Extensibly Free Arrows

LambdaConf 2018
Jake Keuhlen
https://github.com/jkeuhlen/talks

About me

- B.S. in Engineering Physics | ~5 years software experience
- Software Engineer at Holland and Hart LLP in Boulder
  - Building cutting edge automation systems for the legal world
  - We're hiring! Currently hiring another data scientist (Python,TensorFlow,etc) but also looking for additional Haskell developers for our backend.
- @jkeuhlen on Slack

HOLLAND & HART

https://github.com/jkeuhlen/talks

Roadmap

- Extensible Types
- Free Structures
- Arrows
- Building an Extensibly Free Arrow
- Utility

?

Feel free to ask questions at any time

https://github.com/jkeuhlen/talks

Extensible

- Closed Types
- ADT’s
- “It is very **cheap to add a new operation** on things: you just define a new function. All the old functions on those things continue to work unchanged.”
- “It is very **expensive to add a new kind of thing**: you have to add a new constructor [to] an existing data type, and you have to edit and recompile every function which uses that type.”

- Open Types
- Classes
- “It is very **cheap to add a new kind of thing**: just add a new subclass, and as needed you define specialized methods, in that class, for all the existing operations. The superclass and all the other subclasses continue to work unchanged.”
- “It is very **expensive to add a new operation on things**: you have to add a new method declaration to the superclass and potentially add a method definition to every existing subclass. In practice, the burden varies depending on the method.”

Quotes from @NormanRamsey (emphasis mine) via SO - https://stackoverflow.com/questions/870919/why-are-haskell-algebraic-data-types-closed

Extensible

- Extensible types are an easily definable type that pulls together groups of other, fully defined types.
- Extensible types attempt to solve the problems of both open and closed types by
  - defining minimal requirements for functions
  - defining easy ways to lift into extensible types
- Extensible constraints allow for reasoning more about the types of your functions

Extensible

Sum Types

- `type Bool = True | False`
- Sums are an OR type
  - They can only take on a single of their possible values at any time
- Either a b

Product Types

- `type Product = (String,Int)`
- Products are an AND type
  - They must contain values for all of their constituent types
- (,) a b

Extensible

- Sums

```
data (a :|: b) = DataL a | DataR b
deriving (Show, Eq)

class SumClass c s where
  peek :: c -> Maybe s
  lft  :: s -> c

type (w :>|: a)  = (SumClass w a)
```

- Products

```
data (a :&: b) = Prod a b deriving Show

class ProductClass c s where
  grab :: c -> s
  stash  :: s -> c -> c

type (c :>&: a)  = (ProductClass c a)
```

Extensible

- Heterogeneous lists
- Extensions to type class instances
- Type-level programming
- Complex and growing environments (state/reader)
- Replacement for monad transformers (extensible effects)

Free

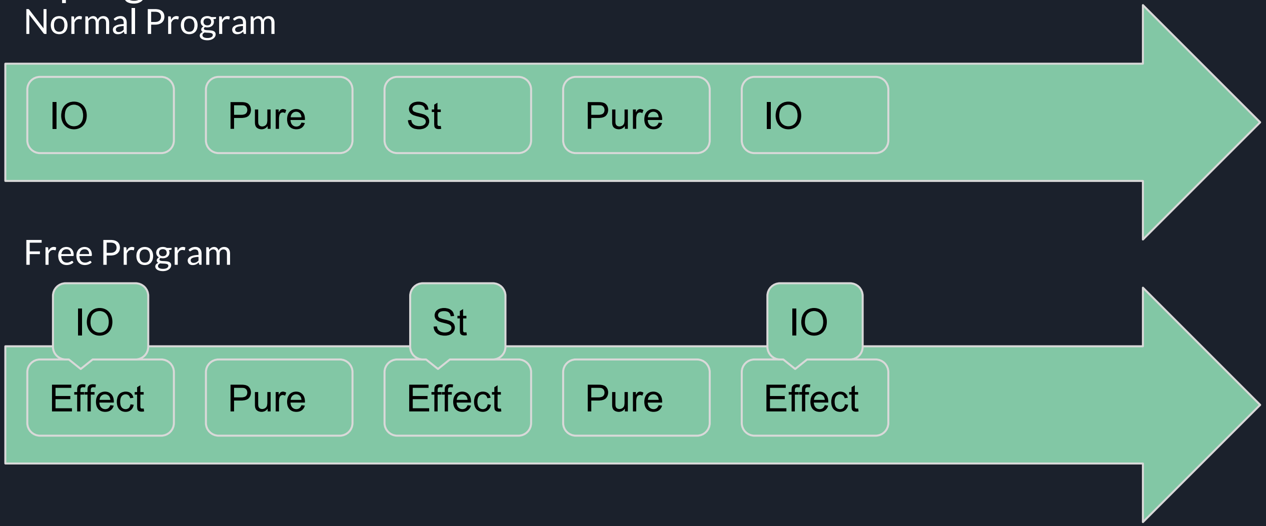
- A “Free” structure is the minimal definition of that structure.
  - This means it is the structure that just barely satisfies the structures laws
  - E.g. A Free Monad is the Monad that just barely satisfies the monad laws
- For every structure that has a set of defining laws, there exists a free object of that structure.

