COMPUTATIONAL EFFECTS FROM A MATHEMATICAL PERSPECTIVE

SANAD KADU

Abstract. This report is based on papers by John Powers, Gordon Plotkin, and Martin Hyland on the relevance of universal algebra to computational effects, using the notion of Lawvere theory in a computational context. We explore the two ways to encapsulate computational effects, viz. monads and Lawvere theories, and how they relate to each other, leading to the notion of algebraic effects. Finally, we study the particular case of continuations as a universal effect and its relation to Yoneda embedding.

Contents

1. Introduction 2
1.1. Lawvere theories 3
1.2. Model of a Lawvere Theory 4
1.3. Category of models 4
1.4. Monads 4
1.5. Connection to Haskell 5
2. Universal Algebra and Computational Effects 6
2.1. Algebra for monads 7
3. Lawvere Theories and Monads 7
3.1. From Monads to Lawvere Theories 7
3.2. From Lawvere Theories to Monads 9
4. Advantages of Lawvere Theories 9
5. Encoding Effects 10
6. Lawvere Theories to Algebraic Effects & Handlers 10
7. Case of Continuations 11
Appendix A. Appendix 14
A.1. Category 14
A.2. Opposite Category 14
A.3. Subcategory 14

A.4. Skeleton 15
A.5. Set 15
A.6. Functor 15
A.7. Natural Transformation 15
A.8. Composition of Natural Transformationsn 16
A.9. Adjunctions 16
A.10. Yoneda Lemma 17
Category of Presheaves 17
References 20

1. Introduction

Universal algebra studies the unity between different algebraic structures, such as rings, groups, and modules. It involves the study of descriptions of a particular structure, called theories. A theory (Σ, E) consists of a specification Σ of operations on that structure and laws or equations E that these operations must satisfy. There is a clear distinction between syntax and semantics, where syntax is the operations and equations, whereas semantics is the model (algebra) that could be described in terms of some other foundational structure e.g., a set(X) of form,

$$\forall op \in \Sigma, f : X^n \to X$$

where n is the arity of f.

There is a relationship between universal algebra and computational effects in that one can model notions of computation using algebraic operations and equations. We will be looking at universal algebra from a categorical perspective. There are two categorical formulations of universal algebra viz. Lawvere theories and monads. First, in a category $\mathbf{C}$ with products, an n-ary operation on an object c is defined to be a morphism from $\mathbf{C}^n$ to C from this, we get the concepts of algebra, a homomorphism. The second formulation comes from the insight that free algebras of a variety contain all the information about the variety. From those, we can create a triple in $\mathbf{C}$, i.e. the monad, and define the notion of algebra and homomorphism in $\mathbf{C}$.

1.1. **Lawvere theories.** Consider $\aleph_0$ be the skeleton of a category of finite sets and all functions between them so that it has finite associative co-products. Since $\aleph_0$ is equipped with finite co-products, the opposite category $\aleph_0^{op}$ must have finite products. Given such a structure, one can create a Lawvere theory as a generic small category $\mathcal{L}$ equipped with finite products, an object x and a product preserving identity-on-object functor:

$$I : {\aleph_0}^{op} \to \mathbf{L}$$

By identity on objects, we mean that objects of any Lawvere theory $\mathcal{L}$ are precisely the objects of $\aleph_0$, and all the functions between these objects have corresponding maps in $\mathbf{L}$. Every object in the theory is a finite power x^n of generic object x. Likewise, the morphisms from $x^n \to x$ are n-ary operations possible on x. These maps are referred to as *operations*, where those arising from $\aleph_0$ are called the *basic product operations*.

The notion of interpreting one theory in terms of another requires some morphisms between Lawvere theories, which yields the category of Lawvere theories called **Law**.

1.1.1. *How to create a Lawvere theory?* We start with (Σ, E) and construct a category such that,

- $obj(\mathbf{L}) = \mathbb{N}$
- $arrows(\mathbf{L}) = n \to m$ are m-length tuples of terms in n variables.
- The projection morphism of finite products that takes in the ith component of a product and gives the term for the ith variable.
- Composition is given by substitution of terms.
- The inclusion functor $I : {\aleph_0}^{op} \to \mathbf{L}$ picks out the projection morphism in $\mathbf{L}$.

1.1.2. *Multisorted Lawvere Theories.* A multi sorted lawvere theory over the set S of sorts is $\mathbf{L}_{msort}$ a small category with products and a function $\phi : S \to \mathbf{L}_{msort}$ such that the product preserving functor is subjective $\Pi : (\mathbf{Set}/\mathbf{S}) \to \mathbf{L}_{msort}$ i.e an operation of arity $x_1, x_2 \cdots x_n \to y$ in $\mathbf{L}_{msort}$ is a morphism of for $\Pi(n, x) \to \phi(y)$ in $\mathbf{L}_{msort}$.

