# API4KP Proofs

Tara Athan[1]

Athan Services (athant.com), West Lafayette, Indiana, USA
taraathan@gmail.com

**Abstract.** These are the proofs that the M-tree monads satisfy the monad laws.

## 1 Monad Laws

In seminal work that established a theoretical foundation for proving the equivalence of programs, Moggi [**?**] applied the notion of monad from category theory [**?**] to computation. As defined in category theory, a monad is an endofunctor on a category C (a kind of mapping from C into itself) which additionally satisfies some requirements (the monad laws). In functional programming, monads on a particular category are of interest - the category with types as objects and programs as arrows. For example, the List[_] typeclass is a monad, e.g. List[Int] is a type that is a member of List[_].

Each monad has the following transformations (exemplified for the List monad where lists are denoted with angle brackets)

- unit = $\eta^M_A$: A ⇒ M[A] lifts the input into the monad (e.g. unit(2) = ⟨2⟩)
- join = $\mu^M_A$: M[M[A]] ⇒ M[A] collapses nested monad instances by one level (e.g. join(⟨⟨1, 2⟩, ⟨3, 4⟩⟩) = ⟨1, 2, 3, 4⟩)
- map = $M$: (A ⇒ B) ⇒ (M[A] ⇒ M[B]) takes a function between two generic types and returns a function relating the corresponding monadic types (e.g. map( s → 2*s)(⟨1, 2⟩) = ⟨2, 4⟩)

Note that we choose the $\eta^M$ (unit) $\mu^M$ (join) and $M$ map transformations [**?**] as fundamental in this development of the monad laws because it is useful for later discussion on tree structures, whereas the usual theoretical treatment, based on $\eta^M$ and bind = $\mu^M \circ M$, is more concise.

The map transformation is the defining characteristic of any functor, and must satisfy the functor laws:

**Functor Identity** map(id)(y) = y
**Functor Associativity** map(f o g) = map(f) o map(g)

where id is identity function s ⇒ s. The unit and join transformations arise from natural transformations related to M, satisfying:

**Unit from Natural Transformation** map(f)(unit(x)) = unit(f(x))
**Join from Natural Transformation** join(map(join)(x)) = join(join(x))

Further, these transformations must obey the monad laws:

**Monad Left Identity** join(map(unit)(x)) = x
**Monad Right Identity** join(unit(x)) = x
**Monad Associativity** join(map(map(f))(x)) = map(f)(join(x))

Letting the map function for a functor F be denoted by $F$, unit for type A by $\eta^F_A$, join for type A by $\mu^F_A$, and the identity arrow for type A be $1_A$ then the laws take the form

$$
\begin{aligned}
F(1_A) &= 1_{F[A]} \\
F(g \circ f) &= F(g) \circ F(f) \\
F(f) \circ \eta^F_A &= \eta^F_B \circ f \\
\mu^F_{F[A]} \circ F(\mu^F_A) &= \mu^F_{F[A]} \circ \mu^F_{F^2[A]} \\
\mu^F_A \circ F(\eta^F_A) &= 1_{F[A]} \\
\mu^F_A \circ \eta^F_{F[A]} &= 1_{F[A]} \\
\mu^F_B \circ F(F(f)) &= F(f) \circ \mu^F_A
\end{aligned}
$$

where f: $A \Rightarrow B$, g: $B \Rightarrow C$ .

Monads of relevance to API4KP include

**Option:** handles nullability, has subclasses Some, which wraps a knowledge resource, and None, which is empty
**Try:** handles exceptions, has subclasses Success, which wraps a knowledge resource, and Failure, which wraps a (non-fatal) exception
**Future:** handles concurrency, describes a process whose output may become available at some time
**IO:** handles IO side-effects, wraps a knowledge resource and an *item configuration*
**Task:** handles general side-effects, wraps a knowledge resource and a description of a side-effectful task
**Observable:** handles streams, wraps a sequence of knowledge resources that become available over time
**Key-Value Map:** handles labelled structure, a knowledge resource is associated with each key in some set
**Heterogeneous List:** handles a specified pattern of knowledge resource subclasses (e.g. RuleML rulebase together with an OWL ontology defining the sort hierarchy, CL text with sidecar RDF metadata).
**State:** handles state, wraps a knowledge resource (the state) and implements state transitions

These monad functors may be composed; for example, given a basic knowledge expression type E, the type (State o Try o List) [E] = State[Try[List[E]]] may be defined. In general, the composition of monads is not necessarily a monad.

