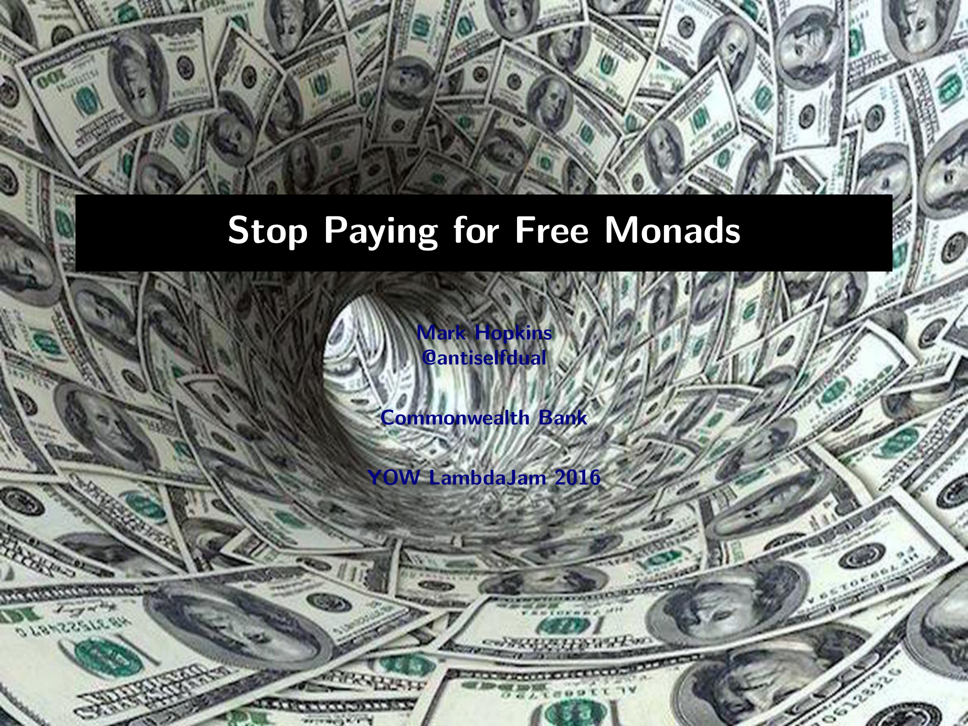


Stop Paying for Free Monads

Mark Hopkins
@antiselfdual

Commonwealth Bank

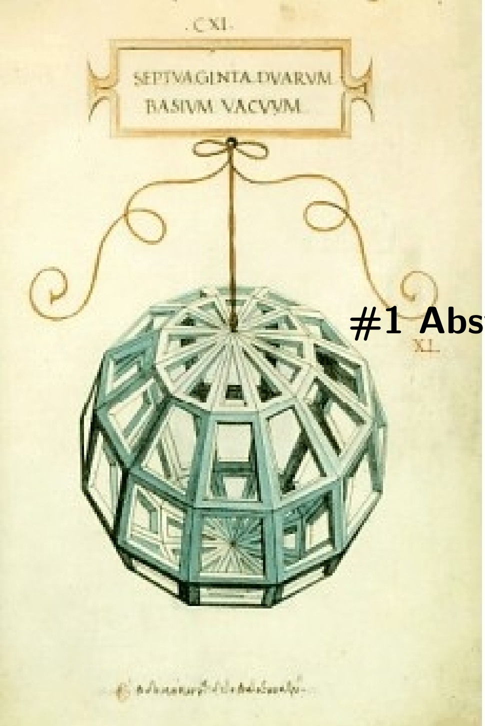
YOW LambdaJam 2016

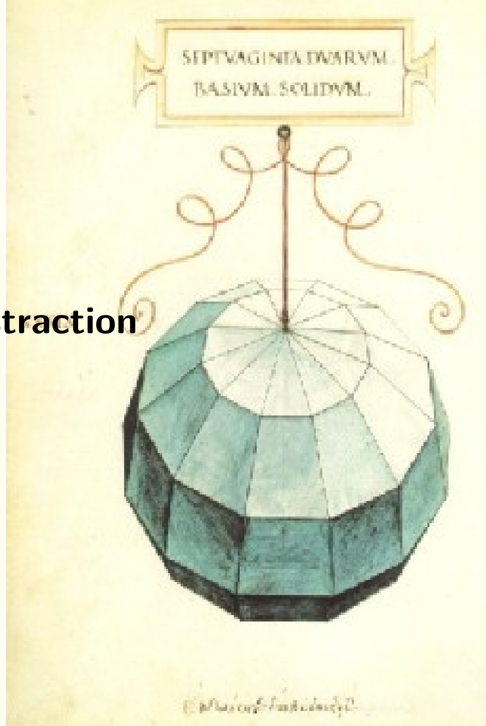
```haskell
prog tbl = do
  rows <- loadDb tbl
  dir <- mkTempDir
  report <- performAnalysis rows dir
  delete dir
  upload report s3Credentials $ "/reports/" ++ show tbl
  log $ "Wrote TPS report for " + show tbl
```

Abstraction and **modularity** for **effectful** programs.