Free

- Free is a useful abstraction for separating side effects from your program
- It allows us to structure our programs as pure data structures which are open to interpretation by simply annotating where effects belong in a program

Normal Program

Free Program



Free

A (totally hypothetical, I swear this wasn't my fault) Example Case

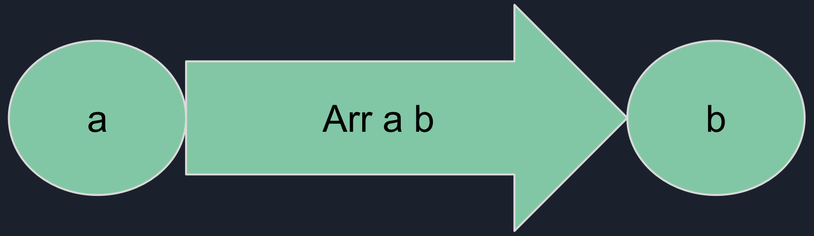
- You have a system that sends a set of automated emails
- Your system just sent automated spam to ~1,000 client email addresses due to some bug
- While investigating the bug, you trigger the spam again
- You have maybe 50 different places in your code that the email could have been sent from
- What's the fastest way to ensure that this never happens again?
- Change your interpreter!

Free

A (totally hypothetical, I swear this wasn't my fault) Example Case

- With a Free architecture, rather than having IO calls to sendmail peppered throughout the code, we have calls to an effect we have defined: SendMail
- Now we can simply adjust the interpreter for what SendMail means to prevent future problems
- Whenever we interpret a SendMail effect, we can purge all external email addresses

Arrows



- Arrows are generalizations of functions
- Arrows are useful for modelling time-sequential operations (circuitry, pipelines, etc.)
  - Arrows can be used for parallel computations as well
- Can be given additional typeclass instances (Applicative, Functor, Bifunctor) to make custom arrows even more powerful
- Arrows add an additional layer of structure on top of your computation
  - (Arrow arr) => arr a b vs (Monad m) => m b

Arrows

- Kleisli Arrows allow us to replicate any monad inside an arrow
- Arrows at first glance seem less powerful than Monads because we need more than just return and >>= to replicate monadic computations
- Free monads allow you to create different interpretations of effects
- Free arrows allow you to interpret entire programs differently

Extensibly Free Arrow

- An “Extensibly Free Arrow’ is a design pattern that combines all of these pieces.
- There are multiple ways to define each of the parts, each with varying pros and cons.
- We’ll walk through one approach using GADTs to encapsulate both the arrow structure and our free effects.
- Let’s build it!

FreeA - Definition

```haskell
data FreeA eff a b where
    Pure :: (a -> b) -> FreeA eff a b
    Effect :: eff a b -> FreeA eff a b
    Seq :: FreeA eff a b -> FreeA eff b c -> FreeA eff a c
    Par :: FreeA eff a1 b1 -> FreeA eff a2 b2 -> FreeA eff (a1, a2) (b1, b2)
    -- Apply -- | Arrow apply
    -- FanIn -- | Arrow Choice
    -- Spl   -- | Arrow Choice
```

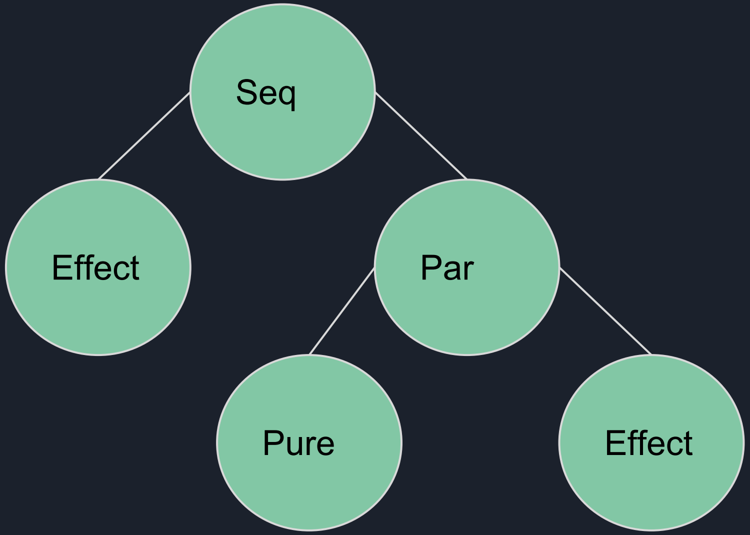
FreeA - Definition

```
instance C.Category (FreeA eff) where
    id = Pure id
    (.) = flip Seq

instance Arrow (FreeA eff) where
    arr = Pure
    first f = Par f C.id
    second f = Par C.id f
    (***) = Par
```