## 2 Either Bifunctor

Because the Either bifunctor (with objects of types Left or Right) is utilized to define the M-tree monad, the following functions regarding Either are useful in the proofs that follow. The function $\vee$ takes two arguments of type function and returns a function that applies those functions in a map-like fashion to the left and right types, respectively.

$$\vee : (A => C) => (B => D) => ((A \,\mathrm{or}\, B) => (C \,\mathrm{or}\, D))$$

Special cases are the right- and left- biased maps

$$R(g) = \vee(1)(g)$$

$$L(f) = \vee(f)(1)$$

which define right- and left-biased Either functors. These functors have the unusual property of commutativity with each other:

$$R(g) \,\mathrm{o}\, L(f) = L(f) \,\mathrm{o}\, R(g) = \vee(f)(g)$$

The function $\gamma$ is similar to $\vee$ in that it takes two arguments of type function, but these two functions are required to have the same output type. The function returned by $\gamma$ unwraps the Either into the common output type.

$$\gamma : (A => C) => (B => C) => ((A \,\mathrm{or}\, B) => C)$$

In the special case of a diagonal Either $A \,\mathrm{or}\, A$, we have a forgetful transformation that forgets whether the value is left or right:

$$\delta : (A \,\mathrm{or}\, A) \Rightarrow A = \gamma(1_A)(1_A)$$

Note the identities

$$\gamma(f)(h) = \delta \,\mathrm{o} \vee(f)(h) = \delta \,\mathrm{o}\, L(f) \,\mathrm{o}\, R(h)$$
$$\gamma(f)(f) = \delta \,\mathrm{o} \vee(f)(f) = \delta \,\mathrm{o}\, L(f) \,\mathrm{o}\, R(f) = f \,\mathrm{o}\, \delta$$

for $f : A \Rightarrow C$, $h : B \Rightarrow C$.

Let $\eta^L$, $\eta^R$ be the left- and right-biased type constructor for Either, respectively. Clearly The following properties are satisfied

$$L(f) \,\mathrm{o}\, \eta^L = \eta^L \,\mathrm{o}\, f$$
$$R(f) \,\mathrm{o}\, \eta^R = \eta^R \,\mathrm{o}\, f$$

New functors can be defined using the Either bifunctor. For example, given functors M and N, we may define a transformation

$$G(f) = \vee(M(f), N(f))$$

It is necessary to verify that $G$ satisfies the functor identity and associativity laws to ensure that G is really a functor. That is

$$G(1) = 1$$
$$G(f \circ g) = G(f) \circ G(g)$$

Now from the definition

$$\begin{aligned} G(1) &= \vee(M(1), N(1)) \\ &= \vee(1, 1 \\ &= 1 \end{aligned}$$

Further

$$\begin{aligned} G(f \circ g) &= \vee(M(f \circ g), N(f \circ g)) \\ &= L(M(f \circ g) \circ R(N(f \circ g)) \\ &= L(M(f)) \circ L(M(g)) \circ R(N(f)) \circ R(N(g)) \\ &= L(M(f)) \circ R(N(f)) \circ L(M(g)) \circ R(N(g)) \\ &= \vee(M(f), N(f)) \circ \vee(M(g), N(g)) \\ &= G(f) \circ G(g) \end{aligned}$$

where the commutativity of $L$ and $R$ is used. Therefore G is a functor.

## 3 Monadic Tree Structures

The monad structures needed for API4KP are a restricted form of the monads seen in functional programming - rather than applying to category of all types, these monads are functors on a smaller category of knowledge resource types. In DOL, the concept of structured expression using sets is introduced. For example, let B be the category of (basic) Common Logic text expressions, and Q[B] = B or Set[Q[B]], where Set[Q[B]] is the typeclass of SetTree-structured Common Logic expressions. In particular, a member of type SetTree[B] is specified by a Set whose members are either basic leaves (of type B) or structured branches (of type Set[Q[B]] = SetTree[B] = Set[B or SetTree[B]]).