1.2. **Model of a Lawvere Theory.** A model of a Lawvere theory $\mathbf{L}$ in any category $\mathbf{C}$ with finite products is a finite-product preserving functor

$$M : \mathbf{L} \to \mathbf{C}$$

$$M(a \times b) \cong Ma \times Mb$$

Notice here that we only require the preservation of products up to an isomorphism instead of strict preservation. This choice is crucial as it allows one to easily change the base category along with a finite product preserving functor $H : \mathbf{C} \to \mathbf{C}$'.

For example, consider the Lawvere theory of groups $\mathbf{G}$, which is a category with products and a generic group object x inside. A model for $\mathbf{G}$ in some category $\mathbf{C}$ is a product preserving functor $\mathbf{G} \to \mathbf{C}$.

1.3. **Category of models.** Consider a morphism in a Lawvere theory $\mathbf{L}$ over $\mathbf{C}$, of form $m \to n$, which is mapped to some function of form $a^m \to a^n$ in $\mathbf{C}$. If we have two different models M and N, there exists a natural transformation between them that corresponds to a family of functions indexed by n:

$$\mu_n : Mn \to Nn = a^n \to b^n \text{ where } b = N1$$

According to the naturality condition, we can see that n-ary operations or finite products are preserved,

$$Nf \circ \mu_n = \mu_1 \circ Mf \text{ where } f : n \to 1 \text{ is an } n\text{-ary operation in } \mathbf{L}$$

These models are also from a category $\mathbf{Mod}_L(C)$ with natural transformations as morphisms.

1.4. **Monads.** Monads are another category-theoretic formulation of universal algebra. They arise from adjoint pair of functors. A monad on some category $\mathbf{C}$ of types and functions, consists of an endofunctor $T : \mathbf{C} \to \mathbf{C}$ together with two natural transformation:

$$\eta : 1_{\mathbf{C}} \to T \text{ where } 1_{\mathbf{C}} \text{ is the identity functor on } \mathbf{C}$$

$$\mu : T \circ T \to T$$

such that,

(1)
$$\mu \circ T\mu = \mu \circ \mu T$$

$$\begin{tikzcd}
T(T(T(X))) \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T(T(X)) \arrow[d, "\mu"] \\
T(T(X)) \arrow[r, "\mu"'] & T
\end{tikzcd}$$

(2)
$$\mu \circ T\eta = \mu \circ \eta T = 1_{\mathbf{C}}\ T$$

where $1_{\mathbf{C}}$ is the identity morphism from $T \to T$.

$$\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[d, "T\eta"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu"] \\
T(T(X)) \arrow[r, "\mu"'] & T
\end{tikzcd}$$

A *computation* is a process that can do certain operations on data and may eventually finish and return a value. We encapsulate this notion as a triple, (T, η, μ) where if we have values of type X then $T(X)$ is the *computations* of values of type X and $T(T(X))$ are computations of computations of type X and so on. The natural transformation, η_X turns a value into a computation that just returns this value, whereas μ_X turns a computation of computations into their sequence.

1.5. **Connection to Haskell.** In computer science, the notion of monads arises from the imperative that in the category with objects as types and morphisms as programs between types, programs should form a category. We can interpret $\mu : T^2 \to T$ in Haskell as,

```
join :: (Monad m) => m (m a) -> m a
join x = x >>= id
```

Likewise, for $\eta : 1_{\mathbf{C}} \to T$, since, $1_{\mathbf{C}}(a) = a$,

```
return :: a -> m a
```

The programmer implementing these is responsible for ensuring the above criteria are satisfied.

1.5.1. *Monad Laws.* The first Law from category theory,

$$\mu_x T\mu_x = \mu_x \mu_{Tx}$$

corresponds to,

```
join. (fmap join) = join. join. (fmap id)
```

which says that joining after mapping with join is the same as mapping with id and joining twice. Likewise, for the second Law,

$$\mu_x \eta_{Tx} = \mu_x T_{\eta_x} = id$$

we have,

```
join. return = join. fmap return = id
```

2. Universal Algebra and Computational Effects

Universal algebra and computational effects are intimately related in that many of the computational effects emerge from computational operations on particular effects, e.g., raise for exceptions, lookup and update for side effects, read and write for input/output, etc.

Let T denote a computational effect then,

- A *T-program* is a function $f : A \to \mathbb{T}B$ where A is the set of values of type A and $\mathbb{T}B$ is the set of computations of type B.
- T is a monad when it has sufficient structure needed to turn T-programs into a category.
- T-programs between *finite* types define a Lawvere theory, which gives operations and equations for the effect T.

