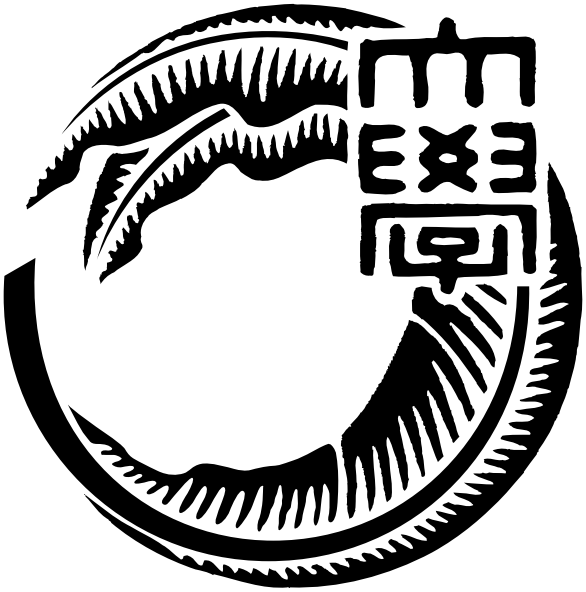
平成26年度 卒業論文

プログラムの変更の圏による形式化



琉球大学工学部情報工学科

115763K 比嘉健太

指導教員 河野真治

目 次

図 目 次

表 目 次

リスト目次

第1章　プログラムの変更の形式化

プログラムの信頼性とはプログラムが正しく動く保証性であり、信頼性は多くの原因により損なわれる。例えば未定義の挙動によりプログラムが停止すること、プログラム内の誤った条件式による誤った計算結果、実行環境やパラメタの変化による誤動作などがある。プログラムの変更は信頼性を低下させるきっかけとなる。変更を形式化することでプログラムの信頼性の変化を指摘する。

プログラムの変更の形式化には Monad を用いる。プログラムにおける Monad とはデータ構造とメタ計算の対応である [8]。プログラムの変更をメタ計算として定義することで、プログラムの変更そのものを計算可能にする。メタ計算として信頼性の解析に利用可能な機構を定義することで、信頼性を保ちながらソフトウェアを開発できることを示す。例えば、プログラムが変更された際に変更前と変更後のプログラムの挙動を比較する機構を提供できる。また、定義したメタ計算の Monad と対応を保証するために、メタ計算が Monad 則を満たすこと証明支援系 Agda [1] により証明する。加えて他の Monad との組み合せが可能であることも証明する。

第2章　プログラムの変更を表現する Delta Monad

Monad を用いてプログラムの変更を形式化していく。第 2 章ではプログラムの変更を表す Delta Monad を定義し、その使用例とメタ計算の例を示す。なお、Monad についての定義と解説は第 3 章と第 4 章にて行なう。

2.1　Monad とメタ計算

まずはプログラムを定義する。プログラムは型付けされた値と、値を値へと写像する関数のみで構成される。プログラムの実行は関数の値への適用とし、入出力といった、値や関数で表現できない計算はメタ計算とする。メタ計算を特定のデータ構造に対応させ、メタ計算が必要な関数は値をデータ構造へと写像することで入出力処理を実現する。メタ計算とデータ構造の対応に用いる性質が Monad である。

例えば、失敗する可能性があるメタ計算 T は式 2.1 のように定義できる。

$$TA = A_{\perp}(i.e. A + \{\perp\}) \tag{2.1}$$

型 A の値に対応するメタ計算 T A は、A と $\perp$ の論理和として表現できる。計算に成功した際は型 A の値を返し、失敗した場合は $\perp$ を返す。

ここで、失敗しない前提で作成されたプログラムに対して、失敗する可能性を表現するメタ計算を対応させる。プログラムは型 A の値 x と、型 A の値を取り型 B の値を返す関数 f, 型 B の値を取り型 C の値を返す関数 g によって構成されるとする (式 2.2)。

$$g(f(x)) \tag{2.2}$$

ここで関数 f は失敗する可能性がある時、f をメタ計算によって拡張することで失敗を実現する。

式 2.1 で定義したように、計算の成功は型 A 値を返し、失敗は $\perp$ を返す。$\perp$ となった時点で計算を中断するメタ計算を定義すれば、失敗を含めた関数を表現できる。計算に失敗した際に対応するメタ計算を定義し、関数をそのメタ計算で拡張することで失敗に対する処理が実現できる。

型 A を持つ x の値をメタ計算と対応して型 T A とした値を x' 、メタ計算による関数の拡張を * と記述すると、式 2.2 の式は式 2.3 のように書ける。

$$g^*(f^*(x')) \tag{2.3}$$

ここで重要な点は、元の関数 f と g から f^* と g^* が導けることである。メタ計算が無い関数 f とメタ計算を持つ関数 f^* が 1 対 1 に対応することは Monad により保証されている。このように、値と関数で表されたプログラムにおいてメタ計算を用いることで、計算を拡張することができる。

2.2 Delta Monad の定義

プログラムとメタ計算の関係としての Monad について述べたところで、プログラムの変更をメタ計算として記述することを考える。

プログラムの変更とは関数や値が変更されることであり、変更される量には単位がある。最初の変更単位をバージョン 1 とし、変更された後のプログラムはバージョンが 1 増加する。任意の型 A に対するメタ計算 T を考えた時、プログラムの変更は式 2.4 のように定義する。

$$TA = V_A \tag{2.4}$$

V はプログラムの全てバージョンの集合であり、V_A とすることで A に対応する値の集合を返すものとする。

式 2.4 のメタ計算として、全ての変更単位で変更されたプログラムを保存する計算を提案する。このメタ計算を Delta Monad と呼ぶ。なお、この Delta Monad が Monad であることの証明は 6 章で行なう

2.3 Haskell における Delta Monad の実装例

式 2.4 のメタ計算をプログラミング言語 Haskell を実装し、プログラムの変更が表現できることを示す。

まずは全てのプログラムのバージョンを表わすデータ型 Delta を考える。Delta の定義はリスト 2.1 とする。

リスト 2.1: Haskell におけるデータ型 Delta の定義

```
1 data Delta a = Mono a | Delta a (Delta a)
```

データ型 Delta はコンストラクタ Delta もしくは Mono によって構成される。バージョンが 1 である値は Mono によって構成される。プログラムを変更する際には、コンストラクタ Delta を用いて記述し、変更後の値と前のバージョンを持つ。Delta で記述するこ

とで、プログラムの変更に沿った長さ 1 以上のリスト構造が構築される。なお、a とは任意の型であり、Delta が任意の型の値に対してもデータ型を構築できることを示す。

Haskell においてメタ計算とデータ型の対応は Monad によって行なうため、 Monad という型クラスが用意されている (リスト 2.2)。

リスト 2.2: Haskell における Monad 型クラス

```
1 class Monad m where
2     return :: a -> (m a)
3     (>>=)  :: (m a) -> (a -> (m b)) -> (m b)
```

型クラスとは特定の性質を持つ型をまとめるための制約であり、制約はデータ型に対して関数の定義を要請する。要請する関数の定義は where 句の後に記述し、関数名と型を定義する。型クラス C が要請した関数を型 a 対してに定義することを「型 a は型クラス C のインスタンスである」と言う。

型クラス Monad に属するために要請される関数は return と >>= である。:: は型注釈であり、 *term* :: *type* のように記述する。なお、関数型は `a -> b` のように引数の型と返り値の型を ->で挟んで記述し、引数が 2 つ以上の関数は `a -> b -> ... -> d` のように ->を増やす。

関数 return は任意の型 a を受けとり、メタ計算と対応された型に対応させて返す。>>= は中置関数であり、left operand と right operand を取る。left operand にメタ計算と対応された値と、right operand にメタ計算と対応された値を返す関数を取り、メタ計算を行ないながら関数を適用する。

データ型 Delta に対応するメタ計算を Monad を用いてリスト 2.3 のように定義する。

リスト 2.3: Haskell におけるデータ型 Delta とメタ計算の関連付け

```
1  headDelta :: Delta a -> a
2  headDelta (Mono  x)   = x
3  headDelta (Delta x _) = x
4
5  tailDelta :: Delta a -> Delta a
6  tailDelta (Mono x)     = Mono x
7  tailDelta (Delta _ ds) = ds
8
9
10 instance Monad Delta where
11   return x = Mono x
12   (Mono x) >>= f     = f x
13   (Delta x d) >>= f  = Delta (headDelta (f x))
14                              (d >>= (tailDelta . f))
```

- 関数 return

  通常の値をメタ計算と対応させるため、値をバージョン 1 の値とする。そのためにコンストラクタ Mono を用いる。

- 中置関数 >>=

  メタ計算を含んだ関数の適用である。通常の関数の適用に加え、バージョンを含んだ計算も行なう。値と関数のそれぞれにおいて同じバージョン選択し、関数を適用する。もしバージョン 1 であった場合はコンストラクタは Mono であるため、Mono が持っている値に対して関数を適用することとなる。もしバージョンが 1 で無い場合のコンストラクタは Delta であるため、先頭の値を用いて計算し、残りの値と結合することとなる。その際、先頭の値を取り出すために headDelta 関数を、先頭以外の値を取り出すために tailDelta 関数を用いる。

なお、中置関数 >>= や headDelta 関数などで用いたコンストラクタによる処理の分岐はパターンマッチと呼ばれる。Haskell ではコンストラクタごとに関数を記述することでパターンマッチを実現する。

2.4 Delta Monad を用いたプログラムの変更の記述例

プログラムの変更を表現するメタ計算に対応するデータ型 Delta が記述できた。2 つのバージョンを持つプログラムを Delta によって表現した例を述べる。

まずは Delta を用いずに記述したプログラムを示す (リスト 2.4)。

リスト 2.4: numberCount バージョン 1

```
1  generator :: Int  -> [Int]
2  generator x     = [1..x]
3
4  numberFilter :: [Int]  -> [Int]
5  numberFilter xs = filter  isPrime 1  xs
6
7  count :: [Int] -> Int
8  count xs        = length xs
9
10 numberCount :: Int -> Int
11 numberCount x = count (numberFilter (generator x))
```

1 から n の間の整数に含まれる特定の数の個数を調べるプログラム numberCount である。このプログラムは以下の 3 つの関数により定義される。

- generator

  1 から n の間の整数を生成する関数である。n を渡すことにより 1 から n の間の整数のリストを返す。

- numberFilter

  数のリストから特定の数のみを絞り込む関数である。バージョン 1 では素数の数のみを絞り込む。

- count

  数の個数を数える関数である。整数はリストで与えられるため、リストの長さが個数であるとした。

ここで numberFilter 関数の定義を変更し、素数による絞り込みから偶数による絞り込みにする。変更後の numberCount プログラムをリスト 2.5 に示す。

リスト 2.5: numberCount バージョン 2

```
1  generator :: Int  -> [Int]
2  generator x     = [1..x]
3
4  numberFilter :: [Int]  -> [Int]
5  numberFilter xs = filter  even 2  xs
6
7  count :: [Int] -> Int
8  count xs        = length xs
9
10 numberCount :: Int -> Int
11 numberCount x = count (numberFilter (generator x))
```

変更は下線の部分で、filter する条件を isPrime から even とした。
numberCount プログラムを Delta によって変更も含めて記述する (リスト 2.6)。

リスト 2.6: Delta を用いたプログラムの例

```
1  generator :: Int -> Delta [Int]
2  generator x = let intList = [1..x] in
3                   Mono intList
4
5  numberFilter :: [Int] -> Delta [Int]
6  numberFilter xs = let primeList =  filter isPrime 1  xs
7                        evenList  =  filter even 2  xs    in
8                   Delta evenList (Mono primeList)
9
10 count :: [Int] -> Delta Int
11 count xs = let primeCount = length xs in
12            Mono primeCount
13
14 numberCount :: Int -> Delta Int
15 numberCount x = count =<< numberFilter =<< generator x
```

変更が無い関数 generator と count の値はバージョン 1 であることを示すために Mono を用いる。変更が存在する numberFiler 関数は Delta を用いて 2 種類の処理を保存している。それぞれのプログラムに対応する部分は下線で示した部分である。

これらの関数 3 つを Monad の =<< によってメタ計算を含む関数呼び出しとして実行する。中置関数 =<< は >>= の左右のオペランドを交換したものである。2 つのバージョンを含めた numberCount を実行すると 2.7 のように、全てのバージョンの結果が得られる。

リスト 2.7: numberCount プログラムの実行結果

```
1 *Example.Delta> numberCount 1000
2 Delta 500 (Mono 168)
```

これは numberCount プログラムに 1000 を与えた結果である。バージョン 1 の時は素数の数を求めるため計算結果は 168 であり、バージョン 2 の時は偶数の数を求めるために計算結果は 500 となる。Delta によりプログラムの変更を表現し、かつ同時に実行することができた。

なお、Delta Monad はプログラムの変更に対する計算もメタ計算として Haskell で実行している。よってメタ計算を変更することで、変更に対する計算を任意に定義し実行する可能となった。つまり、図 2.1 のようにプログラムにおいてバージョンが変わるごとにバージョン間の関係が存在しない状態から、図 2.2 のようにプログラムの変更を含めてプログラムを実行可能となったことを意味する。メタ計算を変更することでプログラムの実行時にバージョン間の挙動の比較することも可能となる。

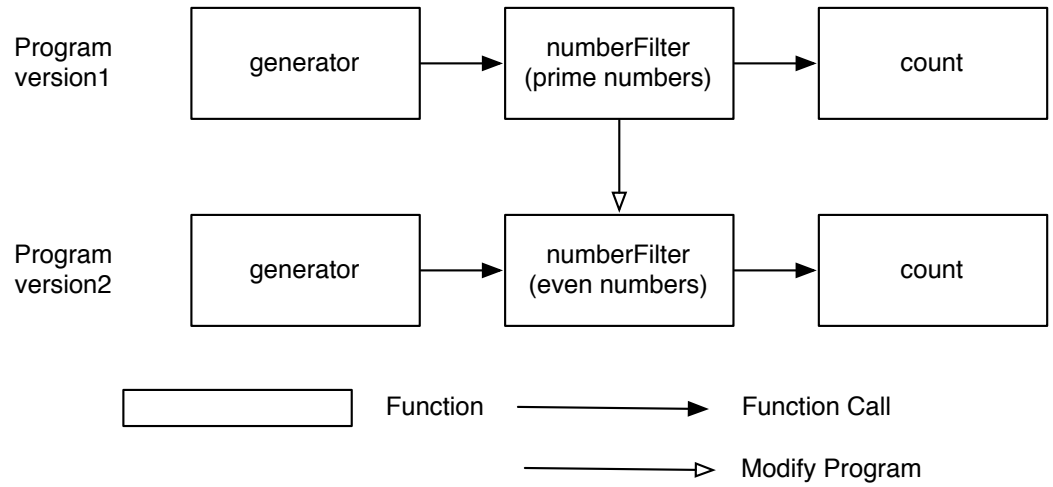


図 2.1: Delta を用いない通常のプログラムの例

2.5 Delta を用いた信頼性向上

2.4 節ではプログラムの変更に対して、変更前と変更後の挙動を保存した例を述べた。プログラムの変更形式化することによって可能となる信頼性向上手法について 2.5 節で述べる。

プログラムの変更を保存することで、以下のようなことが可能となる。

- 異なるバージョンを同時に実行する

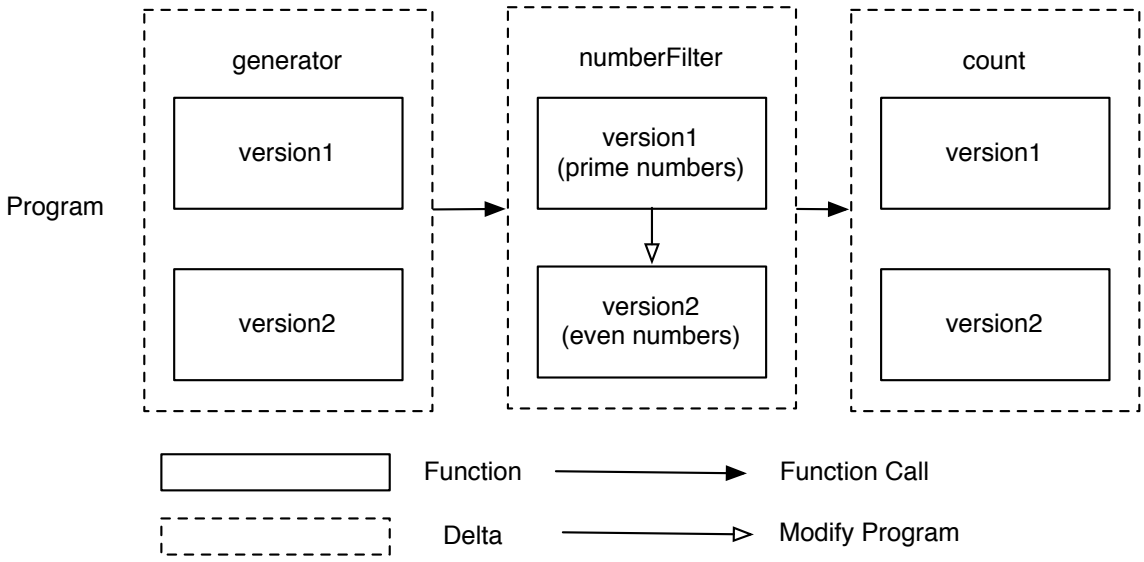


図 2.2: Delta を用いたプログラムの例

プログラムの変更列から任意のバージョン 2 つを取り出し、同時に実行するプログラムを構成する。プログラムを同時に実行することで以下のようなメリットがある。なお、任意の要素どうしの組み合せは category において product として表現されるため、 product と対応があると考えている。