The Set monad is appropriate for defining structured expressions in monotonic logics, like Common Logic, because the order and multiplicity of expressions in a collection has no effect on semantics. The semantics of CL is provided by the CL interpretation structure that assigns a truth-value to each basic CL text expression. The truth-value of a set of CL text expressions is true in an interpretation I if each member of the set maps to true in I. The truth value

I(y) of a SetTree-structured CL expression y is defined to be I(flatten(y)), where flatten(y) is the set of leaves of y.

We generalize this approach for defining the semantics of structured expressions to an arbitrary language L with basic expressions E and M-tree structured expressions. We assume that

- M is a monad on the category of types,
- model-theoretic semantics is supplied through an interpretation structure I defined for basic expressions in E and simply-structured expressions $N_0$= M[Left[E, N[E]]],
- a post-condition contract for side-effects is specified by a truth-valued function P(F, y) for all supported void knowledge actions F and all y in E or $N_0$.

Let $N^M[\_]$ be the M-tree monad corresponding to the minimal (finite) fixed point with respect to N of N[E] = M[E or N[E]], where"A or B" is used as an abbreviation for the unbiased Either[A, B] bifunctor introduced in Sec. 2. The unit function for N is the same as the unit function for M composed with the Left constructor. The map and join functions for N are defined from a recursive application of the map and join functions for M. In particular,

$$\begin{aligned} N^M(f) &= M(\vee(f, N^M(f))) = M(L(f) \circ RN^M(f)) \\ \eta^{N^M} &= \mu^{N_p} \\ \mu^{N^M} &= \mu^M \circ M(\delta \circ R(\eta^M \circ \eta^R \circ \mu^{N^M})) \end{aligned}$$

The satisfaction of the monad laws is dependent on the use of the Either bifunctor to handle the union types, so that the left or right intention is preserved even in the case when the types are not disjoint.

Consider the case where N is an M-tree and let Q[E] = E or N[E], i.e. either the basic type E or the M-tree structured type derived from E. For all x, y $\in$ Q[E], define

**level = $\lambda$:** Q[E] $\Rightarrow$ Left[N[E], N[N[E]] = $\eta^L \circ \delta \circ L(\eta^N)$
**flatten = $\phi$:** Q[E] $\Rightarrow$ E or M[Left[E, N[E]]] = $R(\phi \circ \mu^N \circ M(\lambda))$

If E is a type of basic knowledge resources, then $N^M$[E] is the type of M-tree-structured knowledge resources, and Q[E] = E or $N^M$[E] is the corresponding type of knowledge resources (either basic or structured). By convention, the M-tree monad is named by appending "Tree" to the name of the underlying monad; thus, SetTree[E] = Set[E or SetTree[E]].

Then for all y $\in$ Q[E], we may define the interpretation I(y) = I(flatten(y)), with entailments defined accordingly. Implementations that honor the semantics must satisfy P(F, y) = P(F, flatten(y)), where P is a function representing the post-conditions after execution of side-effectful knowledge operation F on the knowledge resource y.

Like other monads, M-tree monads can be combined through composition. For example, a system of multiple concurrent threads may be modelled as a Set-structure of Stream-structured knowledge resources: SetTree o StreamTree.

## 4 Heterogeneous Structures

Suppose A and B are expression types of two languages where an environment provides a semantics-preserving transformation T from B to A. Further suppose that an interpretation mapping I is defined on A or M[A]. The Either type E = A or B defines the basic knowledge expressions in this environment, while structured expressions are N[E] where N is the M-tree monad N[E]. The Either type Q[E] = E or N[E] contains all expressions in this environment, basic or structured.

