

Distributed Modal Logic

Matt Parsons

May 12, 2016

Abstract

Logic, type theory, and category theory give three equivalent ways of expressing computation. Logical theorems and axioms can be implemented as type systems for programming languages using the Curry-Howard correspondence, providing a novel means for safer code. This paper explores this connection and uses a category theory interpretation of modal logic to present an implementation of modal logic applied to distributed systems.

1 Introduction

Conal Elliot’s functional reactive programming [2] provides a declarative and denotative basis for expressing programs that vary over time. The naive implementation of FRP is prone to space- and time-leaks, which can be fixed in two ways. The first is through category theory: by using an abstraction called an **Arrow**, Hudak et al [6] solve many of these issues. The second is through logic: Jeffery [7] demonstrates that Linear-time Temporal Logic types FRP well, and the additional rules of LTL provide safeguards against these performance issues.

Foner’s “Getting a Quick Fix on Comonads” [4] examined an interpretation of Löb’s theorem in Haskell and determined that it did not properly satisfy the axioms of $S4$ modal logic. By finding an appropriate Haskell type class that fit the logic more closely, he was able to derive a rather natural and efficient implementation of n -dimensional spreadsheets with relative references. The type class in question is `ComonadApply`, a concept imported from category theory.

Tom Murphy VII’s “Modal types for Mobile Code” presented a novel interpretation of $S5$ modal logic applicable to distributed systems. [10] Murphy interprets $\Box A$ to be a continuation yielding an A that may be run anywhere on the network, and $\Diamond A$ to be an address to a remote value of type A .

Logic and categories both offer compelling resources for software developers to improve their work. Murphy does not mention category theory in his work. Is it possible that a categorical interpretation of his distributed system might provide additional insights or implementation tricks?

2 The Logic

Before diving into the logic presented in Murphy’s paper on Lambda 5 [9], let’s review the basic concepts of modal logic.

2.1 Modal Logic

Modal logic is an augmentation of propositional logic that provides a pair of new operators:

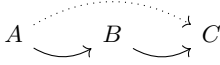


- \Box , signifying ‘always’ or ‘necessarily’
- \Diamond , signifying ‘at some point in the future’ or ‘possibly’

The precise semantics of modal logic are determined by which axioms you take. The common axioms are described here:

- N: Necessity: if A is a theorem, then so is $\Box A$.
- K: Distribution $\Box(A \implies B) \implies (\Box A \implies \Box B)$
- T: Reflexivity $\Box A \implies A$
- 4: $\Box A \implies \Box \Box A$

- B: $A \implies \Box\Diamond A$
- 5: $\Diamond A \implies \Box\Diamond A$

The core of modal logic is a tower of these axioms, each added one at a time: K is the combination of N and K axioms, T adds reflexivity, S4 adds the 4 axiom, and S5 adds either the 5 axiom or the B axiom.

Transitivity	Reflexivity	Symmetry
$\Box A \implies \Box\Box A$	$\Box A \implies A$	$\Diamond A \implies \Box\Diamond A$
		

These axioms speak to the specific characteristics of the accessibility relationship. It's easiest to discuss these in terms of the 'possible worlds' model of modal logic, where A means that A is true in this world, $\Box A$ means that A is necessarily true in all worlds, and $\Diamond A$ means that A is true in some possible world. The three major traits of the accessibility relationship are transitivity (if world A can access world B, and world B can access world C, then A can access C), reflexivity (world A can access world A), and symmetry (if world A can access world B, then world B can access world A). As an aside, the reflexivity and transitivity properties are precisely what is necessary to form a category.

2.2 Propositions as Types

The Curry-Howard correspondence tells us that a logical proposition is equivalent to a type signature in a programming language, a logical proof is equivalent to an expression fitting the type of the sentence, and the evaluation of programs is equivalent to the simplification of proofs. [13] Therefore, the type systems for our programming languages serve as systems to make logical statements about our programs. The more interesting things we can say in our types, the more useful they are in verifying the correctness of our programs. By having these interesting things correspond to a well-behaved formal logic, then we constrain ourselves to making sense. Compile-time verification is particularly appealing: we can't even run the code if it doesn't make logical sense!

2.3 Lambda 5

Let's evaluate the logical system Lambda 5 as defined by Murphy. [9] It's designed to be used as a type theory for distributed computation. The "many worlds" in this logic are the distinct computers running in a distributed network. Murphy's logic interprets $\Box A$ to mean "Mobile code of type A that can be executed on any computer," and $\Diamond A$ to mean "an address of a remote value of type A ."