We can encode different computational effects T as follows,

- list(X) ::= finite list of elements of x
- maybe(X) ::= $X + \bot$
- exception(X) ::= $X + E$
- side-effects(X) ::= $S \to S \times X$
- continuations(X) ::= $(X \to R) \to R$

2.1. **Algebra for monads.** Both algebras and monads are defined as endofunctors. A monad is an endofunctor M equipped with two natural transformations that satisfy some coherence conditions. The components of these transformations at a are:

- $\eta_a : a \to Ma$
- $\mu_a : M(Ma) \to Ma$

Likewise, an algebra for the same endofunctor is a selection of particular object a along with an evaluator morphism,

$$\alpha : Ma \to a$$

If T is any monadic encapsulation of a computational effect, then an algebra for T, T-alg, is a set A along with an evaluator morphism, $alg : T(A) \to A$ such that the following diagrams commute,

$$\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, "id_A"'] & T(A) \arrow[d, "alg"] \\
& A
\end{tikzcd}$$

$$\begin{tikzcd}
T(T(A)) \arrow[r, "id_{T(A)}"] \arrow[d, "Talg"'] & T(A) \arrow[d, "alg"] \\
T(A) \arrow[r, "alg"'] & A
\end{tikzcd}$$

that is,

- $\alpha \circ \eta_A = id_A$
- $\alpha \circ id_{T(A)} = \alpha \circ T\alpha$

3. Lawvere Theories and Monads

3.1. **From Monads to Lawvere Theories.** Suppose $\mathbf{L}$ is a multi-sorted Lawvere theory with a product preserving functor $\Pi : (\mathbf{Set}/\mathbf{S})^{op} \to \mathbf{L}$, which is the identity on objects. There is an adjoint pair of functors between the category of functors $\mathbf{L} \to \mathbf{Set}$ i.e., $\mathbf{Mod}(\mathbf{L}, \mathbf{Set})$ and $(\mathbf{Set}/\mathbf{S})$.

The functor,

$$\mathbf{Set}/\mathbf{S} \xrightarrow{\Pi^{op}} L^{op} \xrightarrow{y} \mathbf{Mod}(\mathbf{L}, \mathbf{Set})$$

is left adjoint to the functor,

$$\mathbf{Mod}(\mathbf{L}, \mathbf{Set}) \xrightarrow{\mathbf{Mod}(\Pi, \mathbf{Set})} \mathbf{Mod}((\mathbf{Set}/\mathbf{S}), \mathbf{Set})$$

To prove this adjunction, we need to show that there exists a natural isomorphism,

$$\mathrm{Hom}_{\mathbf{Mod(L,\ Set)}}((y(\Pi^{op}(n \to S))), G) \cong \mathrm{Hom}_{\mathbf{Set/S}}(n \to S, G \circ \Pi \circ i)$$

where n is any set and i is a canonical map that sends $s \in S$ to objects $s : 1 \to S$ in $(\mathbf{Set}/\mathbf{S})^{op}$.

From yoneda lemma, we know that,

$$\mathrm{Hom}_{\mathbf{Mod(L,\ Set)}}((y(\Pi^{op}(n \to S))), G) \cong G(\Pi(n \to S))$$
$$\mathrm{Hom}_{\mathbf{Set/S}}(n \to S, G \circ \Pi \circ i) \cong \mathbf{Set}/\mathbf{S}(n \to S, G\Pi i)$$

Moreover, since $i : \mathbf{S} \to (\mathbf{Set}/\mathbf{S})^{op}$ is also an embedding,

$$y^{op} : \mathbf{S} \to (\mathbf{Set}^{\mathbf{S}})^{op}$$

from this, we can conclude that the two functors adjoint since there exists an isomorphism between,

$$G\Pi i \cong \mathbf{Set}/\mathbf{S}(i\_, G\Pi i)$$

The monad of lawvere theory $\mathbf{L}$ is arises from the above adjunction as $T : \mathbf{Set}/\mathbf{S} \to \mathbf{Set}/\mathbf{S}$.

If T is any monadic encapsulation of a computational effect and let $\mathbf{L}_T$ denote the category of T-programs between finite types. The opposite category L_T^{op}, obtained by formally reversing the arrows, defines a Lawvere theory.

Furthermore, a model in such a theory is a functor, $L_T^{op} \to \mathbf{Set}$ defined by,

$$\begin{gathered} 1 \to A \\ 2 \to A^2 \\ \cdots \\ n \to A^n \end{gathered}$$

and mapping all arrows $n \to m$ in $\mathbf{L}_T$ to $A^m \to A^n$.