Goals

#1 Abstraction





Interface/implementation separation

Rather than directly implementing our programs, I want to write a **specification** that I later **implement** or **interpret**.

Interface/implementation separation

Rather than directly implementing our programs, I want to write a **specification** that I later **implement** or **interpret**.

This allows me to

- swap out implementations

Interface/implementation separation

Rather than directly implementing our programs, I want to write a **specification** that I later **implement** or **interpret**.

This allows me to

- swap out implementations
- test using mock implementations

Interface/implementation separation

Rather than directly implementing our programs, I want to write a **specification** that I later **implement** or **interpret**.

This allows me to

- swap out implementations
- test using mock implementations
- use notions of interpretation, e.g.

Interface/implementation separation

Rather than directly implementing our programs, I want to write a **specification** that I later **implement** or **interpret**.

This allows me to

- swap out implementations
- test using mock implementations
- use notions of interpretation, e.g.
  - evaluation

Interface/implementation separation

Rather than directly implementing our programs, I want to write a **specification** that I later **implement** or **interpret**.

This allows me to

- swap out implementations
- test using mock implementations
- use notions of interpretation, e.g.
  - evaluation
  - pretty-printing

Interface/implementation separation

Rather than directly implementing our programs, I want to write a **specification** that I later **implement** or **interpret**.

This allows me to

- swap out implementations
- test using mock implementations
- use notions of interpretation, e.g.
  - evaluation
  - pretty-printing
  - serialization

Some different ways of viewing this:

- interface vs implementation

Some different ways of viewing this:

- interface vs implementation
- syntax vs semantics

Some different ways of viewing this:

- interface vs implementation
- syntax vs semantics
- a DSL with multiple interpreters

Some different ways of viewing this:

- interface vs implementation
- syntax vs semantics
- a DSL with multiple interpreters
- compiling a source language to a target language

Additionally, we want **low-cost abstraction**.

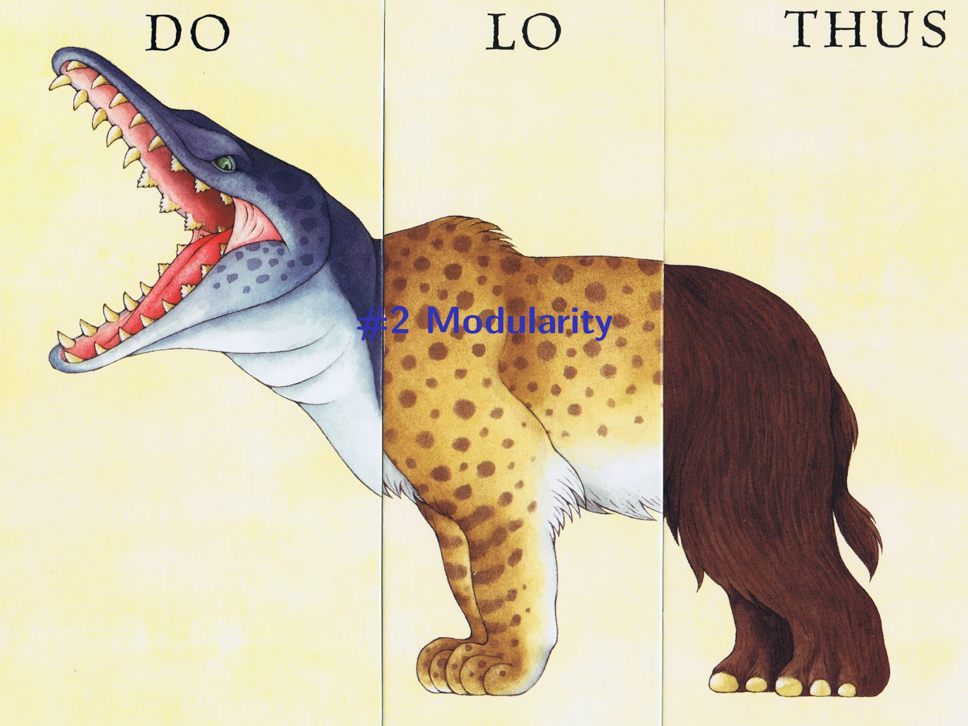
Additionally, we want **low-cost abstraction**.
We don't want to be penalised with

- poorly performing code

Additionally, we want **low-cost abstraction**.
We don't want to be penalised with

- poorly performing code
- code that's overly difficult to read or write.

#2 Modularity



I want **modular languages**.

I want **modular languages**.
I want the ability to

- refactor one big DSL into smaller DSLs

I want **modular languages**.

I want the ability to

- refactor one big DSL into smaller DSLs
- use multiple DSLs together in the same program

I want **modular languages**.

I want the ability to

- refactor one big DSL into smaller DSLs
- use multiple DSLs together in the same program
- extend a language by adding new operations

I want **modular languages**.