Because the relationship between computers on a network is reflexive, transitive, and symmetric, the S5 logic is the basis for this system. Furthermore, because constructive proofs are more useful for type theories, the intuitionistic variant *IS5* is used. Intuitionistic logic dispenses with double negation and the law of excluded middle. In this system, the two operators can no longer be expressed in terms of each other. This makes the B and 5 axioms tricky. Since it's generally not accepted to do $A \implies \Box A$, and the two axioms are able to generate some boxiness given some diamondness, then they *must* be related somehow.

The specific axioms for *IS5* are given below: [5]

1. $\Box(A \implies B) \implies (\Box A \implies \Box B)$
2. $\Box(A \implies B) \implies (\Diamond A \implies \Diamond B)$
3. $\Diamond \perp \implies \perp$
4. $\Diamond(A \wedge B) \implies (\Diamond A \wedge \Diamond B)$
5. $(\Diamond A \implies \Box B) \implies \Box(A \implies B)$
6. $(\Box A \implies A) \wedge (A \implies \Diamond A)$
7. $(\Diamond \Box A \implies \Box A) \wedge (\Diamond A \implies \Box \Diamond A)$

The operational semantics for the Lambda 5 language involve deterministic sequential machines operating on various worlds, termed w_i . A value can be accessed remotely given a world name w_i and the label for the value.

One of the key examples given is the symmetry axiom, $\Diamond \Box A \implies \Box A$. We can read this as “Given a remote address to a value of mobile code that can run anywhere and produce an A , we can retrieve that code and run it here.” The proof term given in the paper is:

$$\lambda x. \text{letd } w. y = x \text{ in } \text{fetch}[w]y$$

where *letd* is a primitive of the \Diamond elimination rule, which binds a pair of variables w (representing the world that the value originated in) and y (the value at that world) in the expression after *in*. **fetch** is a primitive that takes a value $M : \Box A @ w'$ and runs it in the current world w . The signature $M : \Box A @ w'$ can be read as: M is a continuation that can be executed at any world currently located at w' .

3 Category Theory

Category theory is a branch of abstract mathematics that seeks to provide a unifying meta-language to talk about mathematics. The following sections introduce ideas in category theory that we need in order to provide the link to the modal logic above. This theory will inform our implementation in Haskell.

3.1 Category

A category is an algebraic structure consisting of a collection of objects and morphisms (also known as arrows) between objects. For a first intuition, categories generalize sets, and arrows generalize functions where the domain and range are the same. For \mathcal{C} to be a category, the objects and morphisms must satisfy the following properties:

1. Each object has an identity arrow:

$$\forall a \in \mathcal{C}, \exists f \in \mathcal{C}_{\rightarrow} . f \ a = a$$

$$\begin{array}{c} id \\ \curvearrowright \\ a \end{array}$$

2. Arrows compose associatively

$$\forall a, b, c \in \mathcal{C}. (a \rightarrow b) \rightarrow c = a \rightarrow (b \rightarrow c)$$

$$\begin{array}{ccc} a & \xrightarrow{f \circ g} & c \\ & \searrow f \quad \nearrow g & \\ & b & \end{array}$$

Arrows in category theory correspond with implication in logic and function types in type theory. The notion of objects and arrows permits the drawing of illustrative diagrams. The associativity property means that any diagram is *commutative*: any path you follow from one object to another using the directed arrows yields the same final result. If we have two paths through a diagram, then we can select the shortest path and be confident that our result is the same.

3.2 Functor

Arrows in a category don't quite generalize functions in the same way that categories generalize sets. For a category \mathcal{C} , the arrows in the category correspond to endomorphisms: functions with the type $f : \forall a \in \mathcal{C}. a \rightarrow a$. Functions like $\lambda x. x^2 : \mathbb{R} \rightarrow \mathbb{R}$ or $\text{Succ} : \mathbb{N} \rightarrow \mathbb{N}$ are examples.

Functions that map elements from one set to another, like $\lambda xy. \frac{x}{y} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{Q}$, require a more powerful concept. The categorical equivalent is called a functor. A functor F is a mapping between categories \mathcal{C} and \mathcal{D} . F maps every object and arrow in \mathcal{C} to \mathcal{D} with two laws:

$$\begin{array}{ccc} F(a) \in \mathcal{D} & \xrightarrow{F(f)} & F(b) \\ F \uparrow & & \uparrow F \\ a \in \mathcal{C} & \xrightarrow{f} & b \in \mathcal{C} \end{array}$$

3.2.1 Identity

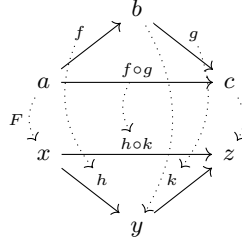
$$\forall a \in \mathcal{C}. id(F a) = F(id a)$$

$$\begin{array}{ccc} a & \xrightarrow{F} & F(a) \\ id \uparrow & \dots F(id) \dots & \uparrow id \\ a & & F(a) \end{array}$$

If a functor maps an object a in \mathcal{C} to b in \mathcal{D} , then the functor must map the identity arrow for a to the identity arrow to b .