3.2. **From Lawvere Theories to Monads.** Given a Lawvere theory $(\mathbf{L}, I)$ one can construct a finitary monad as follows, We can describe the monad T_L by the following colimit (coend):

$$T_L X = \int^{n \in \aleph_0} L(n, 1) \times X^n$$

i.e., we construct the required monad as a categorical colimit. This colimit is the co-product over all natural numbers n of the set $L(n, 1) \times X^n$, factored by identifying elements determined by taking projections and diagonal maps of $\aleph_0^{op}$.

3.2.1. *Example.* A model for the $list$-theory $\mathbf{L}_{list}$ consists of

- a set A.
- a function $A^n \to A$ for each n-ary operation $1 \to n$.
- equations determined by the composition of programs in the respective category $\mathbf{Kl}_{list}$ of list programs.

The category $\mathbf{Kl}_{list}$ of list programs can be generated by using list(X) by enforcing the necessary laws,

- a list program $f : A \rightsquigarrow B$ is a function from A to lists in B i.e $f : A \to list(B)$.
- identity id_A on each object is defined by the morphism, $singleton : A \to list(A)$.
- given $f : A \to list(B)$ $g : list(B) \to list(C)$ the kliesli composite can be given by, applying f after mapping over $list(B)$ and concatenating the result.

$$\begin{tikzcd}
 & list(B) \arrow[dr, "g^\star"] & \\
A \arrow[ur, "f"] \arrow[rr, "g^\star o f"'] & & list(C)
\end{tikzcd}$$

4. Advantages of Lawvere Theories

The above section shows how one can get a monad from a lawvere theory and vice versa. Although there is a direct correspondence between them following are the benefits of lawvere theories over monads [14].

- Each monad acts on only one category whereas one can define models of Lawvere theory in any category with finite products.
- It is easier to have a sum Lawvere theories in **Law** than it is to sum monads [15].
- Lawvere theory operations can be intertwined via tensor products.
- Lawvere theories are generated by computationally natural operations satisfying computationally meaningful equations.

5. Encoding Effects

There are two ways to go about encoding effects. The first involves constructing a monad starting with a notion of computation T and enforcing that T-programs should form a category. On the other hand, we can start with operations that give rise to the effect at hand.

5.0.1. *Example:* Exceptions. We start with a notion of computation $exceptions(X) := X + E$ where E is the set of possible exceptions. We subsequently define a unit as $\eta : X \to X + E$ and bind the operation to form an exception monad.

On the other hand, one can construct a Lawvere theory ($\mathbf{L}_E$) based on an E-indexed family of nullary operations $raise : 0 \to E$ for each $e \in E$ and no equations. Furthermore, we can generate a monad $T_E = \_ + E$ as in some category $\mathbf{C}$ with co-products the forgetful functor $U_L : \mathrm{Mod}(\mathbf{L}_E, \mathbf{C}) \to \mathbf{C}$ induces a monad.

6. Lawvere Theories to Algebraic Effects & Handlers

We saw in the last section how one could encode effects using Lawvere's theories, starting operations on a notion of computation. Consider again the case of exceptions, where we have nullary operations, one for each e in E modeled using $\mathbf{raise}_e$ for raising exceptions. The monad induced by this theory $TX = X + E$ then maps the set X to set $X + E$ i.e. the disjoining union of A and E.

But, in practice, just raising exceptions is not enough. The notion of exceptions is accompanied by exception handling. Therefore, it is natural to consider the natural transformation to model the exception-handling binary operation,

$$\begin{aligned}
&\mathtt{handle}_e : (TX)^2 \to TX\\
&\mathtt{handle}_e(e, b) = e\\
&\mathtt{handle}_e(a, b) = b \ \forall a \neq e
\end{aligned}$$

The above operation, however natural, does not satisfy the coherence conditions needed to be algebraic. To understand this consider X to be the type $\mathtt{Bool}$ then, for any, $f : \mathtt{Bool} \to T(\mathtt{Bool})$ such that $f(\mathtt{true}) = e$ then we have,

$$\begin{aligned}
&f^\dagger(\mathtt{handle}_e(\mathtt{true}, \mathtt{false})) = f^\dagger(\mathtt{true}) = e\\
&\mathtt{handle}_e(f^\dagger(\mathtt{true}), f^\dagger(\mathtt{false})) = \mathtt{handle}_e(e, f^\dagger(\mathtt{false})) = f^\dagger(\mathtt{false})
\end{aligned}$$

We can generalize this notion of handlers by having handlers for all algebraic effects, i.e., handlers are constructs that give semantics to effectful computations. All of the effects from the previous section are algebraic except for continuations. We can think of algebraic operations as *constructors* that create effects and handlers as *deconstructors*. One can create monads from theories based only on constructors, and the existence of deconstructors does not matter.