- 実行系とサブの実行系を実行する

  例えば、あるバージョンでリリースしたプログラムがあるとする。変更を加え、ベータ版としてリリースしたいが動作が不安定である。そこで、リリースしたプログラムからベータ版への変更から、2 つのプログラムが同時に動くようなプログラムを構築する。見掛け上は安定版として動作するが、安定版の実際の入出力を用いてベータ版をテストすることが可能となる。

- バージョン依存のプロトコル間で互換を持つようなプログラムが作成できる

  異なるバージョン間でプロトコルに互換が無いプログラムを考える。バージョン間の互換を含めてメタ計算として定義し、全てのバージョンに対して互換を持つプログラムを構築する。そうすることで、どのバージョンのプロトコルとも互換を持つような変換器を作成できる。

- バージョン間の動作の比較

  プログラムの各バージョンにつき、挙動を示すユニークなトレースが得られるとする。トレースの比較をすることでバージョン間の動作を比較することができる。トレースの比較により以下のようなものを検出できると考えている。

- 過去のバージョンの挙動を破壊する時を検出する

  プログラムの変更の際、トレースを変えてはいけない部分を指定する。変更の際にトレースが保存される変更のみを受けつけるようにする。

- トレースが変化していないことを確認する

  プログラムの変更にはいくつかの種類がある。例えば機能拡張の変更などであればトレースは変化するのが正しい。しかしリファクタリングではプログラムが変更されてもトレースが変わらないのが望ましい。トレース全体が変わらないような変更のみを受けつけることにより、リファクタリングを支援することができる。

- Version Control System との対応

  全ての変更を保存し、任意のバージョンを生成可能であるようなメタ計算を考える。そうした時、プログラムの変更を蓄積するものは git や mercurial といった Version Control System の Repository に相当すると考えられる。Delta により Repository を定義することがでるのならば、branch や merge といった Version Control System の処理に対して形式的な定義を与えることができる。また、 category における colimit と対応があると考えている。

- 変更単位を用いた処理

  変更を適用する際に、特定の処理を実行することもできる。例えばプログラムの変更単位に対してテストを行なうことで、変更に応じてテストの結果の変動が確認できる。プログラム全体に対応するテストを定義し、変更の際にテストが通る変更のみ受け付けるようにすることでテストベースの信頼性を確保できる。

他にもプログラムの変更そのものを処理するプログラムを定義することもできる。この機構を言語処理系に組込むことにより、言語の仕様にプログラムの変更方法も含めることができる。例えば、プログラムの変更は許可されたオペレーション内で行なうといった制約を加えることが可能となる。さらにユーザによりプログラムの変更に対するメタ計算を自由に定義できるような言語処理系を作るとする。その処理系ではこれまでに挙げた全てのメタ計算の例を自由に使用、追加することが可能となる。

このように、プログラムの変更を形式化し、変更に基いたメタ計算を定義することで信頼性を向上できる。

第3章　Categorical Definitions of Monad

2 章ではプログラムの変更がメタ計算として Monad を用いて定義可能であることを示した。3 章では category による Monad の定義と要請される Monad 則について述べる。定義は Monad の解説に必要な部分についてのみ解説する。

3.1　Category

まずは Monad の定義に必要な Category (圏) について述べる。
category は 2 つの要素を持つ。

- object (要素)
- morphism (射, arrow)

  object から object へのマッピング。

  morphism によってマップされる対象を domain と呼び、マップした先の対象を codomain と呼ぶ。つまり domain から codomain への object をマップするものが morphism である。domain A と codomain B を持つ morphism f は式 3.1 のように書くこととする。

$$f : A \to B \tag{3.1}$$

そして category は object と moprhism に対して 2 つの法則を満たす。

- 全ての object について identity mapping が存在する

  identity mapping とは domain と codomain が同じ object を指す morphism のことである。identity mapping は id と略する。id は全ての object に存在するため、object A に対する id は id_A と書く。

- 同じ obejct が domain と codomain になっている 2 つの morphism は合成することができ、合成の順番は結果に影響しない。

morphism の合成には記号 ∘ を用いる。object B から C への morphism f と object A から B への morphism g があった時、 f と g を合成すると式 3.2 となる。

$$f \circ g : A \to C \tag{3.2}$$

改めて、morphism の合成の順序が結果に影響しないことを式にすると式 3.3 のようになる。

$$\begin{aligned} f &: C \to D \\ g &: B \to C \\ h &: A \to B \\ (f \circ g) \circ h = f \circ (g \circ h) &: A \to D \end{aligned} \tag{3.3}$$

例えば、object A,B と A から B への morphism を持つ category は図 3.1 のようになる。

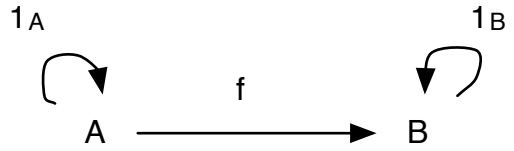


図 3.1: object A,B と morpshim f を持つ category

なお、id は全ての object に存在するために省略して書く。

object を点、morphism を矢印とした図を commutative diagram (可換図式) と呼ぶ。ある object から object への morphism の合成の path に関わらず結果が同じである時、その図を commutative (可換) と呼ぶ。例えば、式 3.3 の category の commutative diagram は図 3.2 のようになる。

commutative diagram が commutative である時、morphism の合成順序の可換性を用いて等価性の証明を行なうことを diagram chasing と呼ぶ。ある性質を category に mapping し、diagram chasing を用いて証明を導くことで性質を解析していく。

3.2 Functor

3.1 節では category を定義した。3.2 節では category から category への mapping である Functor (関手) について解説する。

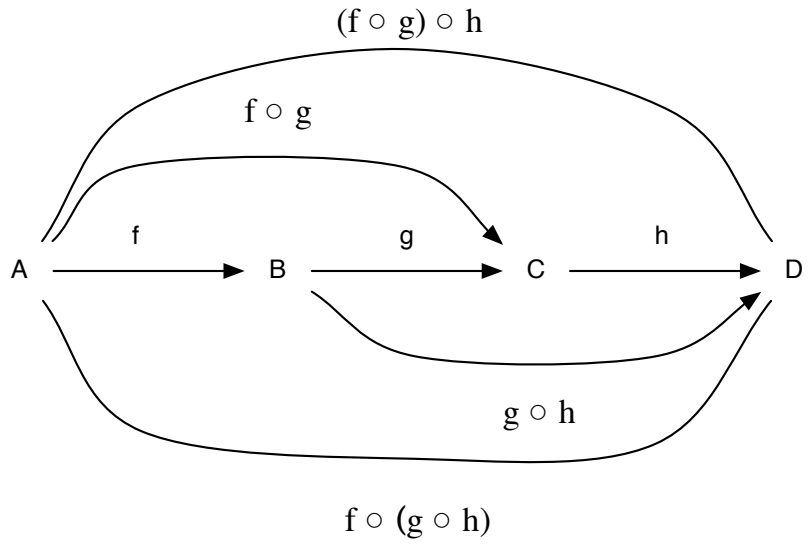


図 3.2: 式 3.3 の morphism の合成法則の category がなす commutative diagram

functor F とはある category C から D への mapping である。functor F を用いて category C の object もしくは morphism である x を category D の object もしくは morphism に対応させることを式 3.4 のように記述する。

$$F(x) \tag{3.4}$$

functor は以下の functor 則を満たす。

- id を保存する

  category C における object A への id 1_A は F A においても id として振る舞う。

$$F(1_A) = 1_{F(A)} \tag{3.5}$$

- morphism の合成を保存する

  category C における morphism f g の合成を考える。合成された $f \circ g$ を F により mapping することと、f と g を個別に F により mapping した後に合成した結果は同じである。

$$F(f \circ g) = F(f) \circ F(g) \tag{3.6}$$

functor の例を挙げる。まず、 3 つの object A,B,C と morphism f,g,h を持つ category C を考える。category C の morphism は式 3.7 であるとする。

$$
\begin{aligned}
f &: A \to C \\
g &: B \to B \\
h &: B \to C \\
f &= g \circ h
\end{aligned}
\tag{3.7}
$$

次に、 2 つの object A', B' と morphism f', g' を持つ category D を考える。category D の morphism は式 3.8 であるとする。

$$
\begin{aligned}
f &: B' \to A' \\
g &: A' \to B'
\end{aligned}
\tag{3.8}
$$

C から D への functor F は式 3.9 のようなものがある。

要素数が異なるため一見正しく mapping できないように思えるが、正しく Functor 則を満たしている。

$$
\begin{aligned}
F(A) &= A' \\
F(B) &= B' \\
F(C) &= A' \\
F(id_A) &= F(id_{A'}) \\
F(id_B) &= F(id_{B'}) \\
F(id_C) &= F(id_{A'}) \\
F(f) &= id_{A'} \\
F(g) &= g' \\
F(h) &= f' \\
F(h \circ g) = F(h) \circ F(g) &= f' \circ g' = id_{A'} = F(f)
\end{aligned}
\tag{3.9}
$$

functor F を用いて category C を category D へと mapping した図が図 3.3 である。

functor により category から category への mapping を考えることが可能となった。

3.3 Natural Transformation

3.2 節では category から category への mapping である functor について述べた。3.3 節では functor と functor の関係である Natural Transformation(自然変換) について述べる。

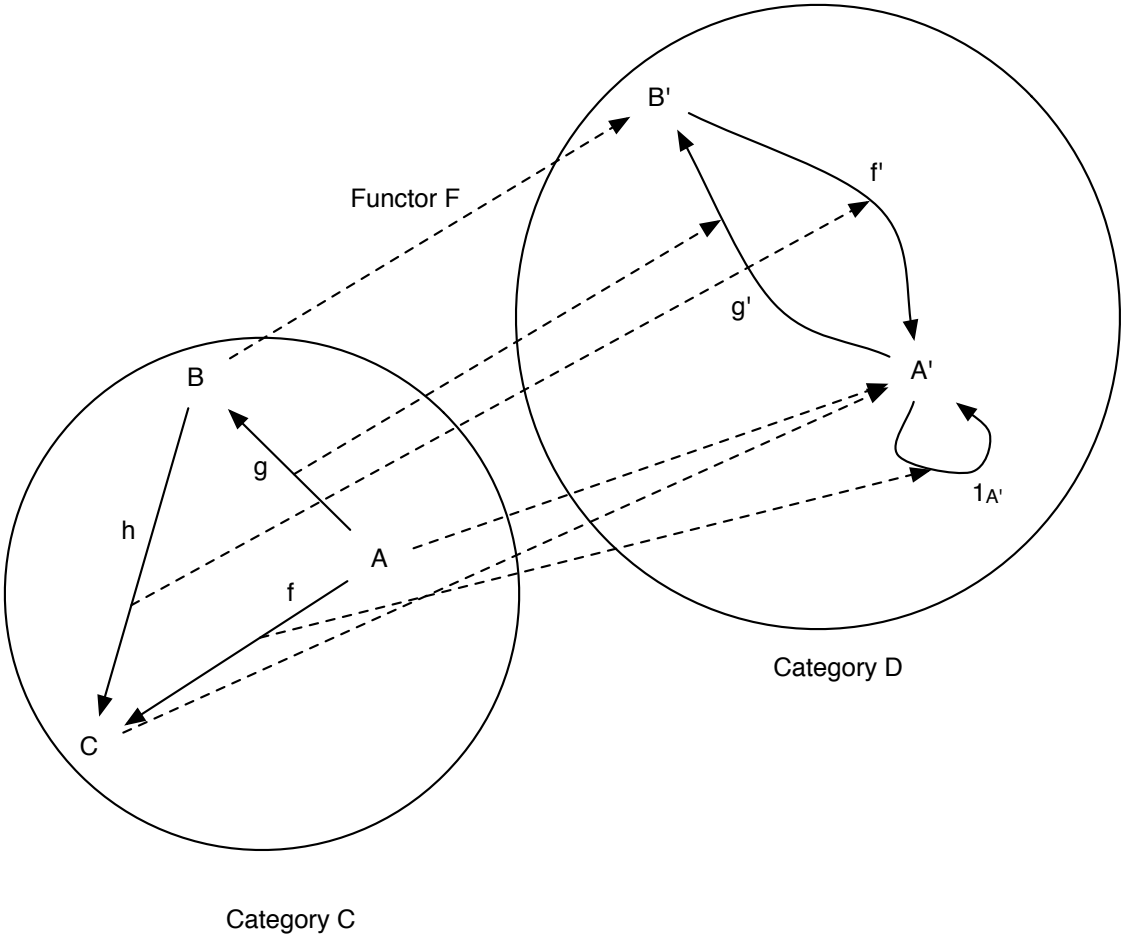


図 3.3: Functor の例

category C から D への functor F, G が存在する時、F から G への変換 t を考える。
t は category C の A に対し、F(A) から G(A) への変換を提供する (式 3.10)。

$$t(A) : F(A) \to G(A) \tag{3.10}$$

t が 式 3.11 を満たす時、t を natural transformation を呼ぶ。

$$\begin{aligned} f : A \to B \\ G(f)t(A) = t(B)F(f) \end{aligned} \tag{3.11}$$

t が natural transformation である時、 category A における A を functor F で移したのちに morphism を適用しても、 functor G で移した後に morphism を適用しても良いという可換性が得られる。式 3.11 の可換図は図 3.4 となる。

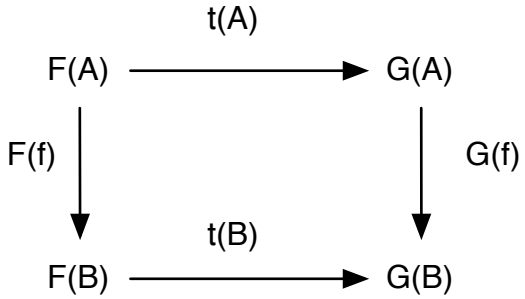


図 3.4: 自然変換の可換図 (式 3.11)

3.4 Monad

3.3 節では functor 間の可換性 natural transformation について述べた。
category, functor, natural transformation を用いて Monad を定義する。
category C における monad とは以下の性質を持つ $triple(T, \eta, \mu)$ である。

- $T : C \to C$

  category C から C への functor T

- $\eta : id_C \to T$

  C から T への natural transformation η

- $\mu : T^2 \to T$

  T^2 から T への natural transformation μ

  なお、T^2 とは TT と同義であるとする。つまり functor T による mapping を 2 回行なうものである。T の数が n である時 T^n と記述する。

この $triple(T, \eta, \mu)$ が図 3.5 の可換図を満たす。なお、A に対する μ を μ_A と記述する。同じように A に対する η を η_A と記述する

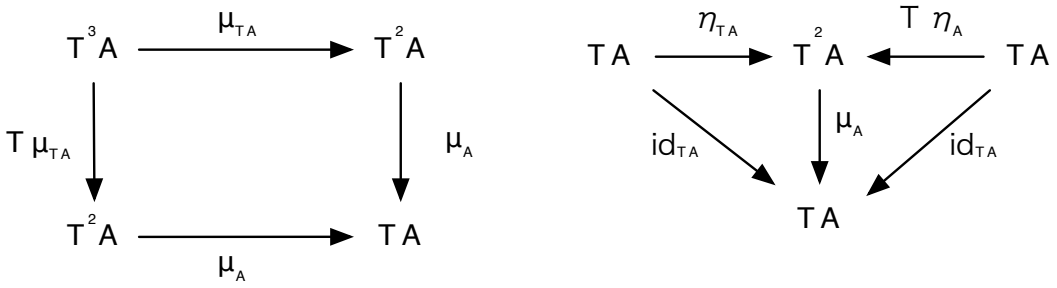


図 3.5: $triple(T, \eta, \mu)$ が Monad であるために満たすべき可換図

図 3.5 で示した可換図は monad が満たすべき Monad 則であり、T に対する演算の可換性を強制する。η は T を 1 つ増やすオペレーションであり、μ は T を 2 つから 1 つに減らすオペレーションである。

左側の可換図は T が 3 つある場合の可換性である。TTT に対して、(TT)T のよう外側の 2 つの T に μ を行なってから結果の TT に対して μ を行なっても、T(TT) のように内側の 2 つの T に μ を行なってから結果の TT に対して μ を行なっても可換であることを示す。つまり、TTT を減らす演算は左から行なっても右から行なっても良いことを示す。

そして右側の可換図は T が 1 つである場合の可換性である。T A に対して η を行ない TT A としてから μ を行なっても、T A の内部の A に対して η を行ない TT A としてか μ を行なっても、T A の id と同じであることを示している。つまり、T に対する演算における左右の単位元となるような性質を $triple(T, \eta, \mu)$ に強制することとなる。

第4章　Monads in Functional Programming

第3章では category による Monad の定義を述べた。第4章ではプログラミングにおける Monad について述べる。プログラムがなす category と functor, natural transformation を定義し、Monad を用いたデータとメタ計算との対応について述べる。