3.2.2 Composition

$$\forall f, g : a \rightarrow b \in \mathcal{C}. F(f \circ g) = F(f) \circ F(g)$$



In the above diagram, the functor F maps each object and arrow from the top category \mathcal{C} to the bottom category \mathcal{D} with the dotted lines representing the mappings. The mapping of composition of arrows must be equal to the composition of the mapping of arrows.

Functors give rise to the category of categories Cat , where objects are categories and morphisms are functors between categories.

3.2.3 Correspondence in Type Theory

Functors correspond to type constructors or type functions in type theory, with the added laws of identity and composition. We can gain an intuition on functors by considering examples of generic or parametric types.

In Haskell, functors (specifically, endofunctors on the category *Hask* of Haskell values) are defined as:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Containers form easy functors to get an intuition about. A container can be a functor if you can provide an implementation of `fmap` which transforms each `a` in the container to an element of type `b`. Many non-container types are functors, too, like functions with a fixed input value:

```
instance Functor ((->) input) where
    fmap :: (a -> b) -> (input -> a) -> (input -> b)
    fmap f g = \x -> f (g x)
```

as well as continuations:

```
newtype Cont r a = Cont { runCont :: (a -> r) -> r }
```

```
instance Functor (Cont r) where
    fmap f c = Cont (\k -> runCont c (k . f))
```

and stateful computation:

```
newtype State s a = State { runState :: s -> (a, s) }
```

```
instance Functor (State s) where
    fmap f sa = State (\s -> let (a, s') = runState sa s
                             in (f a, s'))
```

The `fmap` operation is about generalizing a function that works on ordinary values to work on a whole `Functorful` of ordinary values.

3.3 An Entirely Natural Transformation

A natural transformation is a mapping η between two functors F and G on two categories \mathcal{C} and \mathcal{D} that satisfy the following commutative diagram:

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ \downarrow \eta_X & & \downarrow \eta_Y \\ G(X) & \xrightarrow{G(f)} & G(Y) \end{array}$$

A natural transformation is a way to convert one functor into another. Similar to how categories+functors form a category, natural transformations and functors between two categories also form a category. The objects in the category are the functors, and the morphisms are the natural transformations.

3.3.1 Code Examples

The essence of a natural transformation is that you shouldn't use any of the information from the objects in the underlying category when mapping the functors. In Haskell, this can be expressed in terms of a rank 2 type:

```
type Nat f g = forall x. f x -> g x
```

The use of the rank 2 type constrains the function from being able to view or inspect the values inside the functors.

A function to convert a `[a]` into a `Vector a` is an easy natural transformation, as is the inverse. There's also a natural transformation `[a]` to `Map Int a`, where the integer keys are the indexes in the list.

A natural transformation is not required to preserve all information. `Maybe` has a natural transformation to `List`:

```
maybeToList :: Nat Maybe []
maybeToList (Just x) = [x]
maybeToList Nothing  = []
```

Note that the type signature does not mention the type of the value in the container. We can expand the type signature to read as:

```
maybeToList :: forall a. [a] -> Maybe a
```

We're asserting that this function must work for all types `a`, and that we can't use any information about the `a`s inside in order to implement the function. The reversal of this natural transformation is necessarily lossy:

```
listToMaybe :: Nat [] Maybe
listToMaybe (x:_) = Just x
listToMaybe []    = Nothing
```

3.4 Monad

A monad is an endofunctor (a mapping from \mathcal{C} to \mathcal{C}) that is equipped with two natural transformations:

1. η , or `return`, taking any object from the identity functor $I(\mathcal{C})$ to $F(\mathcal{C})$.
2. μ , or `join`, taking objects from $F(F(\mathcal{C}))$ to $F(\mathcal{C})$.

Monads follow the monoid laws of associativity and identity where the unit is η and \oplus is μ . The identity functor is an easy monad. The natural transformation η is the identity natural transformation. For μ , we can show (by commutativity of the diagram) that taking identity twice is equivalent to taking identity once:

$$\begin{array}{ccccc} & & Id & & \\ & \curvearrowright & & \curvearrowleft & \\ \mathcal{C} & \xrightarrow{Id} & \mathcal{C} & \xrightarrow{Id} & \mathcal{C} \end{array}$$

3.4.1 More Code Examples

Monads can be represented as a type class in Haskell, with the following definition:

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

  ma >>= f = join (fmap f ma)
  join mma = mma >>= \ma -> ma
```

We can provide a default implementation of bind (written here with the infix operator `>>=`) and join written in terms of each other. Lists and optionals form monads.