7. Case of Continuations

The monads that arise from Lawvere theories all have a finite or countably infinite *rank* i.e., they preserve κ-filtered colimits [13] for some cardinal κ. Moreover, we also saw that there are some operations like *handle* that are not algebraic. In the case of continuations, we have a monad of form $R^{R^{-}}$ for an object R (of answers),

- $TX = (X \to R) \to R$
- $\eta_X : X \to (X \to R) \to R = \lambda k.kx$
- $\mu_X : (Y \to R) \to R \to (X \to R) \to R = \lambda k.f(\lambda g.gk)$

Looking at this structure from a Lawvere theory perspective,

- This monad does not have a rank.
- None of the natural operations are algebraic i.e consider the typed version of call/cc, $C :: ((\alpha \to \beta) \to \beta) \to \beta$, this can be modeled as a transformation of form

$$C_X : \neg T \neg TX \to TX$$

where $\neg TX = T\alpha^X$. This is not the form of an algebraic operation. It has been conjectured that its best to think of continuations as a *logical* operation rather than an *algebraic* one [5].

To reason about monads without rank, we need a notion of large lawvere theory.

Furthermore, we saw that to get a monad from a Lawvere theory, one has to use the Yoneda embedding. Incidentally, yoneda embedding is in some sense the same as a CPS monad. We saw that, for continuations η_X does not give a computation that *returns* a value instead it passes the value to a continuation meaning we need to provide a continuation to get the value out. The Yoneda embedding takes a category C^{op} and produces a category $\mathrm{hom}(\mathbf{C}, \mathbf{Set})$ such that there is a functor,

$$y : C^{op} \to \mathrm{hom}(\mathbf{C}, \mathbf{Set})$$

that maps,

- $y(f) : (X \to Y) \mapsto (\mathrm{hom}(Y, \_) \to \mathrm{hom}(X, \_))$
- $y(x) : X \mapsto \mathrm{hom}(X, \_)$

The crux of this result is that, intuitively, any object in any category is completely determined by its relationship to other objects in that category. If we put object R (answer type) in place of the _ then,

- $y(f) : (X \to Y) \mapsto ((Y \to R) \to (X \to R))$
- $y(x) : X \mapsto (X \to R)$

Likewise, in Haskell, we can see the correspondence between yoneda embedding and CPS translation clearly as follows,

```
-- In the category of Hask.
-- Hask ⊂ Set
-- Yoneda Lemma: [Hask, Hask] (hom(c,_), f) ≅ f c
-- where f is a functor
-- and hom-functor, hom(c,_) = (c -> r)

-- Yoneda Lemma: Container for every f
-- with a constructor Yoneda and deconstructor
-- runYoneda. A Yoneda f a contains a polymorphic
-- function (a -> r) -> f r.
```

```
newtype Yoneda f a =
Yoneda { runYoneda :: forall r. Functor f => (a -> r) -> f r}

-- Yoneda Lemma: forall f. Functor f => Yoneda f a ≅ f a
-- i.e eta :: forall x. (c -> x) -> f x ≅ f c

-- LHS of isomorphism:
leftYoneda :: Functor f => Yoneda f a -> f a
leftYoneda (Yoneda f) = f id

-- RHS of isomorphism:
rightYoneda :: Functor f => f a -> Yoneda f a
rightYoneda fa = Yoneda $ \k -> fmap k fa

-- Yoneda Id r ≅ Id r ≅ r
-- runYoneda :: forall r. (a -> r) -> r

-- The Continuation Passing Transform
-- CPS: forall r. (a -> r) -> r
newtype CPS a = CPS { runCPS :: forall r. (a -> r) -> r }
newtype Id a = Id { runId :: a }

-- Consider f: b -> a then, f-cpsed: b -> Yoneda Id a
-- Example: a = Int, b = String
f :: Int -> String
f = show
fCPS :: Int -> Yoneda Id String
fCPS x = Yoneda (\k -> Id (k (f x)))
```

Appendix A. Appendix

A.1. **Category.** A category $\mathbf{C}$ consists of

- a collection of objects.
- a collection of arrows.
- for each arrow f, objects dom_f and cod_f are called the *domain* and *codomain* of f. If $\mathrm{dom}_f = A$ and $\mathrm{cod}_f = B$, we also write $f : A \to B$,
  - given $f : A \to B$ and $g : B \to C$, so that $\mathrm{dom}_g = \mathrm{cod}_f$, there is an arrow $g \circ f : A \to C$,
  - an arrow $\mathbf{1}_A : A \to A$ for every object A of $\mathbf{C}$,
- such that the following laws are valid,
  - Associative law: for every $f : A \to B$, $g : B \to C$ and $h : C \to C$ we have $h \circ (g \circ f) = (h \circ g) \circ f$.
  - Unit law: for every $f : A \to B$ we have $f \circ \mathbf{1}_A = f = \mathbf{1}_B \circ f$ where $\mathbf{1}_A$ is the identity morphism.