I want the ability to

- refactor one big DSL into smaller DSLs
- use multiple DSLs together in the same program
- extend a language by adding new operations
- implement a language in terms of a smaller core language.

I want **modular languages**.
I want the ability to

- refactor one big DSL into smaller DSLs
- use multiple DSLs together in the same program
- extend a language by adding new operations
- implement a language in terms of a smaller core language.

I want **modular languages**.
I want the ability to

- refactor one big DSL into smaller DSLs
- use multiple DSLs together in the same program
- extend a language by adding new operations
- implement a language in terms of a smaller core language.

So we need some kind of “addition of DSLs”.

I want **modular languages**.
I want the ability to

- refactor one big DSL into smaller DSLs
- use multiple DSLs together in the same program
- extend a language by adding new operations
- implement a language in terms of a smaller core language.

So we need some kind of “addition of DSLs”.

There should be a kind of **stability** too:
If I write a program, and then extend the language, my program should still be valid in the new language.

In addition, I also want **modular interpreters**.

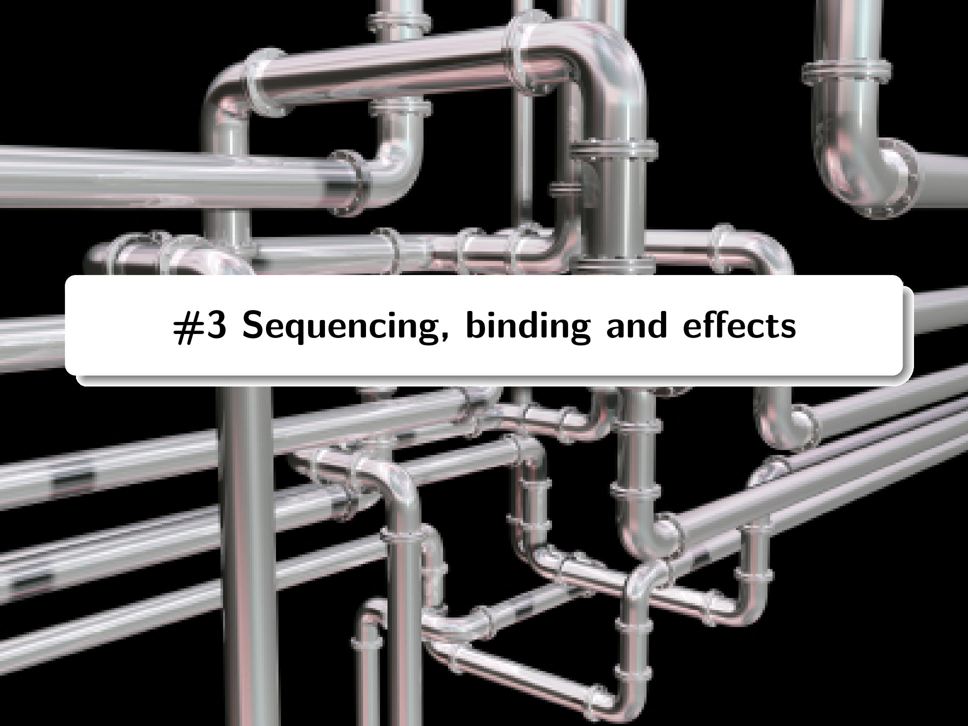
In addition, I also want **modular interpreters**.
If I write interpreters for a number of different languages. . .

In addition, I also want **modular interpreters**.
If I write interpreters for a number of different languages...

...I want to reuse them to interpret a program written using them together.

In other words, we need a solution of the *expression problem*.

#3 Sequencing, binding and effects



The (pure) functional programming vision

Better software through programs we can reason about.

The (pure) functional programming vision

Better software through programs we can reason about.

No matter how complex our programs their behaviour remains predictable.

The lambda calculus

Functions are values: their meaning does not depend on their context.

The lambda calculus

Functions are values: their meaning does not depend on their context.
The laws that function composition satisfies

```
f . id = f = id . f

f . (g . h) = (f . g) . h
```

allow us to reason about our code and to safety refactor.

i.e. **equational reasoning**.

How can we add side effects but retain predictability?

Moggi's insight was to *use the type system*.[1]

[1] Eugenio Moggi, Notions of computation as monads, 1991

Moggi's insight was to *use the type system*.[1]

Types will stand in for effects.