4.1　Category

3.4節では Monad の定義について述べた。4.1節はプログラムと category の対応について述べる。

プログラムは値と関数のみで表され、型付けられる。変数 x が型 A を持つ時、式4.1のように記述する。

$$x : A \tag{4.1}$$

関数は値を受けとり値を返すものとする。A を取り B を返す関数 f の型は式4.2のように記述する。

$$f : A \to B \tag{4.2}$$

そして、ある関数 f の引数の型と、ある関数 g の返り値の型が等しいとき f と g は合成できる。関数合成の記号には $\circ$ を用いる。例えば、 A を取り B を返す関数 f と、 B を取り C を返す関数 g の合成は式4.3のようになる。

$$\begin{aligned} f &: A \to B \\ g &: B \to C \\ g \circ f &: A \to C \end{aligned} \tag{4.3}$$

この時、型を object とし、関数を morphism とする category が構成できる。

この category が category が満たすべき法則を満たしているか確認する。

- 全ての object について identity mapping が存在する

  任意の型 A に対し $A \to A$ である関数 id が定義できれば identitiy mapping が存在することになる。任意の型の値 x を受けとり、その値をそのまま返す関数が id となる。

- 同じ obejct が domain と codomain になっている 2 つの morphism は合成することができ、合成の順番は結果に影響しない。

  morphism は関数である。つまり domain は引数の型であり、 codomain は返り値の型となる。morphism の合成は関数合成に相当し、合成の順序によらず引数と返り値の型は同じとなる。

プログラムに対応する category が構成できた。特に例として用いているプログラミング言語 Haskell では値と関数は型を持つため、 category との対応が分かりやすい。よって例題には Haskell のプログラムを用いることとする。

4.2 Functor

4.1 節ではプログラムと category が対応していることを述べた。4.2 節ではプログラムにおける functor について述べる。

プログラムにおける functor は型引数を持つことのできるデータ型に対応する。型引数を持つデータ型とは、任意のデータ型に対して構成可能なデータ構造であり、List などが相当する。List を型を受けとり型を返す型であると考えると、渡す型が引数のように振る舞うため型引数と呼ばれる。

$$\begin{aligned} A &: Type \\ ListA &: Type \\ List &: Type \to Type \end{aligned} \tag{4.4}$$

つまり、型と関数から構成される category から List 型と List に対応する関数からなる category へと置きかえるような functor が存在すれば良い。

Haskell では functor はリスト 4.1 のように型クラスとして提供される。

リスト 4.1: Haskell における Functor の定義

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

functor であることを保証したい型は f として表される。型クラスの定義から、f において fmap 関数を定義することで Functor となる。fmap は型 a から型 b への関数を取り、

f a を取り f b を返す。つまり、f でない型への演算を f の型においても適用するために変換する関数である。プログラムにおいて morpshim は関数であるため、fmap は $A \to B$ の morphism を $FA \to FB$ へと mapping する役割を担っている。なお、 fmap の型を `fmap :: (a -> b) -> ((f a) -> (f b))` と読むことで、関数を受けとり f における関数に変換していることが分かりやすくなる。

また、fmap の型に fa が存在するように、 f は型を引数として受けとっている。ここで object は型であるため、 A の object を $F(A)$ への mapping する役割を f が担う。よって型引数を持つ型 f と対応する fmap を定義することにより functor が定義できる。

functor の例として、型が Int である変数 x と Int から Bool を返す even 関数を考える。このプログラムがなす category C は object が Int であり、 morphism が even となる。category C を functor によって別の category に写すことができる。

例えば List がなす category D がある。

まず Haskell においてデータ型 List を定義する。空の List は Nil とし、List a に対して a の値を Cons で追加することで List を構築する。

ここで List が Functor とするために、 fmap を考える。fmap は a を取り b を返す関数を取り、List a を取って List b を返す関数である。よって、関数を取って List の全ての要素に適用することで実現できる。

定義した結果がリスト 4.2 である。

リスト 4.2: Haskell における List の例

```
1 data List a = Nil | Cons a (List a) deriving (Show)
2
3 instance Functor List where
4     fmap f Nil = Nil
5     fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

Int 型を持つ値 x と、Int から Bool 返す関数 even を考える。even を x に適用すると Bool となる。

この際、x を持つ List の型は List Int であり、 fmap によって even を List Int に適用すると List Bool となる。Haskell における実行結果はリスト 4.3 のようになる。

x と even からなるプログラムから、型 List と fmap を用いることにより List におけるプログラムでも同じように Bool が得られる。これを通常のプログラムから List のプログラムへの functor とみなす。

このように、型引数を持つ型と fmap による関数の変換を定義することによってプログラムにおける functor を実現する。可換図で表現すると図 4.1 となる。

functor の定義にあたり、3.2 節で示したように Functor 則を満たすようにデータ型と fmap を定義しなくてはならない。

Haskell における Functor 則はリスト 4.4 のように表される [6]。

1 行目が id の保存に、2 行目が関数の合成の保存に対応している。なお、 Haskell における関数合成は中置関数 . により行なわれる。

リスト 4.3: Haskell における List の実行例

```
*Main> let x = 100 :: Int
*Main> let even = even :: Int -> Bool
*Main> :type x
x :: Int
*Main> :type even
even :: Int -> Bool
*Main> :type (even x)
(even x) :: Bool

*Main> let list = Cons x Nil
*Main> :type list
list :: List Int
*Main> :type fmap even list
fmap even list :: List Bool
```

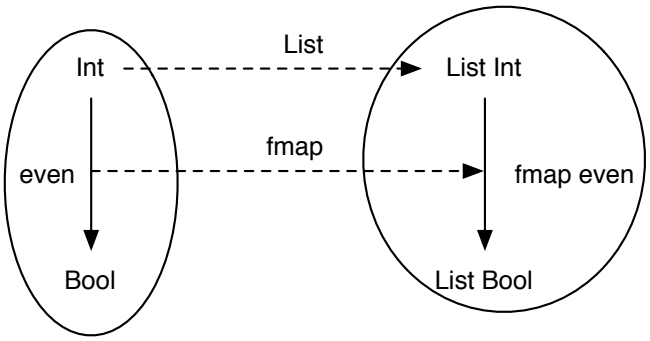


図 4.1: Haskell における Functor の例がなす可換図

4.3 Natural Transformation

4.2 節ではプログラムにおける functor を定義した。4.3 節ではプログラムにおける natural transformation について述べる。

natural transformation は functor と functor 間の変換である。プログラムにおいて functor は型引数を持つデータ構造であったため、データ構造からデータ構造への変換となる。

natural transformation は図 3.4 で示したような可換図が可換になるような t であった。つまりある functor f と g があった時に、 f において morphism を適用して t を適用しても、t を適用してから g において morphism を適用しても結果が変わらないような t を提供できれば良い。

プログラムにおける natural transformation t は図 3.4 を満たすような関数として表現される。Haskell において特定の関数が持つ性質を制約として記述することはできないため、natural transformation の具体的な定義は存在しない。

List における natural transformation の例としては、List の先頭を除いた List を返す tail 関数がある (4.5)。tail 関数を Functor List から Functor List への natural transformation とみなすのである。

リスト 4.4: Haskell における Functor 則

```
fmap id      = id
fmap (p . q) = (fmap p) . (fmap q)
```

リスト 4.5: List の先頭を除いた List を返す tail 関数

```
tail :: List a -> List a
tail Nil         = Nil
tail (Cons x xs) = xs
```

100, 200, 300 の数値を持つ List Int を考える。この List が functor f に相当する。Int を取り Bool を返す関数 even を fmap により適用すると List Bool が得られる。

ここで natural transformation tail を考える。tail は List a から List a への関数であるため、 functor f,g は両方とも List である。natural transformation の定義から、 tail を行なってから fmap even しても、 fmap even を行なってら tail を行なっても結果が同じであれば良い。実行した結果がリスト 4.6 である。

リスト 4.6: tail の実行例

```
*Main> :type Main.tail
Main.tail :: List a -> List a
*Main> let list = Cons 100 (Cons 200 (Cons 300 Nil)) :: List Int
*Main> :type list
list :: List Int
*Main> :type even :: Int -> Bool
even :: Int -> Bool :: Int -> Bool

*Main> :type (fmap even list)
fmap even list :: List Bool
*Main> :type Main.tail (fmap even list)
Main.tail (fmap even list) :: List Bool

*Main> :type (Main.tail list)
Main.tail list :: List Int
*Main> :type fmap even (Main.tail list)
fmap even (Main.tail list) :: List Bool

*Main> Main.tail (fmap even list)
Cons True (Cons True Nil)
*Main> fmap even (Main.tail list)
Cons True (Cons True Nil)
```

tail (fmap even list) と fmap even (tail list) の型が同じように List Bool であり、値も等しいことが分かる。なお、 tail は同一名の関数が既に定義されているため、 Main.tail として名前を指定している。List の値を変換してから先頭を除いても、List の先頭を除いてから値を変換しても結果が同じであることは直感的にも正しいように思える。

これがプログラムにおける natural transformation である。

図で表すと、図 4.2 のような可換図となり、これは可換である。

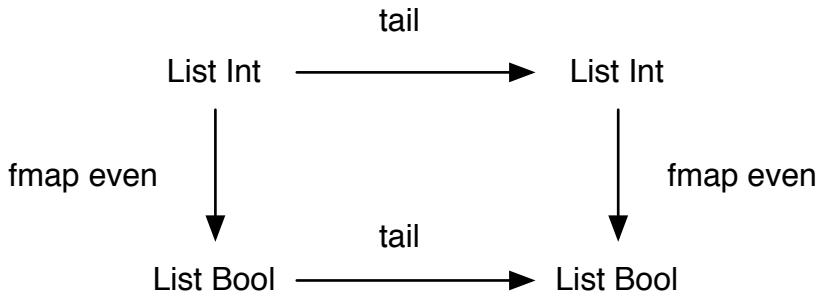


図 4.2: natural transformation tail の可換図

4.4 Monad

4.3 節ではプログラムにおける natural transformation について述べた。4.4 節ではプログラムにおける Monad について述べる。

Monad とは 図 3.5 の可換図を満たす $triple(T, \eta, \mu)$ であった。T は functor であり、η は $A \to TA$ 、 μ は $T^2A \to T$ の natural transformation であった。

Haskell において functor は型引数を持つ型と fmap で表現され、natural transformation は図 3.4 の可換図を満たす関数であった。つまり、 triple を表現するには 型 T と fmap、η と μ の定義することになる。

Haskell において、 η と μ の型は以下のようになる。

- eta : `A -> T A`

  ここで、 T は functor である型引数を持つ型

- mu : `T (T A) -> T A`

  本来の μ は $T^2 \to T$ であるため、 T によって 2 度 mapping された値から T によって mapping された値への関数になる。

Monad 則により、T に対する演算は単位元のように振る舞わせることと、演算の可換性を持つ。ここでメタ計算を T に対する演算として定義する。そうすることで、型 A を持つ値に対する計算から functor T により T の計算へと変換し、メタ計算部分は T に対する演算として行なうことで任意の A に対応するメタ計算を行なうことができるようになる。これがプログラムにおける monad を通したデータ型とメタ計算の対応である。そして、 Monad 則はメタ計算を T に対して η と μ で定義する際に満たすべき制約として機能する。

Haskell において List は monad としても振る舞う。List は nondeterminism (非決定性) を表現するデータ構造である。List に対する演算を行なう際、結果としてありうる値を全て列挙することにより非決定性を表現しようとするものである。

List において非決定性を表現する時、η は値から長さ 1 の List を構築する関数、 μ は List の List から List を返す concat 関数となる。

この η と μ を用いて非決定性を表現する。List a はありうる値の集合である。関数は値を取り、値から取りうる値を List として返す。この関数を List a に対して fmap することで、ありうる値に対して全ての取りうる値を計算することができる。この際、 List a に対して a から List a を返す関数を適用するために List a を持つ List が構築される。ネストされた List を、全ての取りうる値として通常の List に戻すために concat を用いる。concat することで、値と関数から全ての取りうる値が計算できた。

ここで concat は ネストした List から List への関数であるため、 $TTA \to TA$ に相当する μ となっている。η は長さ 1 の List を返すため、 $A \to TA$ となっている。このように、非決定性演算は List データ型と η と μ として Monad を用いて定義できる。

Haskell における monad の型クラスを振り返る (リスト 2.2)。Haskell においては monad は Monad 型クラスとして提供されているが、η と μ による記述はされていない。これは、Haskell がメタ計算との対応として Kleisli Category の triple を採用しているからである。monad と Kleisli triple は互いに同型であるために相互に変換することができる。また、 category や program の文脈が変わることで η を unit と呼んだり、 μ を join と呼んだり、 >>= を bind と呼んだりする。しかし今までの解説には η と μ を用いているために、このまま η と μ で解説していくこととする。

なお、monad と Kleisli triple との変換は Haskell においてはリスト 4.7 のようになる。

リスト 4.7: Haskell における monad と Kleisli triple との変換

```
1 return x = eta x
2 x >>= f  = mu (fmap f x)
3
4 eta x = return x
5 mu  x = (>>= id)
```

では List を用いて非決定性を表現した例を示す。

まずは List を Monad 型クラスのインスタンスとする (リスト 4.8)。

先程述べたように、 η は値を List とする関数、 μ は List を内包する List を全て結合する関数である。

この List Monad を実行した例を述べる。まずはありうる値のリストとして 100, 200, 400 を持つ List の x を考える。次に、値から取りうる計算結果を返す関数として f を定義する。取りうる計算結果として、1 加算した結果、10 加算した結果、2 乗した結果が存在するとし、結果は List として返す。

この x と f から全ての取りうる値を計算した結果がリスト 4.9 である。

3 つのありうる値に対して、取りうる 3 つの計算結果から生成された 9 つの値が全て計算されている。このように、ある性質を持つデータ型と、そのデータ型を返す関数の演算によって通常の計算に加えてメタ計算を実現することができた。

リスト 4.8: Haskell における List に対する monad の定義

```
append :: List a -> List a -> List a
append Nil xs          = xs
append (Cons x xs) xss = Cons x (append xs xss)

concat :: List (List a) -> List a
concat Nil         = Nil
concat (Cons x xs) = append x (Main.concat xs)

eta :: a -> List a
eta x = Cons x Nil

mu :: List (List a) -> List a
mu = Main.concat

instance Monad List where
    return x  = eta x
    li >>= f  = mu (fmap f li)
```

リスト 4.9: Haskell において List を monad として実行した例

```
*Main> let x = Cons 100 (Cons 200 (Cons 400 Nil))
*Main> let f = \x -> Cons (x+1) (Cons (x+10) (Cons (x*x) Nil))
*Main> x >>= f
Cons 101 (Cons 110 (Cons 10000 (Cons 201 (Cons 210 (Cons 40000 (Cons 401
   (Cons 410 (Cons 160000 Nil))))))))
```

これがプログラムにおける Monad であり、結果としてメタ計算とデータ型の対応が得られる。

なお、 Haskell においても Monad 則は存在し、リスト 4.10 の性質を満たすよう η や μ を定義しなくてはならない。

リスト 4.10: Haskell における Monad 則