Using the transformation T from the environment, we may define the interpretation of the M-tree structured expressions in terms of the interpretations of basic expressions in A and operations on monads. In particular,

- S(x) = T(x) if x $\in$ B, x otherwise
- I(x) = I( map(S)(flatten(x)) )

Notice that the expressions of type B are not required to be in a knowledge representation language. They could be in a domain-specific data model based on XML, JSON or SQL. The semantics of expressions of type B are derived from the transformation to type A, the focus knowledge representation language of the environment. API4KP employs this feature to model ontology-based data access (OBDA) and rule-based data access (RBDA).

Structured expressions can always be constructed in a monad that has more structure than necessary for compatibility with the semantics of a given language. For example, List and Stream monads can be used for monotonic, effect-free languages even though the Set monad has sufficient structure for these languages; a forgetful functor is used to define the semantics in the monad with greater structure in terms of the monad of lesser structure. A heterogeneous structure of languages containing some languages with effects and others without effects (e.g. an ECA rulebase supported by ontologies) could thus make primary use of an M-tree monad that preserves order, such as ListTree or StreamTree, while permitting some members of the collection to have a SetTree structure.

## 5 Monadic Trees of Limited Depth

The simplest instances of an M-tree type have only Left components. This is called a simply-structured M-tree, and is defined as $\mathrm{N}_0$[E] = M[Left[E, Unit]]= M[E or Unit] with Unit being the empty type, that is a subtype of all types. Clearly $\mathrm{N}_0$[E] is a subtype of the general M-tree.

We may similarly define a type $\mathrm{N}_k$ = M[E or $\mathrm{N}_{k-1}$] for each positive integer $k > 0$. $\mathrm{N}_0$[E] is a subtype of $\mathrm{N}_1$ and similarly, $\mathrm{N}_k$ is a subtype of $\mathrm{N}_{k+1}$, $k \geq 0$.

M-trees of finite depth may be defined as the smallest parent type of $\mathrm{N}_k$, where each M-tree object has type $\mathrm{N}_k$ for some integer $k \geq 0$.

Each type $\mathrm{N}_k$ may be viewed as a functor, with map, unit and join transformations defined as follows. In the base case

$$\begin{aligned} N_0(f) &= ML(f) \text{ on } \mathrm{N}_0[\mathrm{E}] \\ \eta^{N_0} &= \eta^M \circ \eta^L \text{ on E} \\ \mu^{N_0} &= \mu^M \circ M(\delta) \text{ on } \mathrm{N}_0[\mathrm{N}_0[\mathrm{E}]] \end{aligned}$$

$$\begin{aligned} N_{k+1}(f) &= M(\vee(f, N_k(f))) \text{ on } \mathrm{N}_k[\mathrm{E}] \\ &= M(L(f) \circ RN_k(f)) \\ \eta^{N_{k+1}} &= \eta^M \circ \eta^L \text{ on E} \\ \mu^{N_{k+1}} &= \mu^M \circ M(\delta \circ R(\eta^M \circ \eta^R \circ \mu^{N_k})) \text{ on } \mathrm{N}_{k-j+1}[\mathrm{N}_j[\mathrm{E}]],\ 0 \leq j \leq k \end{aligned}$$

## 6 Proof

We will verify the monad laws for an M-tree N. We may verify the each that $\mathrm{N}_k$ is a functor for each $k \geq 0$, which is sufficient to show that N is a functor. In the base case, $N_0 = ML$. Both $M$ and $L$ are functors, therefore $\mathrm{N}_0$is also a functor.

In the induction step, we assume $\mathrm{N}_k$ is a functor, and consider $N_{k+1}(f) = M(\vee(f, N_k(f)))$. According to the lemma in Sec. 2, then $\vee(f, N_k(f)) = G_k(f)$ defines a functor $\mathrm{G}_k$. Therefore $N_{k+1} = MG_k$ is also defines a functor $\mathrm{N}_{k+1}$.