[1] Eugenio Moggi, Notions of computation as monads, 1991

Looking at our function type `f : a -> b`

Looking at our function type `f : a -> b`

there are three things we can modify:

Looking at our function type `f : a -> b`

there are three things we can modify:

- source `a`

Looking at our function type `f : a -> b`

there are three things we can modify:

- source a
- target b

Looking at our function type `f : a -> b`

there are three things we can modify:

- source `a`
- target `b`
- arrow `->`

Looking at our function type `f : a -> b`

there are three things we can modify:

- source `a`
- target `b`
- arrow `->`

Looking at our function type `f : a -> b`

there are three things we can modify:

- source `a`
- target `b`
- arrow `->`

This gives us three basic type classes for structuring computations

- comonads `w a ->   b`

Looking at our function type `f : a -> b`

there are three things we can modify:

- source `a`
- target `b`
- arrow `->`

This gives us three basic type classes for structuring computations

- comonads `w a ->   b`
- monads `  a -> m b`

Looking at our function type `f : a -> b`

there are three things we can modify:

- source `a`
- target `b`
- arrow `->`

This gives us three basic type classes for structuring computations

- comonads `w a ->   b`
- monads `  a -> m b`
- categories (and arrows) `  a >>>  b`

Equational reasoning

Reinstating the rules that we had for functions gives us the laws these have to obey:

```
f =>= extract = f = extract =>= f
f >=> return  = f =  return >=> f
f >>> id      = f =      id >>> f

(f =>= g) =>= h = f =>= (g =>= h)
(f >=> g) >=> h = f >=> (g >=> h)
(f >>> g) >>> h = f >>> (g >>> h)
```

Equational reasoning

Reinstating the rules that we had for functions gives us the laws these have to obey:

```
f =>= extract = f = extract =>= f
f >=> return  = f =  return >=> f
f >>> id      = f =      id >>> f

(f =>= g) =>= h = f =>= (g =>= h)
(f >=> g) >=> h = f >=> (g >=> h)
(f >>> g) >>> h = f >>> (g >>> h)
```

We can use these to refactor safely, i.e. we have equational reasoning in the presence of effects.

This generalises beyond side effects.
We use **effect** (or **context**) as a way of talking about the enhancement that `m a` brings over the raw type `a`.

We want to reuse these patterns that FP has evolved to deal with effects, sequencing and binding.

#4 Effects in the specification

Most literature talks about a pure specification with effectful interpreters.

#4 Effects in the specification

Most literature talks about a pure specification with effectful interpreters.
But sometimes, creating or handling effects belongs in the specification.

#4 Effects in the specification

Most literature talks about a pure specification with effectful interpreters.
But sometimes, creating or handling effects belongs in the specification.

e.g. we want to

- fail early if some condition is not met.

#4 Effects in the specification

Most literature talks about a pure specification with effectful interpreters.
But sometimes, creating or handling effects belongs in the specification.

e.g. we want to

- fail early if some condition is not met.
- clean up after some operation

#4 Effects in the specification

Most literature talks about a pure specification with effectful interpreters.
But sometimes, creating or handling effects belongs in the specification.

e.g. we want to

- fail early if some condition is not met.
- clean up after some operation

#4 Effects in the specification

Most literature talks about a pure specification with effectful interpreters.
But sometimes, creating or handling effects belongs in the specification.

e.g. we want to

- fail early if some condition is not met.
- clean up after some operation

And this needs to obey appropriate laws.


Solution 1: Reify

Datatypes à la carte, Wouter Swierstra, 2008

Motto:

> *Turn operations into constructors*

Let's start with a simple typed language of console interactions.

Let's start with a simple typed language of console interactions. Following our motto, let's express it as a data type.

```
data Console a where
  Ask  :: String -> Console String
  Tell :: String -> Console ()
```

Let's start with a simple typed language of console interactions. Following our motto, let's express it as a data type.

```haskell
data Console a where
  Ask  :: String -> Console String
  Tell :: String -> Console ()
```

Each operation becomes a constructor.

(co-)Yoneda embedding

CPS transform to get a functor:

```
data Console a =
    Ask String (String -> a)
  | Tell String a
    deriving Functor
```

Adding functors

So far, so good. What about modularity?

Idea: combining languages can literally be a **sum** of functors.

Adding functors

So far, so good. What about modularity?
Idea: combining languages can literally be a **sum** of functors.

```
data (f :+: g) a = Inl f a | Inr g a

instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap k (Inl fa) = fmap k fa
  fmap k (Inr ga) = fmap k ga
```

For instance we can break `Console` up

```
data Ask a = Ask String (a -> String) deriving Functor
data Tell a = Tell String a deriving Functor
```

For instance we can break `Console` up

```
data Ask a = Ask String (a -> String) deriving Functor
data Tell a = Tell String a deriving Functor

type Console = Ask :+: Tell
```

Free monads

We can make a monad out of any functor, using the “free monad” construction.[2]

[2]For more details, see Ken Scambler’s YLJ14 talk: “Run free with the monads: Free Monads for fun and profit”

Free monads

We can make a monad out of any functor, using the “free monad” construction.[2]
What is it, and where does it come from?

[2] For more details, see Ken Scambler's YLJ14 talk: “Run free with the monads: Free Monads for fun and profit”

Free monads

We can make a monad out of any functor, using the “free monad” construction.[2]
What is it, and where does it come from?

It turns out that being “free” is closely connected to our idea of operations as data types.

[2] For more details, see Ken Scambler's YLJ14 talk: “Run free with the monads: Free Monads for fun and profit”

Can we use "operations into constructors" to turn the `Monad` class into a data type?