```
mu . fmap mu  = mu . mu
mu . fmap eta = mu . eta = id

eta . f            = fmap f . eta
mu . fmap (fmap f) = fmap f . mu
```

1 つめの則が図 3.5 における左側の可換図に対応し、T に対する演算の可換性を示す。2 つめの則が図 3.5 における右側の可換図に対応し、T に対する演算における単位元のような振舞いを強制する。3 つめの則が eta に対する natural transformation の要請であり、4 つめの則が mu に対する natural transformation の要請である。

第5章　証明支援系言語 Agda による証明手法

第3章では functor, natural transformation, monad の定義を行ない、第4章では functional programming における対応を述べた。その中で、 Functor 則や Monad 則といった満たすべき性質がいくつか存在した。functional programming において、あるデータ型と対応する計算が Functor 則を満たすかの保証は言語の実装に依存している。例えば、 Haskell において functor は型クラスによって提供されるため、型レベルのチェックを行なう。そのため、型チェックは通るが Functor 則を満たさない functor が定義できてしまう。よって Haskell において Delta Monad を定義しただけでは Delta Monad が Monad であるかの証明が存在しない。そこで証明支援系言語 Agda を用いて、 Delta が Functor 則と Monad 則を満たすことを証明する。

第5章は Agda における証明手法について述べる。

5.1　Natural Deduction

証明には natural deduction(自然演繹) を用いる。natural deduction は Gentzen によって作られた論理と、その証明システムである [5]。命題変数と記号を用いた論理式で論理を記述し、推論規則により変形することで求める論理式を導く。

natural deduction において

$$\begin{array}{c} \vdots \\ A \end{array} \tag{5.1}$$

と書いた時、最終的に命題 A を証明したことを意味する。証明は木構造で表わされ、葉の命題は仮定となる。仮定には dead か alive の2つの状態が存在する。

$$\begin{array}{c} A \\ \vdots \\ B \end{array} \tag{5.2}$$

式 5.2 のように A を仮定して B を導くことができたとする。この時 A は alive な仮定であり、証明された B は A の仮定に依存していることを意味する。

ここで、推論規則により記号 $\Rightarrow$ を導入する。

$$\dfrac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \Rightarrow B} \Rightarrow \mathcal{I}$$

$\Rightarrow \mathcal{I}$ を適用することで仮定 A は dead となり、新たな命題 $A \Rightarrow B$ を導くことができる。A という仮定に依存して B を導く証明から、「A が存在すれば B が存在する」という証明を導いたこととなる。このように、仮定から始めて最終的に全ての仮定を dead とすることで、仮定に依存しない証明を導ける。なお、dead な仮定は `[A]` のように `[]` で囲んで書く。

alive な仮定を dead にすることができるのは $\Rightarrow \mathcal{I}$ 規則のみである。それを踏まえ、natural deduction には以下のような規則が存在する。

- Hypothesis

  仮定。葉にある式が仮定となるため、論理式 A を仮定する場合に以下のように書く。

  A

- Introductions

  導入。証明された論理式に対して記号を導入することで新たな証明を導く。

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B} \wedge \mathcal{I}$$

$$\dfrac{\begin{array}{c}\vdots \\ A\end{array}}{A \vee B} \vee 1\mathcal{I}$$

$$\dfrac{\begin{array}{c}\vdots \\ B\end{array}}{A \vee B} \vee 2\mathcal{I}$$

$$\dfrac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \Rightarrow B} \Rightarrow \mathcal{I}$$

- Eliminations

  除去。ある論理記号で構成された証明から別の証明を導く。

$$\frac{\begin{array}{c}\vdots\\ A \wedge B\end{array}}{A} \wedge 1\mathcal{E}$$

$$\frac{\begin{array}{c}\vdots\\ A \wedge B\end{array}}{B} \wedge 2\mathcal{E}$$

$$\frac{\begin{array}{ccc} & [A] & [B] \\ \vdots & \vdots & \vdots \\ A \vee B & C & C \end{array}}{C} \vee\mathcal{E}$$

$$\frac{\begin{array}{cc} \vdots & \vdots \\ A & A \Rightarrow B \end{array}}{B} \Rightarrow \mathcal{E}$$

記号 $\vee, \wedge, \Rightarrow$ の導入の除去規則について述べた。natural deduction には他にも $\forall, \exists, \bot$ といった記号が存在するが、ここでは解説を省略する。

それぞれの記号は以下のような意味を持つ

- $\wedge$ conjunction。2 つの命題が成り立つことを示す。$A \wedge B$ と記述すると A かつ B と考えることができる。
- $\vee$ disjunction。2 つの命題のうちどちらかが成り立つことを示す。$A \vee B$ と記述すると A または B と考えることができる。
- $\Rightarrow$ implication。左側の命題が成り立つ時、右側の命題が成り立つことを示す。$A \Rightarrow B$ と記述すると A ならば B と考えることができる。

例として、natural deduction で三段論法を証明する。なお、三段論法とは「A は B であり、 B は C である。よって A は C である」といった文を示す。

$$\frac{\dfrac{\dfrac{[A]_{(1)} \quad \dfrac{[(A \Rightarrow B) \wedge (B \Rightarrow C)]_{(2)}}{(A \Rightarrow B)} \wedge 1\mathcal{E}}{B} \quad \dfrac{[(A \Rightarrow B) \wedge (B \Rightarrow C)]_{(2)}}{(B \Rightarrow C)} \wedge 2\mathcal{E}}{C}}{\dfrac{A \Rightarrow C}{((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)} \Rightarrow \mathcal{I}_{(2)}} \Rightarrow \mathcal{I}_{(1)}$$

まず、三段論法を論理式で表す。

「A は B であり、 B は C である。よって A は C である」 が証明するべき命題である。まず、「A は B であり」から、A から性質 B が導けることが分かる。これが $A \Rightarrow B$ となる。次に、「B は C である」から、B から性質 C が導けることが分かる。これが $B \Rightarrow C$ となる。そしてこの 2 つは同時に成り立つ。よって $(A \Rightarrow B) \wedge (B \Rightarrow C)$ が仮定となる。この仮定が成り立つ時に「A は C である」を示せば良い。仮定と同じように「A は C である」は、 $A \Rightarrow C$ と書けるため、証明するべき論理式は $((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$ となる。

証明の手順はこうである。まず条件 $(A \Rightarrow B) \wedge (B \Rightarrow C)$ と A の 2 つを仮定する。条件を $\wedge 1\mathcal{E}$ $\wedge 2\mathcal{E}$ により分解する。A と $A \Rightarrow B$ から B を、 B と $B \Rightarrow C$ から C を導く。ここで $\Rightarrow \mathcal{I}$ により $A \Rightarrow C$ を導く。この際に dead にする仮定は A である。数回仮定を dead にする際は $_{(1)}$ のように対応する [] の記号に数値を付ける。これで残る alive な仮定は $(A \Rightarrow B) \wedge (B \Rightarrow C)$ となり、これから $A \Rightarrow C$ を導くことができたためにさらに $\Rightarrow \mathcal{I}$ を適用する。結果、証明すべき論理式 $((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$ が導けたために証明終了となる。

5.2 Curry-Howard Isomorphism

5.1 節では natural deduction における証明手法について述べた。natural deduction における証明は Curry-Howard Isomorphism により型付き λ 計算と対応している。

λ 計算とは、計算モデルの 1 つであり、計算の実行を関数への適用としてモデル化したものである。関数は λ 式として表され、引数を 1 つ取りその引数を計算する関数は式 5.3 のように記述される。

$$\lambda x.x \tag{5.3}$$

値と λ 式には型を付与することができ、その型の計算が natural deduction と対応している。

例えば命題 A が成り立つためには A という型を持つ値が存在すれば良い。しかしこの命題は A という alive な仮定に依存している。natural deduction では A の仮定を dead にするために $\Rightarrow \mathcal{I}$ により $\Rightarrow$ を導入する。これが λ による抽象化に対応している。

$$\begin{gathered} x : A \\ \lambda x.x : A \to A \end{gathered} \tag{5.4}$$

プログラムにおいて、変数 x は内部の値により型が決定される。特に、x の値が未定である場合は未定義の変数としてエラーが発生する。しかし、 x を取って x を返す関数は定義することはできる。これは natural deduction の $\Rightarrow \mathcal{I}$ により仮定を discharge することに相当する。

このように、 natural deduction における証明はそのまま 型付き λ 計算に変換することができる。

それぞれの詳細な対応は省略するが、表 5.1 のような対応が存在する。

	natural deduction	型付き λ 計算
hypothesis	A	型 A を持つ変数 x
conjunction	$A \wedge B$	型 A と型 B の直積型 を持つ変数 x
disjunction	$A \vee B$	型 A と型 B の直和型 を持つ変数 x
implication	$A \Rightarrow B$	型 A を取り型 B の変数を返す関数 f

表 5.1: natural deuction と 型付き λ 計算との対応

5.1 節でも行なった三段論法の証明を Haskell により行なうとリスト 5.1 となる。

リスト 5.1: Haskell における三段論法の証明

```
1 *Main> :type fst
2 fst :: (a, b) -> a
3 *Main> :type snd
4 snd :: (a, b) -> b
5
6 *Main> :type \cond -> (\a -> (snd cond ((fst cond) a)))
7 \cond -> (\a -> (snd cond ((fst cond) a)))
8   :: (t2 -> t1, t1 -> t) -> t2 -> t
```

証明は λ 式の型として表現される。Haskell における λ 式は `\x -> x` のように表される。`\` と `->` の間が引数であり、 `->` の後に処理を記述する。また、$\wedge$ に対応する直積型は tuple として提供される。A と B の直積型は (A, B) として表現される。そして、$\wedge 1\mathcal{E}$ に対応する処理は関数 fst 、 $\wedge 2\mathcal{E}$ に対応する処理は関数 snd として提供される。fst と snd が $\wedge$ の除去に対応しているのは型を見ることで分かる。

そして三段論法の証明の解説を行なう。三段論法の仮定は、$(A \Rightarrow B) \wedge (B \Rightarrow C)$ であった。この仮定を変数 cond として受けとる。cond に対して fst を行なうことで $(A \Rightarrow B)$ を、snd を行なうことで $(B \Rightarrow C)$ を証明する。ここでさらに仮定 a を導入し、$(A \Rightarrow B)$ と $(B \Rightarrow C)$ に適用することで C を得る。

結果、`\cond -> (\a -> (snd cond ((fst cond) a)))` のような式が得られ、この型を確認すると正しく三段論法の証明となっている。なお、型推論の際に A は t2 に、 B は t1 に、 C は t という型名になっている。

5.3 Agda による証明

5.2 節において型を用いて証明を行なう手法を述べた。しかし、Haskell における型システムは証明を記述するために用いるものではない。よって、依存型を持つ証明支援言語 Agda を用いて証明を記述する。

依存型とは値に依存した型を作ることができる型であり、非常に表現能力が高い。値に型を使うことができるために証明に基づいた証明を記述することができる。

$\Rightarrow \mathcal{E}$ に対応する関数適用を Agda で記述するとリスト 5.2 のようになる。

リスト 5.2: Agda における $\Rightarrow \mathcal{E}$

```
1 module apply_function where
2
3 postulate A : Set
4 postulate f : {B : Set} -> A -> B
5
6 ->E : {B : Set} -> A -> (A -> B) -> B
7 ->E a f = f a
```

Agda による型注釈は `term : type` と記述する。postulate とすると alive な証明を記述することができる。まずは alive な A を仮定している。なお、Set 型は型の型として組込みで用意されている型である。そのため詳細は省略するが、 A は型であると考える。

次に関数 f を仮定する。仮定に用いられている記号 `{}` の内部の式は implicit な paramter である。任意の B の対してこの型が成り立つことを記述している。implicit な parameter は可能であれば Agda が推論する。そのため、f の見掛け上の型は `A -> B` のようになる。ここで、 B に明示的に値を代入することで f の型を変更することもできる。このように Agda では型における引数のようなものが存在し、型の表現能力が高い。

A と f の 2 つを仮定したところで、証明である `->E` を定義する。証明するべき式は `->E` の型として与えられ、証明は `->E` の式として記述する。式の型に対する定義を正しく行なうことで証明を与える。

$\Rightarrow \mathcal{E}$ に対応する `->E` は関数の適用なので、値 a と関数 f を受けとって適用することで B が得られる。なお、仮定を alive のまま証明を記述するのは依存した仮定を残すため、必要な仮定を引数として受けとったり、implicit な parameter として指定して証明するのが良い。

また、Agda は証明に用いる規則なども Agda 内部で定義することができる。例えば、直積型の定義はリスト 5.3 のようなものがある。

関数の定義時に _ とした部分には項を入れることで関数が適用される。つまり _×_ とすれば中置関数を作成することができる。データ型 _×_ は型 A と B を持つ型である。データ型 _×_ はコンストラクタ `<_,_>` により構成され、コンストラクタは A と B を取ることで $A \times B$ 型を返す。

例えば型 A と B から 直積型 $A \times B$ が作成できる証明は constructProduct である。任意の型 A と B を受けとり、コンストラクタ `<_,_>`を用いて $A \times B$ に相当する値を返す。

リスト 5.3: Agda における直積型の例

```
1  module product where
2
3  data _×_ (A B : Set) : Set where
4    <_,_> : A → B → A × B
5
6  constructProduct : {A B : Set} → A → B → A × B
7  constructProduct a b = < a , b >
8
9  patternMatchProduct : {A B : Set} → A × B → B
10 patternMatchProduct (< a , b >) = b
```

また、コンストラクタは引数にてパターンマッチにより分解することができる。その例が patternMatchProduct である。受けとる引数の型は $A \times B$ であるため、ありえるコンストラクタは <_,_>のみである。よって引数を取る際に < a , b > のように対応するコンストラクタと変数を用意することで、コンストラクタに基づいて値を分解することができる。$A \times B$ 型から B を取るために、$A \times B$ 型の引数を変数にを直接束縛せずにコンストラクタで分解し、a と b に束縛することで A と B を得る。b を返すことで型 B の値を返すことができる。そのため、 patternMathProduct は $\wedge 2\mathcal{E}$ そのものである。

5.4 Reasoning

5.3 節では Agda における証明の手法について解説した。5.4 節では Agda における証明の例として、自然数の加法の交換法則を証明する。

まずは自然数を定義する。今回用いる自然数の定義は以下のようなものである。

- 0 は自然数である
- 任意の自然数には後続数が存在する
- 0 はいかなる自然数の後続数でもない
- 異なる自然数どうしの後続数は異なる $(n \neq m \rightarrow Sn \neq Sm)$
- 0 がある性質を満たし、a がある性質を満たせばその後続数 S(n) も自然数である

この定義は peano arithmetic における自然数の定義である。
Agda で自然数型 Nat を定義するとリスト 5.4 のようになる。
自然数型 Nat は 2 つのコンストラクタを持つ。

- O

  引数を持たないコンストラクタ。これが 0 に相当する。

リスト 5.4: Agda における自然数型 Nat の定義

```
1 module nat where
2
3 data Nat : Set where
4   O : Nat
5   S : Nat -> Nat
```

- S

  Nat を引数に取るコンストラクタ。これが後続数に相当する。

よって、数値の 3 は S (S (S O)) のように表現される。S の個数が数値に対応する。
次に加算を定義する (リスト 5.5)。

リスト 5.5: Agda における自然数型に対する加算の定義

```
1 open import nat
2 module nat_add where
3
4 _+_ : Nat -> Nat -> Nat
5 O + m     = m
6 (S n) + m = S (n + m)
```

加算は中置関数 _+_ として定義する。2 つの Nat を取り、Nat を返す。関数 _+_ はパターンマッチにより処理を変える。0 に対して m 加算する場合は m であり、 n の後続数に対して m 加算する場合は n に m 加算した数の後続数とする。S を左の数から右の数へ 1 つずつ再帰的に移していくような加算である。

例えば 3 + 1 といった計算は (S (S (S O))) + (S O) のように記述される。ここで 3 + 1 が 4 と等しいことの証明を行なう。

等式の証明には agda の standard library における Relation.Binary.Core の定義を用いる。

リスト 5.6: Relation.Binary.Core による等式を示す型 ≡

```
1 data _≡_ {a} {A : Set a} (x : A) : A → Set a where
2   refl : x ≡ x
```

Agda において等式は、等式を示すデータ型 ≡ により定義される。≡ は同じ両辺が同じ項に簡約される時にコンストラクタ refl で構築できる。

実際に 3 + 1 = 4 の証明は refl で構成できる (リスト 5.7)。

3+1 という関数を定義し、その型として証明すべき式を記述し、証明を関数の定義として定義する。証明する式は (S (S (S O))) + (S O) ≡ (S (S (S (S O)))) である。今回は _+_ 関数の定義により (S (S (S (S O)))) に簡約されるためにコンストラクタ refl が証明となる。

≡ によって証明する際、必ず同じ式に簡約されるとは限らないため、いくつかの操作が Relation.Binary.PropositionalEquality に定義されている。

リスト 5.7: Agda における 3 + 1 の結果が 4 と等しい証明

```
1 open import Relation.Binary.PropositionalEquality
2 open import nat
3 open import nat_add
4
5 module three_plus_one where
6
7 3+1 : (S (S (S O)))  +  (S O) ≡ (S (S (S (S O))))
8 3+1 = refl
```

- sym : $x \equiv y \to y \equiv x$

  等式が証明できればその等式の左辺と右辺を反転しても等しい。

- cong : $f \to x \equiv y \to fx \equiv fy$

  証明した等式に同じ関数を与えても等式は保たれる。

- trans : $x \equiv y \to y \equiv z \to x \equiv z$

  2 つの等式に表れた同じ項を用いて 2 つの等式を繋げた等式は等しい。

ではこれから nat の加法の交換法則を証明していく (リスト 5.8)。

リスト 5.8: Agda における加法の交換法則の証明