```
instance Monad [] where
  return a = [a]
  join (xs:xss) = xs ++ join xss
```

The list monad is used for nondeterministic programming and is the basis for list comprehensions in Haskell.

```
instance Monad Maybe where
  return a = Just a
  Just a >>= f = f a
  Nothing >>= _ = Nothing
```

The Maybe monad is useful when sequencing actions that might fail, like looking up items in a dictionary.

A monad is a powerful abstraction for sequencing computational effects. Given a value in a monadic structure `m`, we can provide a function that operates on that value and yields more monadic structure. `bind` handles the boilerplate of flattening it back out. Monads are a convenient way to express state, asynchronous computation, and computation that may error.

The use of monads in computer science was initially discovered by Eugenio Moggi [8] and then popularized by Wadler [12] and others for use in functional programming.

3.5 Comonad

A comonad is the categorical dual of a monad. In category theory, all ideas have a dual. The dual of a concept is given by taking the object's diagram reversing all of the arrows. We can arrive at something similar by reversing the arrows of types in type signatures. Flipping the `Monad` function's arrows gives us:

```
return    :: a -> m a
coreturn  :: a <- m a
extract   :: m e -> e

join      :: m (m a) -> m a
cojoin    :: m (m a) <- m a
duplicate :: m e -> m (m e)

bind      :: m a -> (a -> m b) -> m b
cobind    :: m a <- (a <- m b) <- m b
extend    :: m q -> (m q -> e) -> m e
```

Where a monad gives us the ability to take a single value in a structure and generate new structure, a comonad gives us the ability to take a structure of values and yield a single new value. The canonical comonad is the infinite stream data type:

```
data Stream a = a :< Stream a

instance Functor Stream where
  fmap f (a :< sa) = f a :< fmap f sa

instance Comonad Stream where
```

```

extract (a :< _) = a
duplicate s@(a :< as) = s :< duplicate as
extend f = fmap f . duplicate

```

To duplicate a stream, we set the value of index i_n to be the entire stream, and the position at i_{n+1} to be the stream starting from i_{n+1} .

3.6 Adjunctions

An adjunction is a pair of functors which generalize the notion of a Galois connection. From the category \mathcal{C} , the functor R *lifts* an object to the category \mathcal{D} by providing some free structure. The functor L then *loses* some information about an object in \mathcal{D} to map the object back to \mathcal{C} . This isn't necessarily an isomorphism, as forgetting composed with lifting $R \circ L$ loses some information and can't map the object back. However, lifting followed by forgetting $L \circ R$ is equivalent to the Id functor, and $R \circ L \circ R$ is equivalent to R .

In this case, we write $L \dashv R$ to say that L is left adjoint to R and that R is right adjoint to L . An interesting property of an adjunction is that every adjunction gives rise to both a monad through $R \circ L$ and a comonad through $L \circ R$.

3.6.1 A Common Adjunction: Pair and Function

The Haskell tuple (r, a) (also known as $Prod_r a$) and function $r \rightarrow a$ (also known as $Exp_r a$) are both functors, and it is the case that $Prod \dashv Exp$.

```

instance Functor ((,) r) where
    fmap f (r, a) = (r, f a)

instance Functor ((->) r) where
    fmap f k = f . k

class (Functor g, Functor f) => Adjunction f g | f -> g, g -> f where
    unit    :: a -> g (f a)
    counit  :: f (g a) -> a
    left    :: (f a -> b) -> a -> g b
    right   :: (a -> g b) -> f a -> b

instance Adjunction ((,) r) ((->) r) where
    unit a      = \r -> (r, a)
    counit (r, f) = f r

```

We can specialize the type of `unit` to this specific adjunction, which illustrates the monad/comonad:

```

unit    :: a -> r -> (r, a)
counit  :: (r, r -> a) -> a

```

Indeed, we can write a generic instance of the `Monad` type class in Haskell that works with any composition of two functors which form an adjunction. If we flip the composition, then we can write a comonad instance as well. This is witnessed below:

```

newtype Compose f g a = Compose { decompose :: f (g a) }

instance (Adjunction f g, Applicative f, Applicative g) => Monad (Compose g f) where
    return = Compose . unit
    m >>= f = Compose . fmap (right (decompose . f)) . decompose $ m

instance (Adjunction f g) => Comonad (Compose f g) where
    extract = counit . decompose
    extend f = Compose . fmap (left (f . Compose)) . decompose

```

If this is the case, then `Compose ((,) r) ((->) r)` should be a comonad, and `Compose ((->) r) ((,) r)` should be a monad. If we expand the types, we'll see that:

```

Compose ((->) r) ((,) r)
  = (->) r ((,) r a)
  = r -> (r, a)

```

```

Compose ((,) r) ((->) r)
  = (,) r ((->) r a)
  = (r, r -> a)

```

Indeed, the monad formed by the adjunction of the two functors is the **State** monad! The comonad is known as the **Store** comonad, and is less common in practice.

4 Link to Modal Logic

Now that we’ve gathered all of our toys from logic and category theory, it’s time to figure out how they’re related. The type signatures of the functions in the **Monad** and **Comonad** classes look awfully familiar to the axioms for the \Diamond and \Box constructs. Let’s see if there’s more to that.

4.1 Diamonad

The \Diamond operator in modal logic looks like it corresponds to the monad in category theory. Let’s consider the natural transformations $\eta : I(A) \rightarrow M(A)$ and $\mu : M(M(A)) \rightarrow M(A)$. Can we prove these?

η is simple enough. It corresponds with the \Diamond introduction rule $A \vdash \Diamond A$: “If A is true, then A is possibly true.” μ , likewise, is the $\Diamond 4$ axiom.

We can additionally prove the type of `bind :: Monad m => m a -> (a -> m b) -> m b` by taking advantage of the distribution of \Diamond over implication.

$$\frac{\frac{\frac{A \implies \Diamond B}{\Diamond(A \implies \Diamond B)} \Diamond_I}{\frac{\Diamond A \implies \Diamond \Diamond B}{\Diamond \Diamond B} \text{Distribution}}{\frac{\Diamond \Diamond B}{\Diamond B} \mu}$$

4.2 Comonad

The fundamental operations for a comonad are duplicate ($\Box A \implies \Box \Box A$), which is precisely the axiom that gives rise to $S4$ modal logic, and extract ($\Box A \implies A$), which is the \Box elimination rule. Comonads, as functors, also implement `fmap`, and the interaction of `fmap`, `duplicate`, and `extract` must follow the following laws:

$$extract \circ duplicate = id \tag{1}$$

$$fmap \ extract \circ duplicate = id \tag{2}$$

$$duplicate \circ duplicate = fmap \ duplicate \circ duplicate \tag{3}$$

In $S4$ modal logic, a plain comonad is close to the \Box operator, but we don’t have $\Box(a \implies b) \implies (\Box a \implies \Box b)$. The Haskell type class **ComonadApply** equips a **Comonad** with exactly that function, allowing it to satisfy the $S4$ axioms.

As Foner observed [4], all comonads in Haskell have a property known as *functorial strength*. Given some `f a` and a `b`, you can `strong b = fmap (\a -> (a, b))` to lift the `b` value into the **Functor**. This corresponds to a proposition of the form $B \implies \Box A \implies \Box(A \wedge B)$, which is impermissible in the logic. We’ll need to take special care when writing our implementation to avoid this.

4.3 To IS5 and Beyond

The modal logic used in the Lambda 5 system uses $IS5$ as a basis, not $S4$. Extending $S4$ to $S5$ adds the following axiom:

$$\Diamond A \implies \Box \Diamond A$$

This additional axiom expresses the idea that, if A is possible, then the possibility of A is necessary. This makes a category theory interpretation somewhat tricky. One of the necessary features of both \Box and a comonad is that we don't have $A \implies \Box A$ or $f : \forall w. \text{Comonad } w \implies a \rightarrow w a$. In this case, we're relying on some structure of the monad that permits us to push comonadic structure beneath.

In order to extend this interpretation to $S5$, we can't take just any old monad and comonad for our interpretation. We'll need to delve deeper.

4.4 Double the Adjunction, Double the Functor

Suppose we have two categories \mathcal{C} and \mathcal{D} with a triple of functors L, U, R , where $L \dashv U$ and $U \dashv R$, and (as a natural consequence of the composition of adjunctions) $L \dashv U \dashv R$.

$$\begin{array}{ccc} & \xrightarrow{R} & \\ \mathcal{C} & \xleftarrow{U} & \mathcal{D} \\ & \xleftarrow{L} & \end{array}$$

This gives rise to a monad $M = U \circ L$ and a comonad $W = U \circ R$ on \mathcal{C} . Since adjunctions compose, the combination $M \dashv W$ is an adjunction. By assigning M to \Diamond and W to \Box , we've arrived at the axiom we require: $\Diamond \dashv \Box$. [1] Asserting that in Haskell gives us `Adjunction dia box` with a function `unit :: a -> box (dia a)`, which satisfies our requirement.

Our task is to now find the appropriate functors and categories to provide a suitable implementation of Lambda 5.

5 Implementing in Haskell

The theory has presented us with a neat implementation plan. We can arrive at an $S5$ modal logic categorically. For the complete implementation, see the `Logic.Modal Haskell` module.

5.1 Defining the Interface

First, we'll define a type class that represents the axioms of $S4$ modal logic.