Can we use "operations into constructors" to turn the `Monad` class into a data type?

Yes!

Can we use “operations into constructors” to turn the Monad class into a data type?

Yes!

```
class Monad m where
  return :: a -> m a
  join :: m (m a) -> m a
```

Can we use "operations into constructors" to turn the Monad class into a data type?

Yes!

```
class Monad m where
  return :: a -> m a
  join :: m (m a) -> m a

data Free f a where
  Return :: a -> Free f a
  Join :: f (Free f a) -> Free f a
```

```haskell
data Free f a = Return a
              | Join (f (Free f a))
```

`Free f a` is a monad whenever `f` is a functor.

```haskell
data Free f a = Return a
              | Join (f (Free f a))
```

`Free f a` is a monad whenever `f` is a functor.

It's the "naïve free monad."

```haskell
data Free f a = Return a
              | Join (f (Free f a))
```

`Free f a` is a monad whenever `f` is a functor.

It's the "naïve free monad."

What exactly are we claiming when we say it's the "free monad on `f`"?

A small digression into abstract nonsense

Categories have **objects**, and **morphisms** between objects.

Haskell types

Haskell types form a category: a morphism between two types is just a function.

```
f :: a -> b
```

Functors

Functors between Haskell types forms a category, where a morphisms between two functors is a **natural transformation**. A natural transformation is just a polymorphic function

```
n :: f a -> g a
```

Monads

Monads on Haskell types also form a category.

Monads

Monads on Haskell types also form a category.

A monad is a functor with some additional structure (`>>=` and `return`).

Monads

Monads on Haskell types also form a category.

A monad is a functor with some additional structure (>>= and `return`).
So a morphism between monads will be a natural transformation preserving that structure:

```
n . (f >=> g) = (n . f) >=> (n . g)
n . return    = return
```

where

```
(f >=> g) a = f a >>= g
```

The correct notion of an "interpretation" of a monad M_1 into another monad M_2 is just a monad morphism $M_1 \to M_2$.

The correct notion of an "interpretation" of a monad M_1 into another monad M_2 is just a monad morphism $M_1 \to M_2$.

In fact this is important for what will follow.

The correct notion of an "interpretation" of a monad M_1 into another monad M_2 is just a monad morphism $M_1 \to M_2$.

In fact this is important for what will follow.

It means if we write a program in the free monad, then translate, we can be sure we continue to obey the monad laws.

Otherwise, we'll lose predictability i.e. the ability to reason about our code.

Adjunctions

Usually in Haskell, functors go from `Hask` to `Hask`.
But in general, functors go from one category $\mathcal{C}$ to another $\mathcal{D}$.

Adjunctions

Usually in Haskell, functors go from Hask to Hask.
But in general, functors go from one category $\mathcal{C}$ to another $\mathcal{D}$.

An **adjunction** is the name for the situation where we have functors $L :: \mathcal{C} \to \mathcal{D}$, $R :: \mathcal{D} \to \mathcal{C}$. satisfying

$$\mathcal{D}(Lc, d) \cong \mathcal{C}(c, Rd)$$

for all objects c in $\mathcal{C}$ and d in $\mathcal{D}$.

$\mathcal{C}(c_1, c_2) =$ the collection of $\mathcal{C}$ morphisms from c_1 to c_2.
$\mathcal{D}(d_1, d_2) =$ the collection of $\mathcal{D}$ morphisms from d_1 to d_2.

When R is an inclusion of $\mathcal{C}$ into $\mathcal{D}$ we write $F = L$ and say

- F is the "free $\mathcal{D}$ functor on $\mathcal{C}$"

When R is an inclusion of $\mathcal{C}$ into $\mathcal{D}$ we write $F = L$ and say

- F is the "free $\mathcal{D}$ functor on $\mathcal{C}$"
- Fc the "free $\mathcal{D}$ on c".

When R is an inclusion of $\mathcal{C}$ into $\mathcal{D}$ we write $F = L$ and say

- F is the "free $\mathcal{D}$ functor on $\mathcal{C}$"
- Fc the "free $\mathcal{D}$ on c".

When R is an inclusion of $\mathcal{C}$ into $\mathcal{D}$ we write $F = L$ and say

- F is the "free $\mathcal{D}$ functor on $\mathcal{C}$"
- Fc the "free $\mathcal{D}$ on c".

This gives us a simple description of all $\mathcal{D}$ morphisms out of Fc:

$$\mathcal{D}(Fc, d) \cong \mathcal{C}(c, d)$$

When R is an inclusion of $\mathcal{C}$ into $\mathcal{D}$ we write $F = L$ and say

- F is the "free $\mathcal{D}$ functor on $\mathcal{C}$"
- Fc the "free $\mathcal{D}$ on c".

This gives us a simple description of all $\mathcal{D}$ morphisms out of Fc:

$$\mathcal{D}(Fc, d) \cong \mathcal{C}(c, d)$$

they just correspond to the $\mathcal{C}$ morphisms out of c.