```
1  open import Relation.Binary.PropositionalEquality
2  open import nat
3  open import nat_add
4  open ≡-Reasoning
5
6  module nat_add_sym where
7
8  addSym : (n m : Nat) -> n + m ≡ m + n
9  addSym O       O   = refl
10 addSym O    (S m)  = cong S (addSym O m)
11 addSym (S n)   O   = cong S (addSym n O)
12 addSym (S n) (S m) = {!!}
```

証明する式は $n + m \equiv m + n$ である。n と m は Nat であるため、それぞれがコンストラクタ O か S により構成される。そのためにパターンは 4 通りある。

- n = O, m = O

  + の定義により、 O に簡約されるため refl で証明できる。

- n = O, m = S m

  $O + (Sm) \equiv (Sm) + O$ を証明することになる。この等式は + の定義により $O + (Sm) \equiv S(m + O)$ と変形できる。$S(m + O)$ は $m + O$ に S を加えたものであるため、 cong を用いて再帰的に `addSym` を実行することで証明できる。

この2つの証明はこのような意味を持つ。n が 0 であるとき、 m も 0 なら簡約により等式が成立する。n が 0 であり、 m が 0 でないとき、 m は後続数である。よって m が (S x) と書かれる時、 x は m の前の値である。前の値による交換法則を用いてからその結果の後続数も + の定義により等しい。

ここで、 `addSym` に渡される m は 1 つ値が減っているため、最終的には n = 0, m = 0 である refl にまで簡約され、等式が得られる。

- n = S n, m = O

  $(Sn) + O \equiv O + (Sn)$ を証明する。この等式は + の定義により $S(n + O) \equiv (Sn)$ と変形できる。さらに変形すれば $S(n + O) \equiv S(O + n)$ となる。よって `addSym` により O と n を変換した後に cong で S を加えることで証明ができる。

  ここで、 $O + n \equiv n$ は + の定義により自明であるが、$n + O \equiv n$ をそのまま導出できないことに注意して欲しい。+ の定義は左側の項から S を右側の項への移すだけであるため、右側の項への演算はしない。

- n = S n, m = S m

  3 つのパターンは証明したが、このパターンは少々長くなるため別に解説することとする。

3 つのパターンにおいては refl や cong といった単純な項で証明を行なうことができた。しかし長い証明になると refl や cong といった式を trans で大量に繋げていく必要性がある。長い証明を分かりやすく記述するために ≡-Reasoning を用いる。

≡-Reasoning では等式の左辺を begin の後に記述し、等式の変形を $\equiv \langle expression \rangle$ に記述することで変形していく。最終的に等式の左辺を ■ の項へと変形することで等式の証明が得られる。

自然数の加法の交換法則を ≡-Reasoning を用いて証明した例がリスト 5.9 である。特に n と m が 1 以上である時の証明に注目する。

まず `(S n) + (S m)` は + の定義により `S (n + (S m))` に等しい。よって refl で導かれる。なお、基本的に定義などで同じ項に簡約される時は refl によって記述することが多い。

次に `S (n + (S m))` に対して addSym を用いて交換し、 cong によって S を追加することで `S ((S m) + n)` を得る。これは、前 3 パターンにおいて + の右辺の項が 1 以上であっても上手く交換法則が定義できたことを利用して項を変形している。このように同じ法則の中でも、違うパターンで証明できた部分を用いて別パターンの証明を行なうこともある。

最後に `S ((S m) + n)` から `(S m) + (S n)` を得なくてはならない。しかし、 + の定義には右辺に対して S を移動する演算が含まれていない。よってこのままでは証明することができない。そのため、等式 $S(m + n) \equiv m + (Sn)$ を addToRight として定義する。

リスト 5.9: ≡ - Reasoning を用いた証明の例

```
1  open import Relation.Binary.PropositionalEquality
2  open import nat
3  open import nat_add
4  open ≡-Reasoning
5
6  module nat_add_sym_reasoning where
7
8  addToRight : (n m : Nat) → S (n + m) ≡ n + (S m)
9  addToRight O m     = refl
10 addToRight (S n) m = cong S (addToRight n m)
11
12 addSym : (n m : Nat) → n + m ≡ m + n
13 addSym O       O   = refl
14 addSym O    (S m)  = cong S (addSym O m)
15 addSym (S n)   O   = cong S (addSym n O)
16 addSym (S n) (S m) = begin
17   (S n) + (S m)  ≡⟨ refl ⟩
18   S (n + S m)    ≡⟨ cong S (addSym n (S m)) ⟩
19   S ((S m) + n)  ≡⟨ addToRight (S m) n ⟩
20   S (m + S n)    ≡⟨ refl ⟩
21   (S m) + (S n)  ■
```

addToRight の証明の解説は省略する。addToRight を用いることで `S ((S m) + n)` から `(S m) + (S n)` を得られた。これで等式 $(Sm)+(Sn) \equiv (Sn)+(Sm)$ の証明が完了した。

自然数に対する + の演算を考えた時にありえるコンストラクタの組み合せ 4 パターンのいずれかでも交換法則の等式が成り立つことが分かった。このように、Agda における等式の証明は、定義や等式を用いて右辺と左辺を同じ項に変形することで行なわれる。

第6章　Delta Monad が Monad である証明

第 5 章では Agda における証明手法について述べた。第 6 章では Agda を用いて Delta Monad が Monad であることを証明していく。証明のゴールは Delta Monad が Funcor 則と Monad 則を満たすことの証明である。なお、証明は全ての Delta が同じバージョンを持つという制約下で行なった。

6.1　Agda における Delta Monad の表現

6.1 節では Agda において Delta Monad を再定義する。Agda は Haskell において実装されているため、ほとんど同様に定義できる (リスト 6.1)。

リスト 6.1: Agda における Delta Monad のデータ定義

```
1 data Delta {l : Level} (A : Set l) : (Nat → (Set l)) where
2   mono    : A → Delta A (S O)
3   delta   : {n : Nat} → A → Delta A (S n) → Delta A (S (S n))
```

data 型 Delta は型 A の値と Nat を持つ。

level とは型の level の区別に用いられるものである。Agda では型も値として扱えるため、同じ式において型と値が混同することがある。厳密に型と値を区別したい場合、level を定義することで区別する。level は任意の level と、関数 suc により定義される。関数 suc は level を一つ上昇させる関数である。level l を適用した型を用いる時は Set l と記述する。今回は証明する対象のプログラムは Set l の level とし、それ以外は Set (suc l) の level とする。

Nat は自然数であり、プログラムのバージョンに対応する。自然数の定義は 5.4 節にあるリスト 5.4 にならうものとする。

data 型 Delta は 2 つのコンストラクタにより構成される。

- mono

  プログラムの最初の変更単位を受けとり、バージョン 1 とする。型 A を取り、バージョンが 1 の Delta を構成することでその表現とする。

- delta

  変更単位と変更単位の列を受けとり、変更単位を追加する。これは変更によるバージョンアップに相当する。よって任意の 1 以上のバージョンを持つ Delta に変更単位を加えることでバージョンを 1 上昇させる。

Agda においてもデータ型 Delta が定義できた。これからこの定義を用いて Functor 則と Monad 則を証明していく。

6.2 Agda における Functor 則

Agda における Functor 則はリスト 6.2 のように記述した。

リスト 6.2: Agda における Functor 則の定義

```
1  record Functor {l : Level} (F : Set l → Set l) : Set (suc l) where
2   field
3    fmap : {A B : Set l} → (A → B) → (F A) → (F B)
4   field -- laws
5    preserve-id : {A : Set l} (x : F A) → fmap id x ≡ id x
6    covariant   : {A B C : Set l} (f : A → B) → (g : B → C) → (x : F A)
7                    → fmap (g ∘ f) x ≡ ((fmap g) ∘ (fmap f)) x
8   field -- proof assistant
9    fmap-equiv : {A B : Set l} {f g : A → B} →
10     ((x : A) → f x ≡ g x) → (fx : F A) → fmap f fx ≡ fmap g fx
```

Agda ではある性質を持つデータは record 構文によって記述する。record の値は record の定義時に要請した field を全て満たすことで構築される。よって、あるデータ A が持つべき性質は field に定義し、A を用いた証明によって field を満たす。field を満たすことで record が構成できた場合、A がある性質を満たすことを Agda において証明したことになる。

record Functor は implicit な値 level と、型引数を持つ関数 F を持つ。record Functor が取る F は Set l を取り Set l を取る関数である。Set l が証明の対象となるプログラムであるため、関数 F はプログラムの level で表現する。よって、プログラムの level l を取り、変更せずに level l の Set を返すようにする。それに対し、 F に対する証明となる record Functor は、証明の対象とするプログラムでは無い。これは、証明の対象となるプログラムと証明そのものを混同しないために、 Functor の Level を suc により上げる。

record Functor の field には以下のようなものがある。

- fmap

  Functor に要請される、category から category への map 関数である。型の定義はほとんど Haskell と同じである。fmap は任意の型に対して適用可能でなくてはいけないため、map する対象の型 A と map される対象の型 B は implicit とする。

- preserve-id

  Functor 則の id の保存である。型 F A を持つ値 x に対する fmap id と id の等価性がなくてはならない。

- covariant

  Functor 則における関数の合成の保存である。関数 f と g を合成してから fmap により mapping しても、 f と g を個別に mapping してから合成しても等価性を持たなくてはならない。

- fmap-equiv

  ある型 A の値 x に対して等価である関数 f と g を考えた時、 型 F A の値 fx の対し、 fmap f としても fmap g としても等価であることを保証する。これは本来は Functor 則には存在しない。Agda において Monad の証明に必要であったために追加した。

6.3 Delta が Functor 則を満たす証明

6.2 節では Agda における Functor 則の表現について述べた。6.3 節では 6.1 節の Delta Monad の定義を用いて Functor 則を証明していく。

まず、Agda における delta に対する fmap の定義を示す (リスト 6.3)。

リスト 6.3: Agda における Delta に対する fmap の定義

```
1 delta-fmap : {l : Level} {A B : Set l} {n : Nat} ->
2              (A -> B) -> (Delta A (S n)) -> (Delta B (S n))
3 delta-fmap f (mono x)    = mono  (f x)
4 delta-fmap f (delta x d) = delta (f x) (delta-fmap f d)
```

バージョンが 1 以上の Delta について fmap を定義する。関数 f は型 A から B への関数とし、 Delta A から Delta B への関数に mapping する。コンストラクタ 2 つのパターンマッチを使って再帰的に f を delta の内部の値への適応することで fmap を行なう。

定義した データ型 Delta と関数 fmap が Functor 則を満たすか証明していく。なお、今回 Delta は全ての場合においてバージョンを 1 以上持つものとする。その制約は引数の Delta のバージョンに必ず S を付けることで行なう。

- fmap は id を保存する

  リスト 6.4 に証明を示す。

  コンストラクタにようてパターン分けをする mono の場合は fmap の定義により同じ項に変形されるために relf で証明できる。delta の場合、delta の 第一引数は mono の時のように同じ項に変形できる。しかし第二引数は fmap の定義により

リスト 6.4: Delta における fmap も id を保存する証明

```
delta-preserve-id :  {l : Level} {A : Set l} {n : Nat} →
    (d : Delta A (S n)) → (delta-fmap id) d ≡ id d
delta-preserve-id (mono x)    = refl
delta-preserve-id (delta x d) = cong (delta x) (delta-preserve-id d)
```

delta-fmap d id に変形される。見掛け上は等式の左辺と変わらないように見えるが、先頭 1 つめを除いているため、引数で受けたバージョンよりも 1 つだけ値が下がっている。よって最終的にバージョン 1 である mono になるまで再帰的に delta-preserve-id 関数を用いて変形した後に cong によって先頭 1 つめを適用し続けることで証明ができる。

- fmap は関数の合成を保存する

  リスト 6.5 に証明を示す。

リスト 6.5: Delta における fmap も関数の合成を保存する証明

```
delta-covariant : {l : Level} {n : Nat} {A B C : Set l} →
            (f : B → C) → (g : A → B) → (d : Delta A (S n)) →
            (delta-fmap (f ∘ g)) d ≡ ((delta-fmap f) ∘ (delta-fmap g)) d
delta-covariant f g (mono x)    = refl
delta-covariant f g (delta x d) = cong (delta (f (g x)))
                                       (delta-covariant f g d)
```

id の保存のように、コンストラクタによるパターンマッチを行なう。バージョンが 1 の場合は同じものに簡約され、1 より大きい場合は再帰的に変形することで証明できる。

Delta と fmap と 2 つの証明を用いて Functor record を構成する (リスト 6.6)。

リスト 6.6: Delta が Functor 則を満たすことの証明

```
delta-is-functor : {l : Level} {n : Nat} →
                    Functor {l} (\A → Delta A (S n))
delta-is-functor = record { fmap       = delta-fmap
                          ; preserve-id = delta-preserve-id
                          ; covariant  = \f g → delta-covariant g f
                          ; fmap-equiv = delta-fmap-equiv }
```

record が正しく構成できたため、Delta は Functor 則を満たす。Agda ではこのように法則とデータの関連付けを行なう。

6.4 Agda における Monad 則

6.2 節と 6.3 節では Delta が Functor 則を満たすことの証明を行なった。6.4 節では同じように Monad 則の定義を行ない、6.5 節で証明を行なう。

まずは Monad 則の定義を行なう (リスト 6.7)。

リスト 6.7: Agda における Monad 則の定義

```
record Monad {l : Level} (T : Set l → Set l) (F : Functor T)
        : Set (suc l)  where

  field -- category
    mu  :  {A : Set l} → T (T A) → T A
    eta :  {A : Set l} → A → T A

  field -- natural transformations
    eta-is-nt : {A B : Set l} → (f : A → B) → (x : A)       →
                (eta ∘ f) x ≡ fmap F f (eta x)
    mu-is-nt  : {A B : Set l} → (f : A → B) → (x : T (T A)) →
                mu (fmap F (fmap F f) x) ≡ fmap F f (mu x)

  field -- category laws
    association-law : {A : Set l} → (x : (T (T (T A)))) →
                      (mu ∘ (fmap F mu)) x ≡ (mu ∘ mu) x
    left-unity-law  : {A : Set l} → (x : T A)           →
                      (mu ∘ (fmap F eta)) x ≡ id x
    right-unity-law : {A : Set l} → (x : T A)           →
                      id x ≡ (mu ∘ eta) x
```

Monad 則とは $triple(T, \eta, \mu)$ に対して課すべき制約であった。そのためまず要素として T , η, μ が存在する。T は Set l を取り Set l を返す型であり、Functor 則を満たす。η は T を増やす関数であり、 μ は T を減らす関数である。これらが Monad の構成要素を定義する field である。

次に、η と μ は natural transformation である必要がある。よって field に制約として η と μ の natural transformation を定義する。なお、 field という予約語は複数書いても 2 つ目以降は無いものとして振る舞う。

ここで、 `fmap F` という項に注目して欲しい。値 F は Functor T 型を持ち、fmap や 2 つの証明を内包する。fmap や証明へのアクセスは `field-name record-value` と記述する。例えば、リスト 6.6 で定義した Functor Delta の型を持つ値 `delta-is-functor` に対して `fmap delta-is-functor` とすると関数 delta-fmap が得られる。つまり、`fmap F f` とするのは Functor 則が証明された F の fmap に関数 f を適用することとなる。

最後に $triple(T, \eta, \mu)$ に対する Monad 則に相当する等式を記述する。Monad 則は可換図として与えられた T に対する演算の可換性と、T に対する演算の単位元を強制するものであった。T に対する演算の可換性は association-law として、単位元の強制は unity-law として記述できる。unity-law は T に対する演算が右側と左側の二種類があるため、right-unity-law と left-unity-law に分割した。

これら全ての field を満たすような証明を記述できれば、 $triple(T, \eta, \mu)$ は Monad であると言える。

6.5 Delta が Monad 則を満たす証明

6.4 節において Agda における Monad 則の定義を行なった。6.5 節では Delta に対する η と μ の定義と Delta における *triple* が Monad であることの証明を行なう。これから証明する Delta を用いたプログラムは全ての値がバージョンを持ち、全ての関数はバージョンを持った値を返すものとする。さらに、バージョンはプログラム全体で 1 以上の値を持ち、プログラム内で統一されるものとする。

まず、リスト 2.3 で示した Delta Monad のメタ計算を Agda で再定義する (リスト 6.8)。Haskell での定義は Kleisli Trple による定義であるため、リスト 4.7 で示した Kleisli Triple と Monad の対応を用いて変形している。

リスト 6.8: Agda における Delta に対する Monad の定義

```
1  headDelta : {l : Level} {A : Set l} {n : Nat} → Delta A (S n) → A
2  headDelta (mono x)    = x
3  headDelta (delta x _) = x
4
5  tailDelta : {l : Level} {A : Set l} {n : Nat} →
6      Delta A (S (S n)) → Delta A (S n)
7  tailDelta (delta _ d) = d
8
9
10 delta-eta : {l : Level} {A : Set l} {n : Nat} → A → Delta A (S n)
11 delta-eta {n = O}     x = mono x
12 delta-eta {n = (S n)} x = delta x (delta-eta {n = n} x)
13
14 delta-mu : {l : Level} {A : Set l} {n : Nat} →
15            (Delta (Delta A (S n)) (S n)) → Delta A (S n)
16 delta-mu (mono x)    = x
17 delta-mu (delta x d) = delta (headDelta x) (delta-mu (delta-fmap
       tailDelta d))
```

η に相当する delta-eta はメタ計算における T を 1 つ増やす演算に相当する。内包する値のバージョンに依存して Delta が持つバージョンが決まるため、 Nat の値によりバージョン数を決める。なお、パターンマッチの項に存在する `{}` は、 implicit な値のパターンマッチである。例えば `{n = S x}` とすれば、 implicit な値 n は S により構成される時にこの定義が実行され、S を省いた残りの値は x に束縛される。

次に μ に相当する delta-mu を定義する。delta-mu は複数の `TT -> T` に対応するメタ計算であるため、 Delta Delta を受けとり、Delta を返す。これはバージョン管理された値に対し、バージョン管理された関数を適用した場合の値の選択に相当する。μ は値と関数が同じバージョンであるような計算結果を返す。例えば関数と値のバージョンがそれぞれ 5 である時、全ての組み合せは 25 パターンの中から 5 パターンを返す。

同じバージョンである値を選ぶため、先頭のバージョンの値を取る headDelta 関数と、先頭以外のバージョン列を取る tailDelta 関数を用いている。

Delta Monad に対応する $triple(T, \eta, \mu)$ が定義できた。これから Monad 則を満たすことを証明していく。

1. η が natural transformation であること

   まず、 η が natural transformation であることを示す (リスト 6.9)。eta は T を 1 つ増やす演算であるが、値のバージョンに応じて挙動を変える。よって n によりパターンマッチすることで証明も変更する。特に n が O である時は同じ項に簡約されるために refl で証明することができ、n が O 以上であれば再帰的に証明することができる。

   リスト 6.9: Agda における Delta の η が natural transformation である証明