FreeA - Structure

```
Effect arrow1 >>> (Pure arrow2 *** Effect arrow3)
```



FreeA - Structure

- Free Effects can be *interpreted* into various real effects
- Free Effects allow you to separate the representation of effects from the running of effects

Com·pile : produce (something, especially a list, report, or book) by assembling information collected from other sources.

- Free Arrows have additional structure and can also be *compiled*
- Free Arrows allow you to separate the representation of your program from the running of your program

FreeA - Free Effects

```haskell
data PrintX a b where
  Print :: PrintX Text ()

interpPrintX :: (MonadIO m) => PrintX a b -> FreeA (Kleisli m) a b
interpPrintX Print = liftK (\x -> liftIO $ T.putStrLn x)

interpPrintXToFile :: (MonadIO m) => PrintX a b -> FreeA (Kleisli m) a b
interpPrintXToFile Print = liftK (\x -> liftIO $ T.writeFile "output.txt" x)
```

FreeA

```haskell
compileA :: forall eff arr a0 b0. (Arrow arr) => (forall a b. eff a b
-> arr a b) -> FreeA eff a0 b0 -> arr a0 b0
compileA exec = go
  where
    go :: forall a b . (Arrow arr) => FreeA eff a b -> arr a b
    go freeA = case freeA of
        Pure f -> arr f
        Seq f1 f2 -> go f2 C.. go f1
        Par f1 f2 -> go f1 *** go f2
        Effect eff -> exec eff
```

FreeA

```haskell
evalKleisliA :: forall m a b .
  ( Monad m ) => FreeA (Kleisli m) a b -> Kleisli m a b
evalKleisliA = go
  where
    go :: forall m a b . (Monad m) => FreeA (Kleisli m) a b -> Kleisli m a b
    go freeA = case freeA of
        Pure f -> Kleisli $ return . f
        Effect eff -> eff
        Seq f1 f2 -> go f2 C.. go f1
        Par f1 f2 -> go f1 *** go f2

liftK :: Monad m => (b -> m c) -> FreeA (Kleisli m) b c
liftK eff = Effect (Kleisli $ \x -> eff x)
```

Extensible

```haskell
data (f :+: g) a b =
       InL (f a b)
      | InR (g a b)
type (w :>+: a)  = (Sum2 w a)

lftEff :: (eff :>+: f)
  => FreeA f a b
  -> FreeA eff a b
lftEff = fmapEff lft2
```

```haskell
fmapEff :: forall b c eff1 eff2 .
  (forall bb cc . eff1 bb cc -> eff2 bb cc)
  -> FreeA eff1 b c -> FreeA eff2 b c
fmapEff fxn = go
  where
    go :: forall b c . FreeA eff1 b c -> FreeA
eff2 b c
    go (Effect eff) = Effect $ fxn eff
    go (Pure x) = Pure x
    go (Seq f1 f2) = go f2 C.. go f1
    go (Par f1 f2) = go f1 *** go f2
```

Combined

```
printA :: (eff :>+: PrintX) => FreeA eff Text ()
printA = lftE Print

storeA :: (eff :>+: StoreX) => FreeA eff String ()
storeA = lftE Store

extensibleArrow :: (eff :>+: PrintX, eff :>+: StoreX) => FreeA eff Text ()
extensibleArrow = proc x -> do
  printA -< x
  storeA -< T.unpack x
  Pure id -< ()
```

Combined

- Write isolated algebras for your different tools
- Combine them easily to build actual programs

Running FreeA

```
runKleisli (evalKleisliA $ compileA (interpPrintX <#>
interpStoreXToFile) extensibleArrow) ("Extensible Arrow" :: Text)
```

- Your effect order needs to be determined when your arrow is run
- Interpreters can be combined to form a larger, single interpreter for the compile step