In our case, our adjunction looks like

$$\mathsf{Mnd}(\mathsf{Free}\,F, M) \cong \mathsf{Func}(F, M)$$

In our case, our adjunction looks like

$$\mathsf{Mnd}(\mathsf{Free}\,F, M) \cong \mathsf{Func}(F, M)$$

If F breaks up into a sum of functors ...

$$F = F_1 + \cdots + F_k,$$

In our case, our adjunction looks like

$$\mathsf{Mnd}(\mathsf{Free}\, F, M) \cong \mathsf{Func}(F, M)$$

If F breaks up into a sum of functors ...

$$F = F_1 + \cdots + F_k,$$

then

$$\begin{aligned} & \mathsf{Mnd}(\mathsf{Free}\,(F_1 + \cdots + F_k), M) \\ \cong \;& \mathsf{Func}(F_1 + \cdots + F_k, M) \\ \cong \;& \mathsf{Func}(F_1, M) \times \cdots \times \mathsf{Func}(F_k, M) \end{aligned}$$

In other words to interpret a program written in the free monad of a sum of languages $F_1, \ldots, F_k$

In other words to interpret a program written in the free monad of a sum of languages F_1, . . . , F_k

we just have to give a (functor) interpretation for each language.

In other words to interpret a program written in the free monad of a sum of languages $F_1, \ldots, F_k$

we just have to give a (functor) interpretation for each language.

So this is the sense in which we have modular interpreters.

Example

```haskell
data Ask a        = Ask String (String -> a)
data Tell a       = Tell String a
data Lookup k v a = Lookup k ((Maybe v) -> a)
```

```haskell
checkQuota :: Free (Ask :+: (Lookup String Int) :+: Tell) ()
checkQuota = do
  name <- Join . Inl $ Ask "What's your name" Return
  quota <- Join . Inr . Inl $ Lookup name Return
  Join . Inr . Inr $ Tell
    (   "Hi "
     ++ name ++ ", your quota is "
     ++ show (fromMaybe 0 quota)
    )
    (Return ())
```

```haskell
checkQuota :: Free (Ask :+: (Lookup String Int) :+: Tell) ()
checkQuota = do
  name <- Join . Inl $ Ask "What's your name" Return
  quota <- Join . Inr . Inl $ Lookup name Return
  Join . Inr . Inr $ Tell
    (   "Hi "
     ++ name ++ ", your quota is "
     ++ show (fromMaybe 0 quota)
    )
    (Return ())
```

The injections Inl and Inr break our stability goal!

```haskell
checkQuota :: Free (Ask :+: (Lookup String Int) :+: Tell) ()
checkQuota = do
  name <- Join . Inl $ Ask "What's your name" Return
  quota <- Join . Inr . Inl $ Lookup name Return
  Join . Inr . Inr $ Tell
    (   "Hi "
     ++ name ++ ", your quota is "
     ++ show (fromMaybe 0 quota)
    )
    (Return ())
```

The injections Inl and Inr break our stability goal!

We'll have to modify the embeddings *everywhere* if we want to add another language.

Solution: polymorphism!

Introduce a type class to manage the injections for us

```
class f :<: g where
  inj :: f a -> g a
```

Solution: polymorphism!

Introduce a type class to manage the injections for us

```
class f :<: g where
  inj :: f a -> g a
```

Add instances to capture composition of injections.

```
instance f :<: f where ...
instance f :<: (f :+: g) where ...
instance (f :<: h) => f :<: (g :+: h) where ...
```

A polymorphic injection function

```
inject :: (f :<: g) => f (Free g a) -> Free g a
inject = Join . inj
```

A polymorphic injection function

```
inject :: (f :<: g) => f (Free g a) -> Free g a
inject = Join . inj
```

Rewrite our program in terms of smart constructors

```
ask :: Ask :<: f => String -> Free f String
ask s = inject $ Ask s Return
```

Invisible abstraction

```haskell
checkQuota :: Free (Ask :+: Lookup String Int :+: Tell) ()
checkQuota = do
  name <- ask "What is your name?"
  quota <- lookup name
  tell $ "Hi " ++ name ++ ", your quota is " ++
    (show (fromMaybe 0 quota))
```

Invisible abstraction

```haskell
checkQuota ::
(Ask :<: f, Lookup String Int :<: f, Tell :<: f) =>
Free f ()
checkQuota = do
  name <- ask "What is your name?"
  quota <- lookup name
  tell $ "Hi " ++ name ++ ", your quota is " ++
    (show (fromMaybe 0 quota))
```

Interpretation

```haskell
class (Functor f, Monad m) => Interpret f where
  intp :: f a -> m a

instance (Interpret f m, Interpret g m) =>
  Interpret (f :+: g) m where
    intp (Inl fa) = intp fa
    intp (Inr ga) = intp ga

interpret :: Interpret f m => Free f a => m a
interpret (Return a) = return a
interpret (Join fa) = join $ intp (interpret <$> fa)
```