### 6.1 Unit Belongs to the Monoid

Next we will verify that $\eta^{N_k} = \eta^M \circ \eta^L$ is a natural transformation of $\mathrm{Id} \to \mathrm{N}_k$ for $k \geq 0$. Thus it is necessary to show (on any base type A, with $f$: A $\Rightarrow$ B)

$$N_k(f) \circ \eta^M \circ \eta^L = \eta^M \circ \eta^L \circ f$$

For $k = 0$

$$N_0(f) \circ \eta^M \circ \eta^L = ML(f) \circ \eta^M \circ \eta^L$$

Because M is a monad with $\eta^M$ the natural transformation for Id $\to$ M, we then have

$$N_0(f) \circ \eta^M \circ \eta^L = \eta^M \circ L(f) \circ \eta^L$$

Because $\eta^L$ is a natural transformation for Id $\to$ L, we further have

$$N_0(f) \circ \eta^M \circ \eta^L = \eta^M \circ \eta^L \circ f$$

as required.

For the induction step, suppose that the law is satisfied for some $k \geq 0$. We wish to show

$$N_{k+1}(f) \circ \eta^M \circ \eta^L = \eta^M \circ \eta^L \circ f$$

From the definition of $N_{k+1}$ the left-hand side is expanded

$$\begin{aligned} N_{k+1}(f) \circ \eta^M \circ \eta^L &= M(L(f) \circ RN_k(f)) \circ \eta^M \circ \eta^L) \\ &= \eta^M \circ L(f) \circ RN_k(f) \circ \eta^L \end{aligned}$$

The R functor on the output of $\eta^L$ is like identity, so we have

$$\begin{aligned} N_{k+1}(f) \circ \eta^M \circ \eta^L &= \eta^M \circ L(f) \circ \eta^L \\ &= \eta^M \circ \eta^L \circ f \end{aligned}$$

as before, and the law is verified. Note that in the induction step, the assumption is not actually used, so the law could be proved without induction.

### 6.2 Closure of the Monoid

We need to show that on the type $\mathrm{N}_{q-r}[\mathrm{N}_r[\mathrm{N}_{p-q}[\mathrm{A}]]]$ $(p \geq q \geq r \geq 0)$

$$\mu^{N_p} \circ N_{q-r}(\mu^{N_{r+p-q}}) = \mu^{N_p} \circ \mu^{N_q}$$

In general, on $\mathrm{N}_{q-r}[\mathrm{N}_r[\mathrm{N}_{p-q}[\mathrm{A}]]]$

$$\begin{aligned} \mu^{N_p} \circ \mu^{N_q} &= \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}})) \circ \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_q})) \\ &= \mu^M \circ \mu^M \circ M(M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}}) \circ \delta \circ R(\eta^{MR} \circ \mu^{N_q})) \\ &= \mu^M \circ M(\mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}}) \circ \delta \circ R(\eta^{MR} \circ \mu^{N_q})) \\ &= \mu^M \circ M(\mu^{N_{p-1}} \circ \delta \circ R(\eta^{MR} \circ \mu^{N_q})) \\ &= \mu^M \circ M(\delta \circ L(\mu^{N_{p-1}}) \circ R(\mu^{N_{p-1}}) \circ R(\eta^{MR} \circ \mu^{N_q})) \\ &= \mu^M \circ M(\delta \circ L(\mu^{N_{p-1}}) \circ R(\mu^{N_{p-1}} \circ \eta^{MR} \circ \mu^{N_q})) \\ &= \mu^M \circ M(\delta \circ L(\mu^{N_{p-1}}) \circ R(\mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}}) \circ \eta^{MR} \circ \mu^{N_q})) \\ &= \mu^M \circ M(\delta \circ L(\mu^{N_{p-1}}) \circ R(\mu^M \circ \eta^M \circ \delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}}) \circ \eta^R \circ \mu^{N_q})) \\ &= \mu^M \circ M(\delta \circ L(\mu^{N_{p-1}}) \circ R(\delta \circ \eta^R \circ \eta^{MR} \circ \mu^{N_{p-1}} \circ \mu^{N_q})) \\ &= \mu^M \circ M(\delta \circ L(\mu^{N_{p-1}}) \circ R(\eta^{MR} \circ \mu^{N_{p-1}} \circ \mu^{N_q})) \end{aligned}$$