```
1  delta-eta-is-nt : {l : Level} {A B : Set l} → {n : Nat}
2        (f : A → B) → (x : A) →
3        (delta-eta {n = n} ∘ f) x ≡ delta-fmap f (delta-eta x)
4  delta-eta-is-nt {n = O}   f x = refl
5  delta-eta-is-nt {n = S n} f x = begin
6    (delta-eta ∘ f) x                          ≡⟨ refl ⟩
7    delta-eta (f x)                            ≡⟨ refl ⟩
8    delta (f x) (delta-eta (f x))
9    ≡⟨ cong (\de → delta (f x) de) (delta-eta-is-nt f x) ⟩
10   delta (f x) (delta-fmap f (delta-eta x)) ≡⟨ refl ⟩
11   delta-fmap f (delta x (delta-eta x))     ≡⟨ refl ⟩
12   delta-fmap f (delta-eta x)               ■
```

2. μ が natural transformation であること

   次に μ が natural transformation であることを示す (リスト 6.10)。

   μ の証明もバージョンによるパターンマッチで行なわれる。バージョンが 1 である場合は refl で証明できるが、1 以上の場合に同じ証明で再帰的に定義できない。変形中に内部の Delta と外部の Delta のバージョン数が異なる項が存在し、バージョン数が等しい前提の μ に適用できないためである。その項に対する演算を含めた等式を別の証明 `tailDelta-to-tail-nt` として分離した。

リスト 6.10: Agda における Delta の μ が natural transformation である証明

```
tailDelta−to−tail−nt : {l : Level} {A B : Set l} (n m  : Nat)
                 (f : A → B) (d : Delta (Delta A (S (S m))) (S n)) →
     delta−fmap tailDelta (delta−fmap (delta−fmap f) d) ≡ delta−fmap (delta−fmap f) (delta−fmap tailDelta d)
tailDelta−to−tail−nt O O f (mono (delta x d)) = refl
tailDelta−to−tail−nt O (S m) f (mono (delta x d)) = refl
tailDelta−to−tail−nt (S n) O f (delta (delta x (mono xx)) d) = begin
  delta (mono (f xx)) (delta−fmap tailDelta (delta−fmap (delta−fmap f) d))
  ≡⟨ cong (\de → delta (mono (f xx)) de) (tailDelta−to−tail−nt n O f d) ⟩
  delta (mono (f xx)) (delta−fmap (delta−fmap f) (delta−fmap tailDelta d))
  ■
tailDelta−to−tail−nt (S n) (S m) f (delta (delta x (delta xx d)) ds) = begin
  delta (delta (f xx) (delta−fmap f d)) (delta−fmap tailDelta (delta−fmap (delta−fmap f) ds))
  ≡⟨ cong (\de → delta (delta (f xx) (delta−fmap f d)) de) (tailDelta−to−tail−nt n (S m) f ds) ⟩
  delta (delta (f xx) (delta−fmap f d)) (delta−fmap (delta−fmap f) (delta−fmap tailDelta ds))
  ■

delta−mu−is−nt : {l : Level} {A B : Set l} {n : Nat} → (f : A → B) → (d : Delta (Delta A (S n)) (S n))
           → delta−mu (delta−fmap (delta−fmap f) d) ≡ delta−fmap f (delta−mu d)
delta−mu−is−nt {n = O} f (mono d) = refl
delta−mu−is−nt {n = S n} f (delta (delta x d) ds) = begin
  delta (f x) (delta−mu (delta−fmap tailDelta (delta−fmap (delta−fmap f) ds)))
  ≡⟨ cong (\de → delta (f x) (delta−mu de)) (tailDelta−to−tail−nt n n f ds ) ⟩
  delta (f x) (delta−mu (delta−fmap (delta−fmap f) (delta−fmap tailDelta ds)))
  ≡⟨ cong (\de → delta (f x) de) (delta−mu−is−nt f (delta−fmap tailDelta ds)) ⟩
  delta (f x) (delta−fmap f (delta−mu (delta−fmap tailDelta ds)))
  ■
```

3. T に対する演算の単位元が存在する

   図 3.5 で示した右側の可換図に相当する。単位元に相当する演算は右側と左側の 2 つが存在するため、証明も 2 つに分割する。

   右側単位元の証明をリスト 6.11 に示す。

   リスト 6.11: Agda における Delta に対する演算に右側単位元が存在する証明

```
delta−right−unity−law : {l : Level} {A : Set l} {n : Nat} →
   (d : Delta A (S n)) → (delta−mu ∘ delta−eta) d ≡ id d
delta−right−unity−law (mono x)    = refl
delta−right−unity−law (delta x d) = begin
  (delta−mu ∘ delta−eta) (delta x d)
  ≡⟨ refl ⟩
  delta−mu (delta−eta (delta x d))
  ≡⟨ refl ⟩
  delta−mu (delta (delta x d) (delta−eta (delta x d)))
  ≡⟨ refl ⟩
  delta (headDelta (delta x d)) (delta−mu (delta−fmap tailDelta (delta−eta (delta x d))))
  ≡⟨ refl ⟩
  delta x (delta−mu (delta−fmap tailDelta (delta−eta (delta x d))))
  ≡⟨ cong (\de → delta x (delta−mu de)) (sym (delta−eta−is−nt  tailDelta (delta x d))) ⟩
  delta x (delta−mu (delta−eta (tailDelta (delta x d))))
  ≡⟨ refl ⟩
  delta x (delta−mu (delta−eta d))
  ≡⟨ cong (\de → delta x de) (delta−right−unity−law d) ⟩
  delta x d
  ≡⟨ refl ⟩
  id (delta x d)  ■
```

   バージョンが 1 である時は同じ項となるため refl となる。バージョンが 1 以上である時は再帰的に定義することになるが、途中で η が natural transformation である性質を利用している。

   次に左側単位元の証明をリスト 6.12 に示す。

   これまでの証明と同様にバージョンが 1 である場合は定義から同じ項となる。バージョンが 1 以上の場合、Functor 則の関数の合成の保存を利用して証明を再帰的に行なう。

リスト 6.12: Agda における Delta に対する演算に左側単位元が存在する証明

```
delta−left−unity−law : {l : Level} {A : Set l} {n : Nat} → (d : Delta A (S n)) →
                       (delta−mu ∘ (delta−fmap delta−eta)) d ≡ id d
delta−left−unity−law (mono x)    = refl
delta−left−unity−law {n = (S n)} (delta x d) = begin
  (delta−mu ∘ delta−fmap delta−eta) (delta x d)          ≡⟨ refl ⟩
  delta−mu ( delta−fmap delta−eta (delta x d))          ≡⟨ refl ⟩
  delta−mu (delta (delta−eta x) (delta−fmap delta−eta d))  ≡⟨ refl ⟩
  delta (headDelta {n = S n} (delta−eta x)) (delta−mu (delta−fmap tailDelta (delta−fmap delta−eta d)))
  ≡⟨ refl ⟩
  delta x (delta−mu (delta−fmap tailDelta (delta−fmap delta−eta d)))
  ≡⟨ cong (\de → delta x (delta−mu de)) (sym (delta−covariant tailDelta delta−eta d)) ⟩
  delta x (delta−mu (delta−fmap (tailDelta ∘ delta−eta {n = S n}) d))  ≡⟨ refl ⟩
  delta x (delta−mu (delta−fmap (delta−eta {n = n}) d))  ≡⟨ cong (\de → delta x de) (delta−left−unity−law d) ⟩
  delta x d ≡⟨ refl ⟩
  id (delta x d)  ■
```

4. T に対する演算の可換性が存在する

図 3.5 で示した左側の可換図に相当する。演算の可換性の証明をリスト 6.13 に示す。

リスト 6.13: Agda における Delta に対する演算に可換性が存在する証明

```
delta−fmap−mu−to−tail : {l : Level} {A : Set l} (n m : Nat) →
   (d : Delta (Delta (Delta A (S (S m))) (S (S m))) (S n)) →
  delta−fmap tailDelta (delta−fmap delta−mu d)
  ≡
  (delta−fmap delta−mu (delta−fmap (delta−fmap tailDelta) (delta−fmap tailDelta d)))
delta−fmap−mu−to−tail O O (mono (delta d ds)) = refl
delta−fmap−mu−to−tail O (S n) (mono (delta (delta x (delta xx d)) (delta (delta dx (delta dxx dd)) ds))) = refl
delta−fmap−mu−to−tail (S n) O (delta (delta (delta x (mono xx)) (mono (delta dx (mono dxx)))) ds) = begin
  delta (mono dxx) (delta−fmap tailDelta (delta−fmap delta−mu ds))
  ≡⟨ cong (\de → delta (mono dxx) de) (delta−fmap−mu−to−tail n O ds) ⟩
  delta (mono dxx)
    (delta−fmap delta−mu
     (delta−fmap (delta−fmap tailDelta) (delta−fmap tailDelta ds)))
  ■
delta−fmap−mu−to−tail (S n) (S m) (delta (delta (delta x (delta xx d))
                                  (delta (delta dx (delta dxx dd)) ds)) dds) = begin
  delta (delta dxx (delta−mu (delta−fmap tailDelta (delta−fmap tailDelta ds))))
    (delta−fmap tailDelta (delta−fmap delta−mu dds))
  ≡⟨ cong (\de → delta (delta dxx (delta−mu (delta−fmap tailDelta (delta−fmap tailDelta ds)))) de)
        (delta−fmap−mu−to−tail n (S m) dds) ⟩
  delta (delta dxx (delta−mu (delta−fmap tailDelta (delta−fmap tailDelta ds))))
    (delta−fmap delta−mu (delta−fmap (delta−fmap tailDelta) (delta−fmap tailDelta dds)))
  ■


delta−association−law : {l : Level} {A : Set l} {n : Nat} (d : Delta (Delta (Delta A (S n)) (S n)) (S n)) →
          ((delta−mu ∘ (delta−fmap delta−mu)) d) ≡ ((delta−mu ∘ delta−mu) d)
delta−association−law {n =  O} (mono d)                    = refl
delta−association−law {n = S n} (delta (delta (delta x d) dd) ds) = begin
  delta x (delta−mu (delta−fmap tailDelta (delta−fmap delta−mu ds)))
  ≡⟨ cong (\de → delta x (delta−mu de)) (delta−fmap−mu−to−tail n n ds) ⟩
  delta x (delta−mu (delta−fmap delta−mu (delta−fmap (delta−fmap tailDelta) (delta−fmap tailDelta ds))))
  ≡⟨ cong (\de → delta x de)
        (delta−association−law (delta−fmap (delta−fmap tailDelta) (delta−fmap tailDelta ds))) ⟩
  delta x (delta−mu (delta−mu (delta−fmap (delta−fmap tailDelta) (delta−fmap tailDelta ds))))
  ≡⟨ cong (\de → delta x (delta−mu de)) (delta−mu−is−nt tailDelta (delta−fmap tailDelta ds) ) ⟩
  delta x (delta−mu (delta−fmap tailDelta (delta−mu (delta−fmap tailDelta ds))))
  ■
```

μ を用いて TTT から T にする際に、右側 2 つの T に対して先に μ を用いるか、左側 2 つから先に用いるかの可換性である。よって `Delta (Delta (Delta A))` から `Delta A` とする演算の等式となる。

バージョンが 1 である場合はやはり同じ項に簡約される。しかしバージョンが 1 以上である場合は複雑な式変形を行うため、部分的な証明 `delta-fmap-mu-to-tail` を

定義し、μ が適用できない項を変形している。

`delta-fmap-mu-to-tail` に加え、 μ が natural transformation であることを利用して再帰的に定義し証明した。

Delta Monad の定義と、5 つの証明を用いて Moand の record が構築できることを確認する (リスト 6.14)。

リスト 6.14: Agda における Delta が Monad 則を満たす証明

```
delta-is-monad : {l : Level} {n : Nat} →
        Monad {l} (\A → Delta A (S n))  delta-is-functor
delta-is-monad = record
               { eta              = delta-eta;
                 mu               = delta-mu;
                 eta-is-nt        = delta-eta-is-nt;
                 mu-is-nt         = delta-mu-is-nt;
                 association-law  = delta-association-law;
                 left-unity-law   = delta-left-unity-law ;
                 right-unity-law  = \x → (sym (delta-right-unity-law x)) }
```

全ての Delta が同じバージョンを持つ時、 $triple(T, \eta, \mu)$ は Monad 則を満たすことを示せた。

第7章　任意の Monad と Delta の組み合せ

第 6 章ではプログラムの変更を表現する Delta Monad が Monad であることを証明した。第 7 章では、Delta Monad と、他の Monad との組み合せを考える。特に他の Monad のメタ計算も同時に実行できるようなデータ DeltaM を定義し、利用例を示す。

7.1　Monad と組み合せた Delta である DeltaM

functional programming における monad とはデータ構造をメタ計算の対応付けであった。プログラミング言語 Haskell においては入出力や非決定性演算といった処理が monad として提供される。そのため、プログラムの変更を表現する Delta Monad では他の monad と組み合せることが可能でなくてはならない。Delta Monad を拡張し、Delta Monad の内部の monad のメタ計算を実行するデータ型 DeltaM を Haskell で定義する。

データ型 DeltaM と、DeltaM に対する Monad の instance 定義を示す (リスト 7.1)。

データ型 DeltaM は 2 つの型引数を持つ。型引数 a と型引数を持つ型引数 m である。m が Monad であり、 a が Monad が内包する型を表す。m a 型を内部に持つ Delta から DeltaM コンストラクタを用いて DeltaM 型を構成できる。型名とコンストラクタ名が同一になっているので注意が必要である。

次にいくつかの関数を定義する。unDeltaM は DeltaM から Delta へと戻す関数である。headDeltaM は DeltaM から先頭の値を取り出す。バージョン管理された Delta から最新のバージョンを取り出すことに相当する。tailDeltaM は DeltaM から先頭の値を取り除く。これはバージョン管理された Delta から最新以外のバージョンに対して処理を行う時に用いる。

DeltaM の instance 定義では >>= ではなく mu を介して行なう。ここで 2 つの新たな記法を解説する。

- `(Functor m) => ...`

  mu や Monad の instance 宣言に利用されている記法で、型制約を示す記号である。ある型 m が Functor の instance であることを示す。例えば、 mu に記述されている `(Functor m, Monad m) => ...` は型 m が Functor と Monad の instance である時に mu に適用できることを示す。型制約により、型 m a は Functor である保証がつくため、 m a の値に対して fmap をかけることできる。

リスト 7.1: DeltaM の定義と Monad の instance 定義

```
1  data DeltaM m a = DeltaM (Delta (m a)) deriving (Show)
2
3  -- DeltaM utils
4
5  unDeltaM :: DeltaM m a -> Delta (m a)
6  unDeltaM (DeltaM d) = d
7
8  headDeltaM :: DeltaM m a -> m a
9  headDeltaM (DeltaM d) = headDelta d
10
11 tailDeltaM :: DeltaM m a -> DeltaM m a
12 tailDeltaM (DeltaM d) = DeltaM $ tailDelta d
13
14 -- DeltaM instance definitions
15
16 mu :: (Functor m, Monad m) => DeltaM m (DeltaM m a) -> DeltaM m a
17 mu d@(DeltaM (Mono _))    =
18         DeltaM (Mono  ((>>= id) (fmap headDeltaM (headDeltaM d))))
19 mu d@(DeltaM (Delta _ _)) =
20         DeltaM (Delta ((>>= id) (fmap headDeltaM (headDeltaM d)))
21                       (unDeltaM (mu (fmap tailDeltaM (tailDeltaM d)))))
22
23 instance (Functor m, Monad m) => Monad (DeltaM m) where
24     return x = DeltaM (Mono (return x))
25     d >>= f  = mu (fmap f d)
```

- `d@(DeltaM (Mono ...))`

  パターンマッチにおける記法で、as パターンと呼ばれる。パターンマッチにおいてコンストラクタを指定したのち、全体の値を `@` の前の変数に束縛する。例えば `m@(Mono x)` と指定すると、 `Mono x` が m に束縛される。この記法を採用するメリットは、コンストラクタでパターンマッチしたものの、値はコンストラクタを含んだまま処理したい時にタイプ数を省略できることにある。

関数 mu では Delta におけるバージョンの管理と内部の Monad のメタ計算の両方を一度に行なう。

バージョンが 1 であった場合は内部の DeltaM から fmap headDeltaM を用いて値を取り出し、 `>>= id` で内部の Monad のメタ計算を行なう。同じように、バージョンが 1 以上であった場合は先頭のバージョンに対しメタ計算を行なってから、残りのバージョンに対して再帰的に mu を行なう。

mu が定義できたので、DeltaM を Monad の instance とする。m が Functor と Monad である時のみ、 DeltaM は Monad となる。

return は通常の値を内部の Monad の return でメタ計算に対応させ、バージョン 1 である Mono としてから DeltaM とする。中置関数 >>= は fmap と mu を用いて定義する。

以上で Delta Monad と 他の Monad を組み合せるための DeltaM が定義できた。

7.2 DeltaM を用いたプログラムの例

DeltaM を用いてプログラムを記述する。今回は簡易的なプログラムのトレースとして、実行時の値を文字列として残すこととする。Haskell では処理に対するログなどを残すための Monad として Writer という Monad を提供している。リスト 2.6 で示した numberCount プログラムに対し、Writer Monad と Delta Monad を組み合せてトレースの比較を組込む (リスト 7.2)。

リスト 7.2: DeltaM を用いたプログラムの例