```
class CModalS4 box where
  axiom4 :: box p -> box (box p)
  axiomT :: box p -> p
  axiomK :: box (p -> q) -> box p -> box q
```

We can demonstrate that this is exactly equivalent to the `ComonadApply` type class by providing an implementation solely in terms of that type class. To avoid requiring an `UndecidableInstance`, a `newtype` wrapper will be used to disambiguate the context. The only point of the newtype wrapper is to serve as a witness that the type `w` is an instance of `ComonadApply`.

```
newtype S4Witness w a = Valhalla { witnessMe :: w a }

instance ComonadApply w => CModalS4 (S4Witness w) where
  axiom4 = Valhalla . fmap Valhalla . duplicate . witnessMe
  axiomT = extract . witnessMe
  axiomK (Valhalla f) (Valhalla a) = Valhalla (f <@> a)
```

Since this establishes that they're equivalent, we can relegate `ModalS4` as a type alias. Equipped with this, the definition for `ModalS5` is the addition of the B axiom in type signature form:

```
type ModalS4 = ComonadApply

class (Monad dia, ModalS4 box) => ModalS5 box dia where
  axiomB :: a -> box (dia a)
```

As is customary when doing fancy things with Haskell, we'll provide some infix type operator aliases to pretend that we're real mathematicians. These will help clarify the instance contexts when defining the generic `Monad` and `Comonad` instance

```

type f :: g = Compose f g
type f -| g = Adjunction f g

instance (f -| g, Applicative f, Applicative g) => Monad (g :: f) where
    return = Compose . unit
    m >>= f = Compose . fmap (rightAdjunct (getCompose . f)) . getCompose $ m

instance (f -| g) => Comonad (f :: g) where
    extract = counit . getCompose
    extend f = Compose . fmap (leftAdjunct (f . Compose)) . getCompose

```

The `Applicative` instances are required, as all `Monads` must be instances of the `Applicative` type class in recent versions of Haskell (and morally in all versions of Haskell). And now, for the final bit of fun:

```

instance (g -| u, u -| f, ModalS4 (u :: f), Applicative u, Applicative g)
    => ModalS5 (u :: f) (u :: g) where
    axiomB = unit

```

Provided that `g` is left adjoint to `u`, `u` is left adjoint to `f`, `u` and `g` are both `Applicative` functors, and finally that the composition of `u` and `f` satisfy the `ModalS4` axioms, then the compositions of `u`, `f`, and `g` satisfy the `ModalS5` axioms automatically. Now that we've defined the logical type class, all that's left is to provide the functors `u`, `f`, and `g` that satisfy the required constraints.

5.2 Mechanical Satisfaction

In keeping with the spirit of programming in a lazy language, I'll lazily define the necessary functors as empty type constructors, and have the compiler assert the various requirements we need. We'll fill in only the bare minimum to get it to compile, punting any real design decisions until later.

```

data U a
data F a
data G a

type Box = U :: F
type Dia = U :: G

pls :: ModalS5 Box Dia => ()
pls = ()

```

By repeatedly requesting the `:type` of `pls` in `GHCi`, this causes the solver to attempt to find instances and resolve them. When it fails, it reports the failure, and we can fill in a pretend implementation. The list it provides is:

- `Applicative G`
- `Applicative U`
- `ComonadApply (U :: F)`
- `Adjunction U F`
- `Functor F`
- `Representable F`
- `Distributive F`
- `Adjunction G U`
- `Representable U`
- `Distributive U`

which we can laughingly provide a dummy implementation as:

```

data U a
    deriving (Functor)

instance Applicative U where
    pure = undefined
    (<*>) = undefined

```

```

instance ComonadApply (U :: F) where
  (<@>) = undefined

instance Adjunction U F where
  unit = undefined
  counit = undefined

instance Adjunction G U where
  unit = undefined
  counit = undefined

instance Representable U where
  type Rep U = U Int
  tabulate = undefined
  index = undefined

instance Distributive U where

data F a
  deriving (Functor)

instance Representable F where
  type Rep F = F Int
  tabulate = undefined
  index = undefined

instance Distributive F where

data G a
  deriving (Functor)

instance Applicative G where
  pure = undefined
  (<*>) = undefined

```

Now that we've mechanically arrived at our constraints, it is time to consider the actual semantics of these types. The semantics of the types will provide the implementation of the data constructors, which will permit implementations of the type classes, which will provide an implementation of the modal logic we require.

5.3 G, F, U and Meaning

5.3.1 Remote Diamond

Lambda 5's $\Diamond A$ represents a remote address of a value of type A . The \Diamond is comparatively conceptually simple to implement. Let's inspect a snippet of code to see how it might work:

```

remoteValue :: Dia Int
remoteValue = ...

someFoo :: Dia Bool
someFoo = do
  value <- remoteValue
  return (10 < value)

```

Supposing we have some `remoteValue` representing an `Int` available at some location, we can bind the `Int` out of that, and do operations on it in the `Dia` monad. Alternatively, we should also be able to attempt to `fetch` the remote value. Fetching will naturally have some concept of failure, and will need to take place in `IO` in order to access network resources. The natural type of `fetch`, then, is:

```
fetch :: Dia a -> ExceptT NetworkError IO a
```

Which, in turn, provides some information as to what the functors for `Dia` might be. `ExceptT NetworkError IO a` is a monad transformer stack representing an `IO` (`Either NetworkError a`). We might be tempted, then, to declare `U` as `IO` and `G` as `Either NetworkError`. There's an issue with that: The `U` functor is shared with the `Comonad`, and `IO` is certainly not a comonad of any sort. This implementation then fails to satisfy our logic. `U` remains elusive, though `G` could very well be `ExceptT NetworkError IO a`.

5.3.2 Boxing it up

Lambda 5's $\Box A$ represents a continuation yielding a value of type A that can be run anywhere on the network. We might be tempted to note that the definition of \Box sounds like a pure computation with no dependencies, equivalent to a lambda expression with no free variables. If we take that road, then \Box is simply the type of pure computations, or the `Identity` comonad. We could dispense with the `Identity`, leaving us an interpretation of \Box that was simply ordinary Haskell values. Wherever we have a comonad on \mathcal{C} , we also have a monad with the reversed morphisms, so we can categorically *shift* the monads up a stack. We'd require a pair of monads M_1 and M_2 , then, one to represent ordinary expressions in modal logic, and another to represent \Diamond .

Unfortunately, this fails to work in Haskell for one reason: free variables! Consider the following snippet:

```
foo :: Int -> Int -> Int
foo x y = x + y * bar

bar :: Int
bar = 6
```

If we tried to `box` up `foo` and send it across the network, it'd blow up at runtime, requesting access to the `bar` term. This demonstrates that the \Box must be a sealed, complete \Box : no free variables. The “effect” of plain Haskell values then is access to free variables in the local machine state. In order to provide computations that may be run remotely, we must eliminate these free variables. The composition $U \circ F$ must provide some means of packaging values and closures and making them available.

For inspiration on our next step, let's consider the introduction rule for \Box :

$$A \vdash \Box A$$

which we may read in plain English as “If it is provable that, with no assumptions, A is true, then $\Box A$ is true.” By substituting provability for runnability and assumptions with dynamic values, we arrive at “If A is runnable with no input at run-time, then $\Box A$ is runnable.” The model that we're seeking is akin to the `constexpr` or `const` from C++ template metaprogramming. Our \Box will be built at compile-time for our run-time programs to be able to use, unpack, and send about the network.

This insight fueled the development of Cloud Haskell. [3] Cloud Haskell features a Template Haskell directive to package a closure up at compile time as long as all of the values are instances of a `Serializable` type class. The `distributed-static` library [11] makes these functions available.

5.4 Head in the Clouds

The Cloud Haskell paper describes a `Static` type that corresponds with values known at compile time or top-level definitions, along with a `Closure` type that represents serialized function closures. The paper describes a primitive, `static e`, which (at compile time) takes an `e :: a` and makes a `Static a` out of it, and another primitive `unstatic :: Static a -> a`. All terms described in the `e` to be `staticed` must be top level definitions or constants, permitting the compiler to evaluate them at compile time, serialize them, and apply the `Static` constructor. If we can define either `duplicate :: Static a -> Static (Static a)` or `extend :: Static a -> (Static a -> b) -> Static b`, then we've got a `Comonad` instance. This instance satisfies the requirements to form the comonad we require in order to implement \Box for our logic.