Since anything that satisfies this definition is a category we can say that category theory is the abstract algebra of arrows. To construct a category, this question has to be answered: given a mathematical structure what are the morphisms preserving this structure? Which in turn gives us the arrows of a category. In each category $\mathbf{C}$ with two objects a and b there exists a collection,

$$\mathrm{hom}(a, b) \;=\; \{f \mid f \text{ is an arrow of } \mathbf{C} \text{ and } f : a \to b\}$$

hom-set is not always a set as sometimes it might be *too big* to be considered under the ZF axioms.

A.2. **Opposite Category.** *Opposite Category* is a category that exists as a dual of any other category. It is constructed by reversing the direction of all the arrows in let's say $\mathbf{C}$ and keeping the objects same to get C^{op}. Since, C^{op} is the dual of $\mathbf{C}$ the product in $\mathbf{C}$ becomes sum in C^{op}, the terminal object becomes the initial and so on.

A.3. **Subcategory.** *Opposite Category* is a category that exists as a dual of any other category. It is constructed by reversing the direction of all the arrows in let's say $\mathbf{C}$ and keeping the objects same to get C^{op}. Since, C^{op} is the dual of $\mathbf{C}$ the product in $\mathbf{C}$ becomes sum in C^{op}, the terminal object becomes the initial and so on.

A.4. **Skeleton.** A strict category $\mathcal{C}$ is called skeletal if any two objects that are isomorphic are actually already equal (in the fixed set of objects $\mathrm{ob}(\mathcal{C})$). A skeleton of a category $\mathcal{C}$ is defined to be a skeletal subcategory of $\mathcal{C}$ whose inclusion functor exhibits it as equivalent to $\mathcal{C}$. A weak skeleton of $\mathcal{C}$ is any skeletal category that is weakly equivalent to $\mathcal{C}$.

A.5. **Set.** is a category of sets and set-theoretic functions as morphisms. This is a category of mathematical objects without any structure and since we associate arrows to enforce structure the morphisms in *Set* are special relations which themselves are basically *sets* of input-output pairs.

A.6. **Functor.** When considering mathematical structures as categories themselves, the morphisms preserving this structure are called functors or covariant functors. These functors act as arrows between categories, implying that categories (specifically, small ones) form a category (**Cat**) with functors as arrows.

A category, in itself, formalizes the intuitive idea of "structure" with two components: objects and morphisms. Similarly, to preserve this structure, a functor between two categories **C** and **D** must have two component functions. First, $F_0 : x \to F(x)$ where $x \in \mathbf{C}$ and $F(x) \in \mathbf{D}$. Second, $F_1 : f \to F(f)$ where f is an arrow in $\mathbf{C}$, i.e., $f : x \to y$, and $F(f)$ is an arrow in $\mathbf{D}$, specifically $F(f) : F(x) \to F(y)$. These functions satisfy certain conditions concerning hom-sets, preserving sources and targets of morphisms, identity morphisms, and compositions of morphisms.

Functors embed the source category into the target category. While they may cover only a part of the target category, all objects in the source category must have a functor to produce an object in the target. However, not all objects in the target category need to be pointed by these functors.

A.7. **Natural Transformation.** When we ask "What are morphisms that preserve this structure?" with the mathematical structure under consideration as functors then the structure-preserving morphisms are called natural transformations.

Given two categories $\mathbf{C}$ and $\mathbf{C}$ and functors $F : \mathbf{C} \to \mathbf{C}$, $G : \mathbf{C} \to \mathbf{C}$ we define a natural transformation as $\alpha : F \Rightarrow G$ such that, α assigns every object x in $\mathbf{C}$, the arrow $\alpha_x : Fx \to Gx$ in $\mathbf{C}$ such that,

$$\text{for any } f : x \to y \text{ in } \mathbf{C} \text{ we have, } \alpha_y \circ F(f) = G(f) \circ \alpha_x.$$

A.8. **Composition of Natural Transformationsn.** Consider two functors F, G between two categories $\mathbf{C}, \mathbf{C}$ such that a /natural transformation/, $\alpha : F \to G$ is taken to be any rule that takes an object $a \in \mathbf{C}$ and an arrow $\alpha_a : Fa \to Ga$ in a way that the following diagram commutes for every arrow $f : a \to b$ where $f \in \mathbf{C}$, In other words, $Gf \circ \alpha_a = \alpha_b \circ Ff$ for all f.