```
1  type DeltaLog     = Writer [String]
2  type DeltaWithLog = DeltaM DeltaLog
3
4  returnW :: (Show a) => a -> DeltaLog a
5  returnW x = do tell ([show x])
6                 return x
7
8  deltaAppend :: Delta a -> Delta a -> Delta a
9  deltaAppend (Mono x) d     = Delta x d
10 deltaAppend (Delta x d) ds = Delta x (deltaAppend d ds)
11
12 deltaFromList :: [a] -> Delta a
13 deltaFromList = (foldl1 deltaAppend) . (fmap return)
14
15
16
17 generatorM :: Int -> DeltaWithLog [Int]
18 generatorM x = let intList = [1..x] in
19     DeltaM (deltaFromList (fmap returnW (replicate 2 intList)))
20
21 numberFilterM :: [Int] -> DeltaWithLog [Int]
22 numberFilterM xs = let primeList = filter isPrime xs
23                        evenList  = filter even xs    in
24     DeltaM (deltaFromList (fmap returnW [primeList, evenList]))
25
26
27 countM :: [Int] -> DeltaWithLog Int
28 countM xs = let numberCount = length xs in
29     DeltaM (deltaFromList (fmap returnW (replicate 2 numberCount)))
30
31 numberCountM :: Int -> DeltaWithLog Int
32 numberCountM x = generatorM x >>= numberFilterM >>= countM
```

Writer と組み合せた DeltaM を DeltaWithLog 型として定義する。型の定義には type 構文を用い、ある型の別名として再定義する。まず、Writer で記録する型を String の List とし、その型を DeltaLog とする。DeltaLog を内包する DeltaM として DeltaWithLog 型定義した。

また、値を文字列のログとするために関数 returnW を定義した。これは通常の値から DeltaLog を生成するための関数である。tell 関数を用いてログを Writer に書き込み、値を return で保持する。

処理を簡易化するために、リストから Delta を作成する関数 deltaFromList を定義しておく。

numberCountM プログラムは numberCount プログラムとほとんど違いは無い。以下の 3 つの関数の組み合せで作成されている。

- generatorM

  n を取り、1 から n までのリストを返すバージョン 1 とバージョン 2 に違いは無い。

- numberCountM

  整数のリストを取り、特定の条件によって絞り込む。バージョン 1 では素数のみを絞り込み、バージョン 2 では偶数を絞り込む。

- countM

  リストを取り、その要素の個数を返す。バージョン 1 とバージョン 2 に違いは無い。

唯一の違いは関数が最後に返す Delta に対して returnW 関数が適用されていることである。returnW を用いることで簡易的な値のトレースが得られる。

実行した結果はリスト 7.3 である。

リスト 7.3: NumberCountM プログラムの実行例

```
1 DeltaM (Delta (WriterT (Identity (8,
2
    ["[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]",
3                                      "[2,3,5,7,11,13,17,19]",
4                                      "8"])))
5         (Mono  (WriterT (Identity (10,
6
    ["[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]",
7                                      "[2,4,6,8,10,12,14,16,18,20]",
8                                      "10"]))))
```

numberCountM プログラムに対して 20 を与えて実行した結果である。なお、結果は読み易いように整形してある。ここで注目して欲しいのが、それぞれがどのような計算結果を辿ったのかを Writer が保持していることである。

2 つの結果とも、まずは generatorM 関数により 1 から 20 までのリストを作る。次に numberFilterM 関数によってフィルタされ、それぞれの結果による値が導かれる。バージョン 1 では素数の個数を数えるために 8 であり、バージョン 2 では偶数の個数を数えるために 10 となる。

このように、Monad と組み合せることでトレースを得ることができた。Writer 以外にも任意の Monad に対して DeltaM が Monad 則を満たす。この証明は非常に長いので付録 B に載せるものとする。

DeltaM を定義した結果、Delta Monad と Monad を組み合せることができた。

第8章　まとめと今後の課題

本研究ではプログラムの変更を Monad を用いて形式化した Delta Monad を提案した。Delta Monad の実装例として Haskell におけるメタ計算でプログラムの変更を定義した。Delta Monad により、複数のバージョンを持つプログラムを表現し、全てのバージョンを同時に実行した。加えてデータ型 DeltaM を定義することで、Delta Monad が他の Monad と組み合せられることを示した。例として Writer Monad を組み合せることで複数のバージョンを同時に実行しながら個々のトレースを得た。トレースを比較することにより、デバッグやテストに有用な情報を提供できることを示した。また、定義した Delta と DeltaM が Monad 則を満たすことを Agda において証明した。

課題として挙げられるのは Delta Monad により表現可能な変更の定義である。プログラムに対する可能な変更を全て Delta Monad で表現可能である証明は行なっていない。証明可能かを確認し、不可能ならば Delta Monad により表現できる変更を定義する。次に、 category theory によるプログラムの変更に対する意味付けがある。category theory では category の構造から性質を導いたり、他の category への関連を導くことができる。プログラムの変更を Monad として表した時に持つ意味や、 category が持つ有益な性質をプログラムに適用したい。特に、複数のプログラムを同時に実行するのは product に、全てのプログラムを生成できる Delta 全体を表す集合は colimit に関連があると考えている。

参考文献

[1] The agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Accessed: 2015/02/17(Tue).

[2] Git. `http://git-scm.com/`. Accessed: 2015/02/17(Tue).

[3] Mercurial scm. `http://mercurial.selenic.com/`. Accessed: 2015/02/17(Tue).

[4] Michael Barr and Charles Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice-Hall, 1990. Second edition, 1995.

[5] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[6] M. P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, December 1993.

[7] Joachim (mathématicien) Lambek and P. J. Scott. *Introduction to higher order categorical logic*. Cambridge studies in advanced mathematics. Cambridge University Press, Cambridge, New York (N. Y.), Melbourne, 1986.

[8] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92, July 1991.

謝辞

本研究の遂行、また本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治准教授に深く感謝致します。

また一年間共に研究を行い、暖かな気遣いと励ましをもって支えてくれた並列信頼研究室のみなさんに感謝致します。

最後に、有意義な時間を共に過ごした情報工学科の学友、並びに物心両面で支えてくれた両親に深く感謝致します。

2015 年 3 月

比嘉健太

付 録A　Haskell による Delta と DeltaM の定義と使用例のプログラム

A.1　ソフトウェアのバージョン一覧

本論文で用いたプログラムやライブラリのバージョンを表 A.1 に示す。

	version
Mac OS X	10.9.5
ghc	7.8.4
Agda	2.4.2.2
agda-stdlib	0.9
cabal	1.22.0.0

表 A.1: 実験環境

A.2　Delta と DeltaM のソースコード

リスト A.1: Delta のソースコード

```
module Delta ( Delta(..) , deltaAppend , headDelta , tailDelta , deltaFromList) where

import Control.Applicative


-- Delta definition

data Delta a = Mono a | Delta a (Delta a) deriving Show

instance (Eq a) => Eq (Delta a) where
   (Mono x) == (Mono y)       = x == y
   (Mono _) == (Delta _ _)    = False
   (Delta x xs) == (Delta y ys) = (x == y)  (xs == ys)

-- basic functions

deltaAppend :: Delta a -> Delta a -> Delta a
deltaAppend (Mono x) d     = Delta x d
deltaAppend (Delta x d) ds = Delta x (deltaAppend d ds)

headDelta :: Delta a -> a
headDelta (Mono  x)   = x
headDelta (Delta x _) = x

tailDelta :: Delta a -> Delta a
tailDelta d@(Mono _)   = d
tailDelta (Delta _ ds) = ds

-- instance definitions

instance Functor Delta where
   fmap f (Mono x)    = Mono  (f x)
   fmap f (Delta x d) = Delta (f x) (fmap f d)

instance Applicative Delta where
   pure f                  = Mono  f
   (Mono f)     <*> (Mono x)    = Mono  (f x)
   df@(Mono f)  <*> (Delta x d) = Delta (f x) (df <*> d)
   (Delta f df) <*> d@(Mono x)  = Delta (f x) (df <*> d)
   (Delta f df) <*> (Delta x d) = Delta (f x) (df <*> d)

bind :: (Delta a) -> (a -> Delta b) -> (Delta b)
bind (Mono x)    f = f x
bind (Delta x d) f = Delta (headDelta (f x)) (bind d (tailDelta . f))

mu :: (Delta (Delta a)) -> (Delta a)
mu d = bind d id

instance Monad Delta where
   return x = Mono x
   d >>= f  = mu $ fmap f d

-- utils

deltaFromList :: [a] -> Delta a
deltaFromList = (foldl1 deltaAppend) . (fmap return)
```

リスト A.2: DeltaM のソースコード

```
module DeltaM (DeltaM(..), unDeltaM, appendDeltaM, tailDeltaM, headDeltaM, checkOut) where

import Control.Applicative
import Delta


-- DeltaM definition (Delta with Monad)

data DeltaM m a = DeltaM (Delta (m a)) deriving (Show)


-- DeltaM utils

unDeltaM :: DeltaM m a -> Delta (m a)
unDeltaM (DeltaM d) = d

headDeltaM :: DeltaM m a -> m a
headDeltaM (DeltaM d) = headDelta d

tailDeltaM :: DeltaM m a -> DeltaM m a
tailDeltaM (DeltaM d) = DeltaM $ tailDelta d

appendDeltaM :: DeltaM m a -> DeltaM m a -> DeltaM m a
appendDeltaM (DeltaM d) (DeltaM dd) = DeltaM (deltaAppend d dd)

checkOut :: Int -> DeltaM m a -> m a
checkOut 0 (DeltaM (Mono x))    = x
checkOut 0 (DeltaM (Delta x _)) = x
checkOut n (DeltaM (Mono x))    = x
checkOut n (DeltaM (Delta _ d)) = checkOut (n-1) (DeltaM d)


-- DeltaM instance definitions

instance (Functor m) => Functor (DeltaM m) where
   fmap f (DeltaM d) = DeltaM $ fmap (fmap f) d

instance (Applicative m) => Applicative (DeltaM m) where
   pure f                                  = DeltaM $ Mono $ pure f
   (DeltaM (Mono f))     <*> (DeltaM (Mono x))     = DeltaM $ Mono $ f <*> x
   df@(DeltaM (Mono f))  <*> (DeltaM (Delta x d))  = appendDeltaM (DeltaM $ Mono $ f <*> x) (df <*> (DeltaM d))
   (DeltaM (Delta f df)) <*> dx@(DeltaM (Mono x))  = appendDeltaM (DeltaM $ Mono $ f <*> x) ((DeltaM df) <*> dx)
   (DeltaM (Delta f df)) <*> (DeltaM (Delta x dx)) = appendDeltaM (DeltaM $ Mono $ f <*> x) ((DeltaM df) <*> (
     DeltaM dx))


mu' :: (Functor m, Monad m) => DeltaM m (DeltaM m a) -> DeltaM m a
mu' d@(DeltaM (Mono _))    = DeltaM $ Mono $ (>>= id) $ fmap headDeltaM $ headDeltaM d
mu' d@(DeltaM (Delta _ _)) = DeltaM $ Delta ((>>= id) $ fmap headDeltaM $ headDeltaM d)
                                    (unDeltaM (mu' (fmap tailDeltaM (tailDeltaM d))))

instance (Functor m, Monad m) => Monad (DeltaM m) where
   return x = DeltaM $ Mono $ return x
   d >>= f  = mu' $ fmap f d
```

付 録B　DeltaM が Monad 則を満たす証明

DeltaM に対する Monad 則の証明も Agda によって行なう。

内部に持つ型 M は型引数を持ち、 Functor と Monad の制約を持つ。ある法則を証明する時は内部の M の証明を使って証明することになる。例えば DeltaM の η が natural transformation である証明には、 M の η が natural transformation である証明を用いる。

また、証明に辺り fmap-equiv という等式を定義している。この等式は同じ値に対して同じ振舞いをする関数を fmap しても同じである等式である。

DeltaM に対する Monad 則の証明がリスト B.1 である。プログラム内部の Delta のバージョン数は全て同じであるとし、1 以上とする。なお、Functor 則に対する証明も行なったが省略する。

リスト B.1: DeltaM が Monad 則を満たす証明

```
1  open import Level
2  open import Relation.Binary.PropositionalEquality
3  open ≡−Reasoning
4
5  open import basic
6  open import delta
7  open import delta.functor
8  open import delta.monad
9  open import deltaM
10 open import deltaM.functor
11 open import nat
12 open import laws
13
14 module deltaM.monad where
15 open Functor
16 open NaturalTransformation
17 open Monad
18
19
20 −− sub proofs
21
22 deconstruct−id : {l : Level} {A : Set l} {n : Nat}
23             {T : Set l → Set l} {F : Functor T} {M : Monad T F} →
24             (d : DeltaM M A (S n)) → deltaM (unDeltaM d) ≡ d
25 deconstruct−id {n = O}   (deltaM x) = refl
26 deconstruct−id {n = S n} (deltaM x) = refl
27
28 headDeltaM−with−f : {l : Level} {A B : Set l} {n : Nat}
29                {T : Set l → Set l} {F : Functor T} {M : Monad T F}
30                (f : A → B) → (x : (DeltaM M A (S n))) →
31                ((fmap F f) ∘ headDeltaM) x ≡ (headDeltaM ∘ (deltaM−fmap f)) x
32 headDeltaM−with−f {n = O} f   (deltaM (mono x))    = refl
33 headDeltaM−with−f {n = S n} f (deltaM (delta x d)) = refl
34
35
36 tailDeltaM−with−f : {l : Level} {A B : Set l} {n : Nat}
37                {T : Set l → Set l} {F : Functor T} {M : Monad T F}
38                (f : A → B) → (d : (DeltaM M A (S (S n)))) →
39                (tailDeltaM ∘ (deltaM−fmap f)) d ≡ ((deltaM−fmap f) ∘ tailDeltaM) d
40 tailDeltaM−with−f {n = O} f (deltaM (delta x d))   = refl
41 tailDeltaM−with−f {n = S n} f (deltaM (delta x d)) = refl
42
43
44
45 fmap−headDeltaM−with−deltaM−mu : {l : Level} {A : Set l} {n : Nat}
46                          {T : Set l → Set l} {F : Functor T} {M : Monad T F} →
47                          (x : T (DeltaM M (DeltaM M A (S n)) (S n))) →
```