In the case $q = r$, on $\mathrm{N}_0[\mathrm{N}_q[\mathrm{N}_{p-q}[\mathrm{A}]]]$ $(p \geq q)$

$$\begin{aligned}
\mu^{N_p} \circ N_0(\mu^{N_{p-1}}) &= \mu^{N_p} \circ ML(\mu^{N_p}) \\
&= \mu^{N_p} \circ ML(\mu^{N_p}) \circ MR(\mu^{N_q}) \\
&= \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}} \circ \mu^{N_q}) \circ ML(\mu^{N_p}) \\
&= \mu^M \circ M(\delta \circ L(\mu^{N_{p-1}}) \circ R(\eta^{MR} \circ \mu^{N_{p-1}} \circ \mu^{N_q}))
\end{aligned}$$

So the law is proved in the case $q = r$. Now assume the law is true for $r + i \geq q \geq r$, for some $i \geq 0$. That is, for $0 \leq j = q - r \leq i$, on $\mathrm{N}_j[\mathrm{N}_r[\mathrm{N}_{p-q}[\mathrm{A}]]]$

$$\mu^{N_p} \circ N_j(\mu^{N_{p-j}}) = \mu^{N_p} \circ \mu^{N_q}$$

Let $j = i + 1$, $q = r + i + 1$ and $s = r + p - q = p - i - 1$. On $\mathrm{N}_j[\mathrm{N}_r[\mathrm{N}_{p-q}[\mathrm{A}]]]$

$$\begin{aligned}
\mu^{N_p} \circ N_{i+1}(\mu^{N_{p-i-1}}) &= \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}}) \circ ML(\mu^{N_s}) \circ MRN_i(\mu^{N_s}) \\
&= \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}}) \circ L(\mu^{N_s}) \circ RN_i(\mu^{N_s})) \\
&= \mu^M \circ M(\delta \circ L(\mu^{N_s}) \circ R(\eta^{MR} \circ \mu^{N_{p-1}} \circ N_i(\mu^{N_s}))) \\
&= \mu^M \circ M(\delta \circ L(\mu^{N_{p-1}}) \circ R(\eta^{MR} \circ \mu^{N_{p-1}} \circ N_i(\mu^{N_s}))) \\
&= \mu^M \circ M(\delta \circ L(\mu^{N_{p-1}}) \circ R(\eta^{MR} \circ \mu^{N_{p-1}} \circ N_i(\mu^{N_{p-i-1}}))) \\
&= \mu^M \circ M(\delta \circ L(\mu^{N_{p-1}}) \circ R(\eta^{MR} \circ \mu^{N_{p-1}} \circ \mu^{N_q})) \\
&= \mu^{N_p} \circ \mu^{N_q}
\end{aligned}$$

The induction step is verified, and so the law holds in the general case.

### 6.3 Left Identity

On $\mathrm{N}_{p-k}[\mathrm{N}_{k-1}[\mathrm{A}]]$,

$$\begin{aligned}
\mu^{N_p} \circ N_{p-k}(\eta^N) &= \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}})) \circ N_{p-k}(\eta^N) \\
&= \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}})) \circ M(L(\eta^N) \circ RN_{p-k-1}(\eta^N)) \\
&= \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}}) \circ L(\eta^N) \circ RN_{p-k-1}(\eta^N)) \\
&= \mu^M \circ M(\delta \circ L(\eta^N) \circ R(\eta^{MR} \circ \mu^{N_{p-1}} \circ N_{p-k-1}(\eta^N))) \\
&= \mu^M \circ M(\delta \circ L(\eta^N) \circ R(\eta^{MR})) \\
&= \mu^M \circ M(\delta \circ L(\eta^N) \circ R(\eta^{MR})) \\
&= \mu^M \circ M(\eta^M \circ \delta \circ L(\eta^L) \circ R(\eta^R)) \\
&= M(\delta \circ L(\eta^L) \circ R(\eta^R)) \\
&= M(1_A) \\
&= 1_A
\end{aligned}$$