Utility

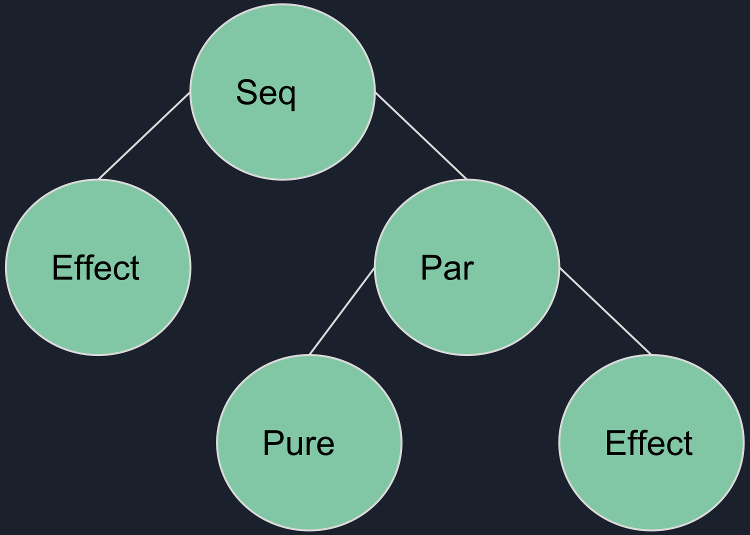
- Any workflows that make more sense using arrows over monads
- Free allows for maximal reuse of Operation Algebras
- Extensibility makes it simple to pull together multiple arrows
- Structure of arrows allows for other fun abstractions

Utility - Arrow Assembly

- See “How to Program like a Five Year Old in Haskell - λC 2017”
- Arrows can be programmatically fit together using their inputs and outputs as shapes to be matched

Utility - Parallelization

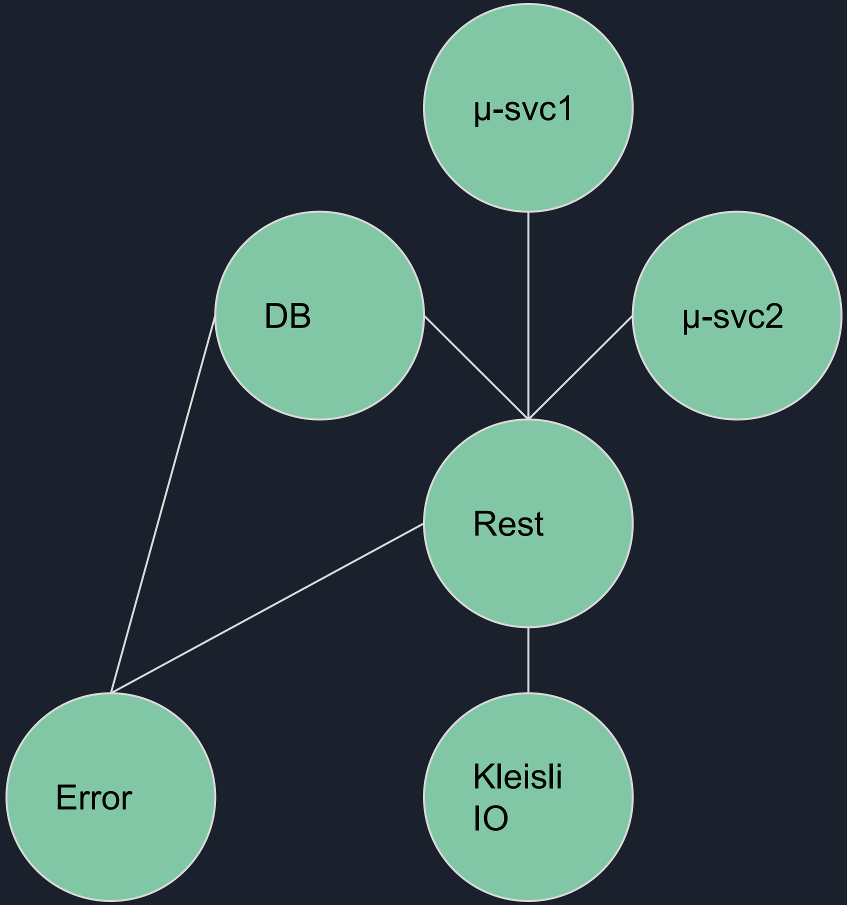
- When compiling FreeA, redefine what it means to Parallelize



Utility - Optimization

- When compiling FreeA, search for known bottlenecks and replace them in the tree with faster versions

Utility - Shared Interpretation



Formulaic Use

1. Determine your Operation Algebras
    a. Joust with the compiler until everything unifies
2. Write extensible arrows that utilize your Algebras
    a. Joust with the compiler until everything unifies
3. Combine extensible arrows to build programs
    a. Joust with the compiler until everything unifies
4. Compile your arrow
    a. Joust with the compiler until everything unifies
5. Run the compiled arrow

Final Thoughts

- Extensibly Free arrows combine all of the utility of free monads and extensible effects together into one super structure
- They can be painful in the final stage
- Free Monads are interpreted, free arrows can be compiled

Questions?