Interpretation

```haskell
class (Functor f, Monad m) => Interpret f where
  intp :: f a -> m a

instance (Interpret f m, Interpret g m) =>
  Interpret (f :+: g) m where
    intp (Inl fa) = intp fa
    intp (Inr ga) = intp ga

interpret :: Interpret f m => Free f a => m a
interpret (Return a) = return a
interpret (Join fa) = join $ intp (interpret <$> fa)
```

interpret is a monad morphism.

```haskell
type MyMonad = ReaderT (Map String Int) IO

instance Interpret Ask MyMonad where ...
instance Interpret Tell MyMonad where ...
instance Interpret (Lookup String Int) MyMonad where ...
```

Let's run our program in `ReaderT (Map String Int) IO`.

```
> runReaderT (interpret checkQuota) $
 [ ("Stu", 33), ("Ann", 55) ]
What's your name?
Ann
Hi Ann, your quota is 55
```

Specification-side effects

It's not clear how we could easily add effects like early termination to our program: we'd need to create a `FreeError f`.

i.e. an analogy of Free f but for `MonadFree`, obeying the appropriate laws and satisfying a universal property.

Other binding constructs

- If we wanted to use comonads instead, we could use cofree comonads.[3]

  We'd have to invent "codata types à la carte".

[3] See Dave Laing's YLJ15 talk "Cofun with Cofree Monads"

Other binding constructs

- If we wanted to use comonads instead, we could use cofree comonads.[3]

  We'd have to invent "codata types à la carte".
- If we wanted arrows, it's not clear what to do.

[3] See Dave Laing's YLJ15 talk "Cofun with Cofree Monads"

Other binding constructs

- If we wanted to use comonads instead, we could use cofree comonads.[3]

  We'd have to invent "codata types à la carte".
- If we wanted arrows, it's not clear what to do.

[3] See Dave Laing's YLJ15 talk "Cofun with Cofree Monads"

Other binding constructs

- If we wanted to use comonads instead, we could use cofree comonads.[3]

  We'd have to invent "codata types à la carte".
- If we wanted arrows, it's not clear what to do.

  Although free arrows ought to exist, we don't yet have a Haskell implementation.

[3] See Dave Laing's YLJ15 talk "Cofun with Cofree Monads"

Limitations

- It's slow.

Limitations

- It's slow.
- It's complex. We have considerable boilerplate: injections, smart-constructors, interpreters.

Limitations

- It's slow.
- It's complex. We have considerable boilerplate: injections, smart-constructors, interpreters.
- ```
  {-# LANGUAGE TypeOperators         #-}
  {-# LANGUAGE DataKinds             #-}
  {-# LANGUAGE RankNTypes            #-}
  {-# LANGUAGE DeriveFunctor         #-}
  {-# LANGUAGE StandaloneDeriving    #-}
  {-# LANGUAGE MultiParamTypeClasses #-}
  {-# LANGUAGE FlexibleContexts      #-}
  {-# LANGUAGE FlexibleInstances     #-}
  {-# LANGUAGE UndecidableInstances  #-}
  {-# LANGUAGE OverlappingInstances  #-}
  ```
  ```

Limitations

- It's slow.
- It's complex. We have considerable boilerplate: injections, smart-constructors, interpreters.
-
  ```
  {-# LANGUAGE TypeOperators         #-}
  {-# LANGUAGE DataKinds             #-}
  {-# LANGUAGE RankNTypes            #-}
  {-# LANGUAGE DeriveFunctor         #-}
  {-# LANGUAGE StandaloneDeriving    #-}
  {-# LANGUAGE MultiParamTypeClasses #-}
  {-# LANGUAGE FlexibleContexts      #-}
  {-# LANGUAGE FlexibleInstances     #-}
  {-# LANGUAGE UndecidableInstances  #-}
  {-# LANGUAGE OverlappingInstances  #-}
  ```
- No backtracking in Haskell's type solver means for `:<:` we can only have nesting on one side of `:+:`

Solution 2: Operations remain functions

Oleg Kiselyov, Typed Tagless Final Interpreters, 2012

Motto

Let the language do the work

A language is a type class parametrized by a data constructor.

A language is a type class parametrized by a data constructor.
i.e. the parameter has kind `* -> *`.

A language is a type class parametrized by a data constructor. i.e. the parameter has kind `* -> *`.

```
class Console r where
  ask  :: String -> r String
  tell :: String -> r ()
```

A language is a type class parametrized by a data constructor. i.e. the parameter has kind `* -> *`.

```
class Console r where
  ask  :: String -> r String
  tell :: String -> r ()

class KeyVal k v r where
  lookup :: k -> r (Maybe v)
  set    :: k -> v -> r ()
```

An interpreter is a type with an instance.

```haskell
instance MonadIO m => Console m where
  ask s = putStrLn s >> getLine
  tell = putStrLn
```

An interpreter is a type with an instance.

```haskell
instance MonadIO m => Console m where
  ask s = putStrLn s >> getLine
  tell = putStrLn

instance (Ord k, MonadState (Map k v) m) =>
  KeyVal k v m where
    lookup = gets . DM.lookup
    set k v = modify $ DM.insert k v
```

Adding languages simply means adding constraints.

It's easy to split languages.

It's easy to split languages.