A.9. **Adjunctions.** An adjunction is a weaker notion of equivalence in the sense that it does not impose the natural isomorphism between the composition of functors and identity. Instead, it imposes the existence of the following isomorphism: for two functors F, G between $\mathbf{C}$ and $\mathbf{D}$ such that for all $c \in \mathbf{C}$ and $d \in \mathbf{D}$,

$$hom_{\mathbf{D}}(Fc, d) \cong hom_{\mathbf{C}}(c, Gd)$$

A.9.1. *Example of Adjunctions.*

Curry in Haskell. In the category of sets ($\mathbf{Set}$) all the objects can be combined by forming their Cartesian product. This is just another object in $\mathbf{Set}$: $A \times B$, which is the set of all ordered pairs (x, y) such that $x \in A$ and $y \in B$.

We can also express another set or object as B^A in that it is the same as $hom_{\mathbf{Set}}(A, B)$ i.e. it has all the functions between A and B.

Now, let's fix the set B we get from the following functors:

$$\_ \times b : \mathbf{Set} \to \mathbf{Set}$$
$$(\_)^b : \mathbf{Set} \to \mathbf{Set}$$

If one thinks of types as sets, then the morphism $\lambda.g : \_ \to (\_)^B$ represents the function that takes in a value from some arbitrary type and returns the function type. Moreover, if we are given this λg then we can easily get $g : \_ \times B \to C$ and vice versa,

From, $g : \_ \times B \to C$, we can define a $g' : \_ \to C^B$ as $(g'(a))(b) = g(a, b)$ where $a \in \_$ and $b \in B$. Similarly from, $f : \_ \to C^B$ we can define a $f' : \_ \times B \to C$ as $f'(a, b) = (f(a))(b)$ where $a \in \_$ and $b \in B$.

Connections of Galois. Partially Ordered Sets: A set where elements have order among them, i.e., some way to compare every single element t, but it is partial in the sense that pairs of elements need not be comparable. Monotone Functions: These are the functions between ordered sets that preserve the order, i.e., when plotted, they either increase or decrease throughout their domains. Now, let $(A, \leq)$ and $(B, \leq)$ be two partially ordered sets, and then the (monotone) Galois connection between these posets consists of two monotone functions: $f : A \to B$ and $g : B \to A$, such that for all $a \in A$ and $b \in B$,

$$f(a) \leq b \text{ if and only if } a \leq g(b)$$

From the lens of category theory, one can say that a Galois connection is nothing but an adjoint $f \dashv g$ equipped with the essential property that an upper or lower adjoint of a Galois connection uniquely determines the other: $f(a)$ is the least element $minB$ in set B when $a \leq g(minB)$ and $g(b)$ is the largest element $maxA$ in set A when $f(maxA) \leq b$.

A.10. **Yoneda Lemma.**

Category of Presheaves.

Presheaf. A presheaf is a $F : \mathbf{C}^{op} \to \mathbf{Set}$ such that for any $x \in \mathbf{C}$, Fx in $\mathbf{Set}$ is the set that represents the *ways* x can occur in F, and any mapping $f : x \to y$ where $f, y \in \mathbf{C}$, the corresponding $Ff : Fy \to Fx$ maps each of the y's of Fy to each of the x's in Fx.

Representable Presheaf. The above presheaf F becomes *representable* when it is naturally isomorphic to a hom-functor $\mathbf{hom_C}(\_, X) : \mathbf{C}^{op} \to \mathbf{Set}$, which maps any object $c \in \mathbf{C}$ to the hom-set $\mathbf{hom_C}(c, X)$, and each $f : c' \to c$ where $f, c' \in \mathbf{C}$ to the function that maps each morphism $c \to X$ to the composite $(c' \to c) \to X$. Here, the object X is uniquely determined up to an isomorphism in $\mathbf{C}$ and is called the representing object.

Yoneda Lemma.

"The set of morphisms from a representable presheaf $y(c)$ into an arbitrary presheaf X is in natural bijection with the set $X(c)$ assigned by X to the representing object c."

In simple words, if we have a functor F that goes from $\mathbf{C}$ to $\mathbf{Set}$, then the natural transformation between F and the hom-functor $\mathbf{hom_C}(\_, c)$ corresponds by a natural isomorphism (set-theoretic bijection) to the set Fc.

Proof Sketch.

