggedOut]
  readSecret : (store : Var) -> ST m String [store ::: Store LoggedIn]
```

PRETTY ERRORS

```
badGet : DataStore m => ST m () []
badGet = do
  st <- connect
  secret <- readSecret st
  ?more
```

```
When checking an application of function Control.ST.>>=:
        Error in state transition:
                Operation has preconditions: [st ::: Store LoggedIn]
                States here are: [st ::: Store LoggedOut]
                Operation has postconditions: \result => [st ::: Store LoggedIn]

                Required result states here are: \result =>
                                                   [st ::: Store LoggedIn]
```

CONCLUSION

- Need to tie this back to Actors.
- Encoding State Machines.
- Session Types
- Effect Systems

RESOURCES

- States All the Way Down, Edwin Brady
- Programming and Reasoning with Algebraic Effects and Dependent Types, Edwin Brady
- Session Types http://simonjf.com/2016/05/28/session-type-implementations.html
- Idris website http://docs.idris-lang.org/