```
class Lookup k v where
  lookup :: k -> r (Maybe v)

class Set k v r where
  set :: k -> v -> r ()
```

It's easy to split languages.

```haskell
class Lookup k v where
  lookup :: k -> r (Maybe v)

class Set k v r where
  set :: k -> v -> r ()
```

... and to combine them.

```haskell
type KeyVal k v r = (Lookup k v r, Set k v r)
```

Requires `-XConstraintKinds`.

So our languages are modular.

We have modular interpreters just by adding new instances for our type.

For a monadic computation, just add a `Monad` constraint.

For a monadic computation, just add a Monad constraint.

```haskell
getQuota ::
( Console r
, Lookup String Int r
, Monad r
) => r ()
getQuota = do
  name <- ask "What's your name?"
  quota <- fromMaybe 0 <$> lookup name
  tell $ "Hi " + name + ", your quota is " ++ show quota
```

Adding effects to the specification is easy.
Just add a different constraint in place of `Monad`.

Adding effects to the specification is easy.
Just add a different constraint in place of Monad.

```
changePwd ::
( Console r
, KeyVal String String r
, MonadThrow r
) => r ()
changePwd = do
  n <- ask "What's your name?"
  p <- ask "What's your password?"
  matches <- (== Just p) <$> lookup n
  unless matches $ throwM WrongPassword
  np <- ask "Enter new password"
  np2 <- ask "Re-enter new password"
  unless (np == np2) $ throwM PasswordsDidNotMatch
  put n np
  tell "Password successfully updated"
```

To have a comonadic or arrowized computation, just add the relevant constraint.

Interpretation is the identity function.

Interpretation is the identity function.
i.e. just select a type (that has an instance for all constraints).

Let's run our program in `StateT (Map String String) IO`.

```
> runStateT changePwd $
  fromList [
    ("anne", "pwd123")
  , ("mark", "p@ssw0rd")
  ]

What's your name?
mark
What's your password?
p@ssw0rd
Enter new password
1337
Re-enter new password
1337
((),fromList [("sue","pwd123"),("mark","1337")])
```

Comparison

À la carte

Free monads over sums of functors.

Interpretation is a monad morphism, assembled in a modular fashion from interpretations (natural transformations) for each component language.

Typed tagless

Languages are (higher-kinded) type classes.
We combine languages by adding constraints.
Interpretations are types with the necessary instances.

Compared to à la carte, the tagless approach

- has minimal runtime overhead

Compared to à la carte, the tagless approach

- has minimal runtime overhead
- has no need for boilerplate

Compared to à la carte, the tagless approach

- has minimal runtime overhead
- has no need for boilerplate
- requires fewer language extensions

Compared to à la carte, the tagless approach

- has minimal runtime overhead
- has no need for boilerplate
- requires fewer language extensions

Compared to à la carte, the tagless approach

- has minimal runtime overhead
- has no need for boilerplate
- requires fewer language extensions

On the other hand, some things are easier with the free monad approach

e.g.

- free monads allow stepping through instruction-by-instruction

Related work

- Stacked `FreeT`.

Related work

- Stacked `FreeT`.
- Alternative (more efficient) implementations of free monads
  e.g. van Laarhooven free monad, “reflection without remorse”

Related work

- Stacked `FreeT`.
- Alternative (more efficient) implementations of free monads
  e.g. van Laarhooven free monad, "reflection without remorse"
- Algebraic effects — avoid a monad stack altogether

  e.g. "Freer monads, more extensible effects"

Related work

- Stacked `FreeT`.
- Alternative (more efficient) implementations of free monads
  e.g. van Laarhooven free monad, "reflection without remorse"
- Algebraic effects — avoid a monad stack altogether

  e.g. "Freer monads, more extensible effects"
- Extensible data types

Related work

- Stacked `FreeT`.
- Alternative (more efficient) implementations of free monads
  e.g. van Laarhooven free monad, “reflection without remorse”
- Algebraic effects — avoid a monad stack altogether

  e.g. "Freer monads, more extensible effects"
- Extensible data types

Related work

- Stacked `FreeT`.
- Alternative (more efficient) implementations of free monads
  e.g. van Laarhooven free monad, "reflection without remorse"
- Algebraic effects — avoid a monad stack altogether

  e.g. "Freer monads, more extensible effects"
- Extensible data types

  see the `compdata` package and "Typed tagless final interpreters".

References

Swierstra, Wouter. “Data types à la carte.” Journal of functional programming 18.04 (2008): 423-436.

Bahr, Patrick, and Hvitved, Tom. “Compositional data types.” Proceedings of the seventh ACM SIGPLAN workshop on Generic programming. ACM, 2011.

Kiselyov, Oleg. “Typed tagless final interpreters.” Generic and Indexed Programming. Springer Berlin Heidelberg, 2012. 130-174.