- Consider the category of presheaves on a locally small category $\mathbf{C}$ denoted as $\mathbf{Set}^{\mathbf{C}^{op}}$ with the functor,

  $$y : C \to \mathbf{Set}^{\mathbf{C}^{op}} \quad \forall c \in \mathbf{C},\ y = c \mapsto \mathbf{hom_C}(\_, c)$$

  y sends each object of $\mathbf{C}$ to the hom-functor into that object, i.e., presheaf represented by c.
- We aim to prove that for any $X \in \mathbf{Set}^{\mathbf{C}^{op}}$, there is an isomorphism between the hom-set of the presheaf functors from $y(c)$ to X and the value of X at c,

  $$\mathbf{hom}_{\mathbf{Set}^{\mathbf{C}^{op}}}(y(c), X) \cong X(c)$$
- Consider the following diagram where $c, b, f \in \mathbf{C}$,

  $$\begin{tikzcd} c \arrow[r, "X"] & hom_{\mathbf{C}}(c,c) \arrow[r, "\eta_c"] \arrow[d] & X(c) \arrow[d] \\ b \arrow[u, "f"] \arrow[r, "hom_{\mathbf{C}}(\_,c)"'] & hom_{\mathbf{C}}(b,c) \arrow[r, "\eta_b"'] & X(b) \end{tikzcd}$$

  Since we know that,
  - $\mathbf{1}_c \in \mathbf{hom_C}(c, c)$
  - $\eta_c(\mathbf{1}_c) \in X(c)$
  - $f \in \mathbf{hom_C}(b, c)$
  - $\eta_b(f) \in X(b)$
- We can have a similar diagram with elements instead of sets,

  $$\begin{tikzcd} 1_c \arrow[r, "\eta_c"] \arrow[d] & \eta_c(1_c) \arrow[d] \\ f \arrow[r, "\eta_b"'] & \eta_b(f) \end{tikzcd}$$

  From the above, it is clear that the natural transformation $\eta : \mathbf{hom_C}(\_, c) \Rightarrow X$ is completely determined by $\eta_c(\mathbf{1}_c) \in X(c)$. This means that to show the naturality condition on any $\eta : \mathbf{hom_C}(\_, c) \Rightarrow X$, it is sufficient to show that η is already fixed by some value $\eta_c(\mathbf{1_C}) \in X(c)$ of its component $\eta_c : \mathbf{hom_C}(c, c) \rightarrow X(c)$ on $\mathbf{1_C}$.

- In other words, each object of $\mathbf{C}$ is uniquely specified by the arrows into it (or out if it), up to isomorphism. The objects of a category can be uniquely defined in terms of the role they play in the category in terms of their interactions with the whole.

References

[1] Plotkin, G., Power, J. (2002). Notions of computation determine monads. In Lecture Notes in Computer Science (pp. 342–356). https://doi.org/10.1007/3-540-45931-6-24

[2] Hyland, M., Power, J. (2007). The category Theoretic Understanding of Universal Algebra: Lawvere Theories and Monads. Electronic Notes in Theoretical Computer Science, 172, 437–458. https://doi.org/10.1016/j.entcs.2007.02.019

[3] Mac Lane, S. (1971). Categories for the working mathematician. In Graduate texts in mathematics. https://doi.org/10.1007/978-1-4612-9839-7

[4] Moggi, E. (1991). Notions of computation and monads. Information Computation, 93(1), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[5] Hyland, M., Levy, P. B., Plotkin, G., Power, J. (2007). Combining algebraic effects with continuations. Theoretical Computer Science, 375(1–3), 20–40. https://doi.org/10.1016/j.tcs.2006.12.026

[6] G. D. Plotkin and A. J. Power, Semantics for Algebraic Operations (extended abstract), in Proc. MFPS XVII (eds. S. Brookes and M. Mislove), ENTCS, Vol. 45, Amsterdam: Elsevier, 2001.

[7] Plotkin, G., Pretnar, M. (2009). Handlers of Algebraic Effects. In Lecture Notes in Computer Science (pp. 80–94). https://doi.org/10.1007/978-3-642-00590-9-7

[8] Boisseau, G., Gibbons, J. (2018). What you needa know about Yoneda: profunctor optics and the Yoneda lemma (functional pearl). Proceedings of the ACM on Programming Languages, 2(ICFP), 1–27. https://doi.org/10.1145/3236779

[9] The Yoneda Lemma without category theory: algebra and applications. http://boole.stanford.edu/pub/yon.pdf

[10] Milewski, B. (2019). Category Theory for Programmers (New Edition, Hardcover).

[11] Lawvere Theory (nCatLab)

[12] Algebraic Theory (nCatLab)

[13] Filtered limit (nCatLab)

[14] A categorial view of computational effects.

[15] Christoph Luth, Neil Ghani. Composing Monads Using Coproducts.

[16] Borceux, 1994: Handbook of Categorical Algebra, Vol. 2, Ch. 4: Monads

[17] Awodey, 2010: Category Theory, 2nd ed., Ch. 10: Monads and algebras

[18] Riehl, 2016: Category Theory in Context, Ch. 5: Monads and their algebras