

Distributed Modal Logic

Matt Parsons

May 1, 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 [5] solve many of these issues. The second is through logic: Jeffery [6] 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” [3] examined an interpretation of Löb’s theorem 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 Distributed Modal 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 Plain Modal Logic

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

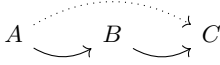


- \Box , signifying ‘always’ or ‘necessarily’
- \Diamond , signifying ‘at some point in the future’ or ‘possibly’

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 A, 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 sentence 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. [12] 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. 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: [4]

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 part 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** takes a value $M : \Box A @ w'$ and runs it in the current world w .

The abstract machine that performs these operations is based on continuations.

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 (or arrows) between objects. Categories generalize sets, and arrows generalize functions where the domain and range are equal.

For \mathcal{C} to be a category, the objects and morphisms must satisfy the following properties:

1. Each object has an identity arrow:

$$\forall a \in \mathcal{C}, \exists f \in \mathcal{C} \rightarrow a \text{ such that } f \circ a = a$$

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

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

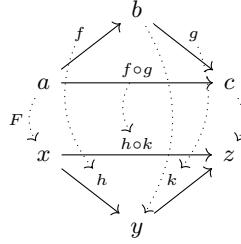
$$\forall a \in \mathcal{C}. id(F\ a) = F(id\ a)$$

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

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'))
```

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.

3.3.1 Code Examples

In Haskell, a natural transformation can be expressed using 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 List
maybeToList (Just x) = [x]
maybeToList Nothing = []
```

The reversal of this natural transformation is necessarily lossy:

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

3.4 Monad

A monad is a functor mapping a category to itself 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 .

3.4.1 Examples

We can imagine a functor as “wrapping” a value. `return` is a way to wrap an unwrapped value. `join` is a way of combining two layers of wrapping into a single wrapping.

Lists, optionals, and functions all form monads. Below, a type signature and implementation for `return` and `join` are provided for several instances:

```
return :: a -> [a]
return a = [a]

return :: a -> Maybe a
return a = Just a

return :: a -> r -> a
return a = \r -> a

join :: [[a]] -> [a]
join (xs : xss) = xs ++ join xss

join :: Maybe (Maybe a) -> Maybe a
```

```

join (Just (Just a)) = Just a
join (Just Nothing)  = Nothing
join Nothing         = Nothing

join :: (r -> r -> a) -> (r -> a)
join f = \r -> f r r

```

bind, also known as flatMap or concatMap, is join . fmap f. The type signature of bind is:

```

bind :: forall m a b. Monad m => m a -> (a -> m b) -> m b
bind ma f = join mapped
  where
    mapped :: m (m b)
    mapped = fmap f ma

```

First, we map over the monadic structure using the Functorial power. Then, the monadic contexts are collapsed using join.

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 [11] 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   :: w e -> e

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

bind      :: m a -> (a -> m b) -> m b
cobind    :: m a <- (a <- m b) <- m b
extend    :: w q -> (w q -> e) -> w 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. This is witnessed below:

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

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

4 Link to Modal Logic

4.1 Diamonad

The \Diamond operator in modal logic 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` $:: \text{Monad } m \Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b$ by taking advantage of the distribution of \Diamond over implication.

$$\begin{array}{c}
\frac{A \implies \Diamond B}{\Diamond(A \implies \Diamond B)} \Diamond_I \\
\frac{\Diamond A \quad \frac{\Diamond(A \implies \Diamond B)}{\Diamond A \implies \Diamond \Diamond B} \text{Distribution}}{\frac{\Diamond \Diamond B}{\Diamond B} \mu}
\end{array}$$

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 precisely the \Box elimination rule. Comonads, as functors, also implement `fmap`, and the interaction of `fmap`, `duplicate`, and `extract` must follow the following laws:

$$\text{extract} \circ \text{duplicate} = \text{id} \quad (1)$$

$$\text{fmap} \text{ extract} \circ \text{duplicate} = \text{id} \quad (2)$$

$$\text{duplicate} \circ \text{duplicate} = \text{fmap duplicate} \circ \text{duplicate} \quad (3)$$

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. The modal logic used in the Lambda 5 system uses *S5* 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 must become more specific with our functors.

4.3 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]

What is the intuition for this trick? R and L are both functors that map \mathcal{C} to \mathcal{D} , and U is the functor that brings those objects back home to \mathcal{C} . By $L \dashv U$, we have that L *lifts* items into \mathcal{C} , and U *forgets* something about the objects when mapping them back to \mathcal{C} .

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. First, we'll define a type class that represents the axioms of *S5* modal logic:

```

class Modal dia box where
  axiomK :: box (a -> b) -> box a -> box b
  axiomT :: box a -> a
  axiom5 :: dia a -> box (dia a)

```

Adjunctions (among other related abstractions) are defined in the Haskell package `adjunction` maintained by Edward Kmett. [7] The type class we'

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] 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.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] Kmett, E. 2016. adjunctions: Adjunctions and representable functors. <http://hackage.haskell.org/package/adjunctions-4.3>.
- [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] 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.
- [12] Wadler, P. 2015. Propositions as types. *Communications of the ACM*. 58, 12 (2015), 75–84.