```
instance Comonad Static where
    extract = unstatic
    duplicate s = static s
```

```
instance ComonadApply Static where
  (<@>) = staticApply
```

We’re safe to use `static` on the `s :: Static a` as the type `Static a` is a proof that `s` is entirely serialized or a top-level definition. Given an appropriate implementation of Cloud Haskell in a forthcoming version of GHC, we’ve successfully provided the \square for implementing Lambda 5.

Cloud Haskell also presents a `Process` monad, which is used to execute code on remote hosts. The abstraction is Erlang-style message passing, so Cloud Haskell does not support `fetch`-style requests by default. The provided primitives in this abstraction are:

- `spawn`, a function that launches a closure on a provided `NodeId`, returning the new `ProcessId`
- `send`, a function that sends a message to a `ProcessId`
- `expect`, a function that blocks the process awaiting a message.

We can implement `fetch` on top of this paradigm:

```
newtype Fetch k a = Fetch k
  deriving (Binary)

newtype Response k a = Response (Maybe a)
  deriving (Binary)

fetch
  :: forall k a. (Serializable k, Serializable a)
  => k
  -> ProcessId
  -> Process (Response k a)

fetch k pid = do
  self <- getSelfPid
  send pid (self, Fetch k :: Fetch k a)
  expect

fetchListen
  :: forall k a. (Serializable k, Serializable a, Ord k)
  => IORef (Map k a)
  -> Process ()

fetchListen ref = forever $ do
  (pid, Fetch k) <- expect :: Process (ProcessId, Fetch k a)
  table <- liftIO $ readIORef ref
  send pid (Response (Map.lookup k table) :: Response k a)
```

`fetch` takes a `k` and a `ProcessId` and sends the lookup request and the current process ID to the target. The `fetchListen` function has a reference to a dictionary of `k` to `a`, awaiting `Fetch` requests, and responding with the value contained therein or `Nothing`.

6 Conclusion

The `static` and `unstatic` primitives require an extension to the Haskell compiler and runtime to implement, and that work has yet to be completed. The library that implements these concepts without compiler extensions [11] has a different model of execution. Since the compiler isn’t able to enforce the correctness of the static values and lookup table, the type of `unstatic` is:

```
unstatic :: Typeable a => RemoteTable -> Static a -> Either String a
```

The `RemoteTable` is a runtime dictionary that maps serialized static labels to their actual values. This function accepts a `RemoteTable` as input and looks up the `Static a` value in the table. The value is stored as a `Dynamic` value, and if the type safe `fromDynamic :: Typeable a => Dynamic -> Maybe a` cast succeeds, then the function returns the `Right` result. If the lookup or cast failed, then the function returns the `Left` error message.

This paper demonstrates that a categorical interpretation of the logical system presented by Murphy corresponds to the theoretical implementation of Cloud Haskell. As of right now, an implementation of Murphy’s Lambda 5 is not possible with the current version of GHC and the `distributed-{process,static}` libraries.

References

- [1] Awodey, S. 2006. *Category theory*. The Clarendon Press Oxford University Press.
- [2] Elliott, C. 1999. From functional animation to sprite-based display. *Practical aspects of declarative languages* (1999).
- [3] Epstein, J. et al. 2011. Towards haskell in the cloud. *SIGPLAN Not.* 46, 12 (Sep. 2011), 118–129.
- [4] Foner, K. 2015. Functional pearl: Getting a quick fix on comonads. *Proceedings of the 2015 ACM SIGPLAN symposium on haskell* (New York, NY, USA, 2015), 106–117.
- [5] Galmiche, D. and Salhi, Y. 2010. Logic for programming, artificial intelligence, and reasoning: 16th international conference, IPAR-16, dakar, senegal, april 25–May 1, 2010, revised selected papers. E.M. Clarke and A. Voronkov, eds. Springer Berlin Heidelberg. 255–271.
- [6] Hudak, P. et al. 2003. Arrows, robots, and functional reactive programming. *Advanced functional programming*. J. Jeuring and S. Jones, eds. Springer Berlin Heidelberg. 159–187.
- [7] Jeffrey, A. 2012. LTL types fRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. *Proceedings of the sixth workshop on programming languages meets program verification* (New York, NY, USA, 2012), 49–60.
- [8] Moggi, E. 1991. Notions of computation and monads. *Information and Computation*. 93, 1 (1991), 55–92.
- [9] Murphy, T. et al. 2004. A symmetric modal lambda calculus for distributed computing. *Logic in computer science, 2004. proceedings of the 19th annual IEEE symposium on* (2004), 286–295.
- [10] Murphy, T.V. 2008. *Modal types for mobile code*. ProQuest.
- [11] Vries, E. de 2016. distributed-static. <https://hackage.haskell.org/package/distributed-static>.
- [12] Wadler, P. 1995. Monads for functional programming. *Advanced functional programming, first international spring school on advanced functional programming techniques-tutorial text* (London, UK, 1995), 24–52.
- [13] Wadler, P. 2015. Propositions as types. *Communications of the ACM*. 58, 12 (2015), 75–84.