### 6.4 Right Identity

On $\mathrm{N}_{p-q-1}[\mathrm{N}_q\mathrm{A}]$,

$$\begin{aligned}
\mu^{N_p} \circ \eta^N &= \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}})) \circ \eta^M \circ \eta^L \\
&= \mu^M \circ M(\delta) \circ \eta^N \\
&= \mu^M \circ \eta^M \circ \delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}}) \circ \eta^L \\
&= (\mu^M \circ \eta^M) \circ (\delta \circ \eta^L) \\
&= 1_A
\end{aligned}$$

### 6.5 Associativity

We wish to show that on N[N[A]]

$$\mu^N \circ NN(f) = N(f) \circ \mu^N$$

In terms of the trees of finite depth, this becomes, on $\mathrm{N}_{p-q}[\mathrm{N}_q\mathrm{A}]$

$$\mu^{N_p} \circ N_{p-q}N_q(f) = N_p(f) \circ \mu^{N_p}$$

From the left-hand side

$$\begin{aligned}
\mu^{N_p} \circ N_{p-q}N_q(f) &= \mu^{N_p} \circ M(L(N_q(f)) \circ RN_{p-q-1}N_q(f))) \\
&= \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}})) \circ M(L(N_q(f)) \circ RN_{p-q-1}N_q(f))) \\
&= \mu^M \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}}) \circ L(N_q(f)) \circ RN_{p-q-1}N_q(f))) \\
&= \mu^M \circ M(\delta \circ L(N_q(f)) \circ R(\eta^{MR} \circ \mu^{N_{p-1}}) \circ RN_{p-q-1}N_q(f))) \\
&= \mu^M \circ M(\delta \circ L(N_q(f)) \circ R(\eta^{MR} \circ \mu^{N_{p-1}} \circ N_{p-q-1}N_q(f))) \\
&= \mu^M \circ M(\delta \circ L(N_q(f)) \circ R(\eta^{MR} \circ N_{p-1}(f) \circ \mu^{N_{p-1}})) \\
&= \mu^M \circ M(\delta \circ L(N_q(f))) \circ MR(\eta^{MR} \circ N_{p-1}(f) \circ \mu^{N_{p-1}}) \\
&= \mu^M \circ M(\delta \circ L(N_q(f)) \circ R(\eta^{MR} \circ N_{p-1}(f) \circ \mu^{N_{p-1}})) \\
&= \mu^M \circ M(\delta \circ L(N_p(f)) \circ R(\eta^{MR} \circ N_{p-1}(f) \circ \mu^{N_{p-1}}))
\end{aligned}$$

From the right-hand side

$$
\begin{aligned}
N_p(f) \circ \mu^{N_p} &= M(L(f) \circ RN_{p-1}(f))) \circ \mu^{N_p} \\
&= M(L(f) \circ RN_{p-1}(f))) \circ \mu^{M} \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}})) \\
&= M(L(f) \circ RN_{p-1}(f))) \circ \mu^{M} \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}})) \\
&= \mu^{M} \circ MM(L(f) \circ RN_{p-1}(f))) \circ M(\delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}})) \\
&= \mu^{M} \circ M(M(L(f) \circ RN_{p-1}(f)) \circ \delta \circ R(\eta^{MR} \circ \mu^{N_{p-1}})) \\
&= \mu^{M} \circ M(\delta \circ LM(L(f) \circ RN_{p-1}(f)) \circ RM(L(f) \circ RN_{p-1}(f)) \circ R(\eta^{MR} \circ \mu^{N_{p-1}})) \\
&= \mu^{M} \circ M(\delta \circ L(ML(f) \circ MRN_{p-1}(f)) \circ R(ML(f) \circ MRN_{p-1}(f) \circ \eta^{MR} \circ \mu^{N_{p-1}})) \\
&= \mu^{M} \circ M(\delta \circ L(N_p(f)) \circ R(MRN_{p-1}(f) \circ \eta^{MR} \circ \mu^{N_{p-1}})) \\
&= \mu^{M} \circ M(\delta \circ L(N_p(f)) \circ R(\eta^{MR} \circ N_{p-1}(f) \circ \mu^{N_{p-1}}))
\end{aligned}
$$

Therefore, associativity of the monoid is proved.

We have now shown that all laws that are required for the M-tree to be a monad are satisfied.