```
48 |              fmap F (headDeltaM ∘ deltaM−mu) x ≡ fmap F (((mu M) ∘ (fmap F headDeltaM)) ∘ headDeltaM) x
49 | fmap−headDeltaM−with−deltaM−mu {n = O}   x = refl
50 | fmap−headDeltaM−with−deltaM−mu {n = S n} x = refl
51 |
52 |
53 | fmap−tailDeltaM−with−deltaM−mu : {l : Level} {A : Set l} {n : Nat}
54 |                         {T : Set l → Set l} {F : Functor T} {M : Monad T F} →
55 |           (d : DeltaM M (DeltaM M (DeltaM M A (S (S n))) (S (S n))) (S n)) →
56 |            deltaM−fmap (tailDeltaM ∘ deltaM−mu) d ≡ deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap tailDeltaM)) ∘
       tailDeltaM) d
57 | fmap−tailDeltaM−with−deltaM−mu {n = O} (deltaM (mono x)) = refl
58 | fmap−tailDeltaM−with−deltaM−mu {n = S n} (deltaM d)     = refl
59 |
60 |
61 |
62 |
63 |
64 | −− main proofs
65 |
66 | deltaM−eta−is−nt : {l : Level} {A B : Set l} {n : Nat}
67 |              {T : Set l → Set l} {F : Functor T} {M : Monad T F} →
68 |              (f : A → B) → (x : A) →
69 |              ((deltaM−eta {l} {B} {n} {T} {F} {M} )∘ f) x ≡ deltaM−fmap f (deltaM−eta x)
70 | deltaM−eta−is−nt {l} {A} {B} {O} {M} {fm} {mm} f x   = begin
71 |  deltaM−eta {n = O} (f x)           ≡⟨ refl ⟩
72 |  deltaM (mono (eta mm (f x)))      ≡⟨ cong (\de → deltaM (mono de)) (eta−is−nt mm f x) ⟩
73 |  deltaM (mono (fmap fm f (eta mm x)))  ≡⟨ refl ⟩
74 |  deltaM−fmap f (deltaM−eta {n = O} x)  ■
75 | deltaM−eta−is−nt {l} {A} {B} {S n} {M} {fm} {mm} f x = begin
76 |  deltaM−eta {n = S n} (f x)
77 |  ≡⟨ refl ⟩
78 |  deltaM (delta−eta {n = S n} (eta mm (f x)))
79 |  ≡⟨ refl ⟩
80 |  deltaM (delta (eta mm (f x)) (delta−eta (eta mm (f x))))
81 |  ≡⟨ cong (\de → deltaM (delta de (delta−eta de))) (eta−is−nt mm f x) ⟩
82 |  deltaM (delta (fmap fm f (eta mm x)) (delta−eta (fmap fm f (eta mm x))))
83 |  ≡⟨ cong (\de →  deltaM (delta (fmap fm f (eta mm x)) de)) (eta−is−nt delta−is−monad (fmap fm f) (eta mm x)) ⟩
84 |  deltaM (delta (fmap fm f (eta mm x)) (delta−fmap (fmap fm f) (delta−eta (eta mm x))))
85 |  ≡⟨ refl ⟩
86 |  deltaM−fmap f (deltaM−eta {n = S n} x)
87 |  ■
88 |
89 |
90 |
91 |
92 | deltaM−mu−is−nt : {l : Level} {A B : Set l} {n : Nat}
93 |             {T : Set l → Set l} {F : Functor T} {M : Monad T F}
94 |             (f : A → B) → (d : DeltaM M (DeltaM M A (S n)) (S n)) →
95 |             deltaM−fmap f (deltaM−mu d) ≡ deltaM−mu (deltaM−fmap (deltaM−fmap f) d)
96 | deltaM−mu−is−nt {l} {A} {B} {O} {T} {F} {M}  f (deltaM (mono x)) = begin
97 |  deltaM−fmap f (deltaM−mu (deltaM (mono x)))
98 |  ≡⟨ refl ⟩
99 |  deltaM−fmap f (deltaM (mono (mu M (fmap F headDeltaM x))))
100 |  ≡⟨ refl ⟩
101 |  deltaM (mono (fmap F f (mu M (fmap F headDeltaM x))))
102 |  ≡⟨ cong (\de → deltaM (mono de)) (sym (mu−is−nt M f (fmap F headDeltaM x))) ⟩
103 |  deltaM (mono (mu M (fmap F (fmap F f) (fmap F headDeltaM x))))
104 |  ≡⟨ cong (\de → deltaM (mono (mu M de))) (sym (covariant F headDeltaM (fmap F f) x)) ⟩
105 |  deltaM (mono (mu M (fmap F ((fmap F f) ∘ headDeltaM) x)))
106 |  ≡⟨ cong (\de → deltaM (mono (mu M de))) (fmap−equiv F (headDeltaM−with−f f) x) ⟩
107 |  deltaM (mono (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x)))
108 |  ≡⟨ cong (\de → deltaM (mono (mu M de))) (covariant F (deltaM−fmap f) (headDeltaM) x) ⟩
109 |  deltaM (mono (mu M (fmap F (headDeltaM {M = M}) (fmap F (deltaM−fmap f) x))))
110 |  ≡⟨ refl ⟩
111 |  deltaM (mono (mu M (fmap F (headDeltaM {M = M}) (headDeltaM {M = M} (deltaM (mono (fmap F (deltaM−fmap f) x)))))))
112 |  ≡⟨ refl ⟩
113 |  deltaM−mu (deltaM (mono (fmap F (deltaM−fmap f) x)))
114 |  ≡⟨ refl ⟩
115 |  deltaM−mu (deltaM−fmap (deltaM−fmap f) (deltaM (mono x)))
116 |  ■
117 | deltaM−mu−is−nt {l} {A} {B} {S n} {T} {F} {M} f (deltaM (delta x d)) = begin
118 |  deltaM−fmap f (deltaM−mu (deltaM (delta x d))) ≡⟨ refl ⟩
119 |  deltaM−fmap f (deltaM (delta (mu M (fmap F (headDeltaM {M = M}) (headDeltaM {M = M} (deltaM (delta x d)))))
120 |                        (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (tailDeltaM (deltaM (delta x d))))))))
121 |  ≡⟨ refl ⟩
122 |  deltaM−fmap f (deltaM (delta (mu M (fmap F (headDeltaM {M = M}) x))
123 |                        (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))))
124 |  ≡⟨ refl ⟩
125 |  deltaM (delta (fmap F f (mu M (fmap F (headDeltaM {M = M}) x)))
126 |            (delta−fmap (fmap F f) (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))))
127 |  ≡⟨ cong (\de → deltaM (delta de
128 |              (delta−fmap (fmap F f) (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d)))))))
129 |         (sym (mu−is−nt M f (fmap F headDeltaM x)))  ⟩
130 |  deltaM (delta (mu M (fmap F (fmap F f) (fmap F (headDeltaM {M = M}) x)))
131 |            (delta−fmap (fmap F f) (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))))
```

```
  ≡⟨ cong (\de → deltaM (delta (mu M de) (delta−fmap (fmap F f) (unDeltaM {M = M} (deltaM−mu (deltaM−fmap
     tailDeltaM (deltaM d)))))))
        (sym (covariant F headDeltaM (fmap F f) x)) ⟩
  deltaM (delta (mu M (fmap F ((fmap F f) ∘ (headDeltaM {M = M})) x))
            (delta−fmap (fmap F f) (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))))
  ≡⟨ cong (\de → deltaM (delta (mu M de) (delta−fmap (fmap F f) (unDeltaM {M = M} (deltaM−mu (deltaM−fmap
     tailDeltaM (deltaM d)))))))
        (fmap−equiv F (headDeltaM−with−f f) x) ⟩
  deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x))
            (delta−fmap (fmap F f) (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x))
            (unDeltaM {M = M} (deltaM−fmap f (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))))
  ≡⟨ cong (\de → deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x)) (unDeltaM de)))
        (deltaM−mu−is−nt {l} {A} {B} {n} {T} {F} {M} f (deltaM−fmap tailDeltaM (deltaM d))) ⟩
  deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap (deltaM−fmap {n = n} f) (deltaM−fmap {n = n} (tailDeltaM
     {n = n}) (deltaM d))))))
  ≡⟨ cong (\de → deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x)) (unDeltaM {M = M} (deltaM
     −mu de))))
        (sym (deltaM−covariant (deltaM−fmap f) tailDeltaM (deltaM d))) ⟩
  deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap {n = n} ((deltaM−fmap {n = n} f) ∘ (tailDeltaM {n = n})) (
     deltaM d)))))

  ≡⟨ cong (\de → deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x)) (unDeltaM {M = M} (deltaM
     −mu de))))
         (sym (deltaM−fmap−equiv (tailDeltaM−with−f f) (deltaM d))) ⟩
  deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap (tailDeltaM ∘ (deltaM−fmap f)) (deltaM d)))))
  ≡⟨ cong (\de → deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x)) (unDeltaM {M = M} (deltaM
     −mu de))))
        (deltaM−covariant tailDeltaM (deltaM−fmap f) (deltaM d)) ⟩
  deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM−fmap (deltaM−fmap f) (deltaM d))))))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F ((headDeltaM {M = M}) ∘ (deltaM−fmap f)) x))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F (deltaM−fmap f)) d)
     )))))
  ≡⟨ cong (\de → deltaM (delta (mu M de) (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−
     fmap (fmap F (deltaM−fmap f)) d)))))))
        (covariant F (deltaM−fmap f) headDeltaM x) ⟩
  deltaM (delta (mu M (fmap F (headDeltaM {M = M}) (fmap F (deltaM−fmap f) x)))
                (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F (deltaM−fmap f
     )) d))))))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F (headDeltaM {M = M}) (headDeltaM {M = M} (deltaM (delta (fmap F (deltaM−fmap f) x) (
     delta−fmap (fmap F (deltaM−fmap f)) d))))))
                (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (tailDeltaM (deltaM (delta (fmap F (deltaM−
     fmap f) x) (delta−fmap (fmap F (deltaM−fmap f)) d)))))))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM (delta (fmap F (deltaM−fmap f) x) (delta−fmap (fmap F (deltaM−fmap f)) d)))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM−fmap (deltaM−fmap f) (deltaM (delta x d)))
  ■
```

```
deltaM−right−unity−law : {l : Level} {A : Set l} {n : Nat}
                  {T : Set l → Set l} {F : Functor T} {M : Monad T F} →
                  (d : DeltaM M A (S n)) → (deltaM−mu ∘ deltaM−eta) d ≡ id d
deltaM−right−unity−law {l} {A} {O} {M} {fm} {mm} (deltaM (mono x)) = begin
  deltaM−mu (deltaM−eta (deltaM (mono x)))          ≡⟨ refl ⟩
  deltaM−mu (deltaM (mono (eta mm (deltaM (mono x))))) ≡⟨ refl ⟩
  deltaM (mono (mu mm (fmap fm (headDeltaM {M = mm})(eta mm (deltaM (mono x))))))
  ≡⟨ cong (\de → deltaM (mono (mu mm de))) (sym (eta−is−nt mm headDeltaM (deltaM (mono x)))) ⟩
  deltaM (mono (mu mm (eta mm ((headDeltaM {l} {A} {O} {M} {fm} {mm}) (deltaM (mono x)))))) ≡⟨ refl ⟩
  deltaM (mono (mu mm (eta mm x))) ≡⟨ cong (\de → deltaM (mono de)) (sym (right−unity−law mm x)) ⟩
  deltaM (mono x)
  ■
deltaM−right−unity−law {l} {A} {S n} {T} {F} {M} (deltaM (delta x d)) = begin
  deltaM−mu (deltaM−eta (deltaM (delta x d)))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM (delta (eta M (deltaM (delta x d))) (delta−eta (eta M (deltaM (delta x d))))))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F (headDeltaM {M = M}) (headDeltaM {M = M} (deltaM (delta {l} {T (DeltaM M A (S (S n)))}
     {n} (eta M (deltaM (delta x d))) (delta−eta (eta M (deltaM (delta x d))))))))))
           (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (tailDeltaM (deltaM (delta (eta M (deltaM (delta
     x d))) (delta−eta (eta M (deltaM (delta x d))))))))))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F (headDeltaM {M = M}) (eta M (deltaM (delta x d))))))
```

```
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−eta (eta M (deltaM (delta x d))))
    )))))
  ≡⟨ cong (\de → deltaM (delta (mu M de)
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−eta (eta M (deltaM (delta x d))))
    ))))))
     (sym (eta−is−nt M headDeltaM (deltaM (delta x d)))) ⟩
  deltaM (delta (mu M (eta M (headDeltaM {M = M} (deltaM (delta x d)))))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−eta (eta M (deltaM (delta x d))))
    )))))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (eta M x))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−eta (eta M (deltaM (delta x d))))
    )))))
  ≡⟨ cong (\de → deltaM (delta de (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−eta (eta
     M (deltaM (delta x d)))))))))
        (sym (right−unity−law M x)) ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−eta (eta M (deltaM (delta
     x d)))))))))
  ≡⟨ refl ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM (delta−fmap (fmap F tailDeltaM) (delta−eta (eta M (deltaM
     (delta x d)))))))))
  ≡⟨ cong (\de → deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM de)))))
        (sym (delta−eta−is−nt (fmap F tailDeltaM)  (eta M (deltaM (delta x d))))) ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM (delta−eta (fmap F tailDeltaM (eta M (deltaM (delta x d)))))
    )))
  ≡⟨ cong (\de → deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM (delta−eta de))))))
        (sym (eta−is−nt M tailDeltaM (deltaM (delta x d)))) ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM (delta−eta (eta M (tailDeltaM (deltaM (delta x d))))))))
  ≡⟨ refl ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM (delta−eta (eta M (deltaM d))))))
  ≡⟨ refl ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM−eta (deltaM d)))))
  ≡⟨ cong (\de → deltaM (delta x (unDeltaM {M = M} de))) (deltaM−right−unity−law (deltaM d)) ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM d)))
  ≡⟨ refl ⟩
  deltaM (delta x d)
  ■


deltaM−left−unity−law : {l : Level} {A : Set l} {n : Nat}
                  {T : Set l → Set l} {F : Functor T} {M : Monad T F}
                  (d : DeltaM M A (S n)) → (deltaM−mu ∘ (deltaM−fmap deltaM−eta)) d ≡ id d
deltaM−left−unity−law {l} {A}  {O} {T} {F} {M} (deltaM (mono x)) = begin
  deltaM−mu (deltaM−fmap deltaM−eta (deltaM (mono x))) ≡⟨ refl ⟩
  deltaM−mu (deltaM (mono (fmap F deltaM−eta x)))     ≡⟨ refl ⟩
  deltaM (mono (mu M (fmap F (headDeltaM {M = M}) (headDeltaM {M = M} (deltaM (mono (fmap F deltaM−eta x)))))))
     ≡⟨ refl ⟩
  deltaM (mono (mu M (fmap F (headDeltaM {M = M}) (fmap F deltaM−eta x))))
  ≡⟨ cong (\de → deltaM (mono (mu M de))) (sym (covariant F deltaM−eta headDeltaM x)) ⟩
  deltaM (mono (mu M (fmap F ((headDeltaM {n = O} {M = M}) ∘ deltaM−eta) x)))
  ≡⟨ refl ⟩
  deltaM (mono (mu M (fmap F (eta M) x)))
  ≡⟨ cong (\de → deltaM (mono de)) (left−unity−law M x) ⟩
  deltaM (mono x)
  ■
deltaM−left−unity−law {l} {A} {S n} {T} {F} {M} (deltaM (delta x d)) = begin
  deltaM−mu (deltaM−fmap deltaM−eta (deltaM (delta x d)))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM (delta (fmap F deltaM−eta x) (delta−fmap (fmap F deltaM−eta) d)))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F (headDeltaM {n = S n} {M = M}) (headDeltaM {n = S n} {M = M} (deltaM (delta (fmap F
     deltaM−eta x) (delta−fmap (fmap F deltaM−eta) d))))))
  (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (tailDeltaM (deltaM (delta (fmap F deltaM−eta x) (delta−
     fmap (fmap F deltaM−eta) d)))))))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F (headDeltaM {n = S n} {M = M}) (fmap F deltaM−eta x)))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F deltaM−eta) d))))))
  ≡⟨ cong (\de → deltaM (delta (mu M de) (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−
     fmap (fmap F deltaM−eta) d)))))))
        (sym (covariant F deltaM−eta headDeltaM x)) ⟩
  deltaM (delta (mu M (fmap F ((headDeltaM {n = S n} {M = M}) ∘  deltaM−eta) x))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F deltaM−eta) d))))))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F (eta M) x))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F deltaM−eta) d))))))
  ≡⟨ cong (\de → deltaM (delta de (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (
     fmap F deltaM−eta) d)))))))
        (left−unity−law M x) ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F deltaM−eta)
```

```
    d))))))
  ≡⟨ refl ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM−fmap (tailDeltaM {n = n})(deltaM−fmap (deltaM−eta {n = S n})(deltaM d))))))
  ≡⟨ cong (\de → deltaM (delta x (unDeltaM {M = M} (deltaM−mu de)))) (sym (deltaM−covariant tailDeltaM deltaM−eta (deltaM d))) ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((tailDeltaM {n = n}) ∘ (deltaM−eta {n = S n})) (deltaM d)))))
  ≡⟨ refl ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM−mu (deltaM−fmap deltaM−eta (deltaM d)))))
  ≡⟨ cong (\de → deltaM (delta x (unDeltaM {M = M} de))) (deltaM−left−unity−law (deltaM d)) ⟩
  deltaM (delta x (unDeltaM {M = M} (deltaM d)))
  ≡⟨ refl ⟩
  deltaM (delta x d)
  ■

deltaM−association−law : {l : Level} {A : Set l} {n : Nat}
                  {T : Set l → Set l} {F : Functor T} {M : Monad T F}
                  (d : DeltaM M (DeltaM M (DeltaM M A (S n)) (S n))  (S n)) →
                  deltaM−mu (deltaM−fmap deltaM−mu d) ≡ deltaM−mu (deltaM−mu d)
deltaM−association−law {l} {A}  {O} {T} {F} {M} (deltaM (mono x))   = begin
  deltaM−mu (deltaM−fmap deltaM−mu (deltaM (mono x)))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM (mono (fmap F deltaM−mu x)))
  ≡⟨ refl ⟩
  deltaM (mono (mu M (fmap F (headDeltaM {M = M}) (headDeltaM {M = M} (deltaM (mono (fmap F deltaM−mu x)))))))
  ≡⟨ refl ⟩
  deltaM (mono (mu M (fmap F (headDeltaM {M = M}) (fmap F deltaM−mu x))))
  ≡⟨ cong (\de → deltaM (mono (mu M de))) (sym (covariant F deltaM−mu headDeltaM x)) ⟩
  deltaM (mono (mu M (fmap F ((headDeltaM {M = M}) ∘ deltaM−mu) x)))
  ≡⟨ refl ⟩
  deltaM (mono (mu M (fmap F (((mu M) ∘ (fmap F headDeltaM)) ∘ headDeltaM) x)))
  ≡⟨ cong (\de → deltaM (mono (mu M de))) (covariant F headDeltaM ((mu M) ∘ (fmap F headDeltaM)) x) ⟩
  deltaM (mono (mu M ((((fmap F (((mu M) ∘ (fmap F headDeltaM)))) ∘ (fmap F headDeltaM)) x))))
  ≡⟨ refl ⟩
  deltaM (mono (mu M ((((fmap F (((mu M) ∘ (fmap F headDeltaM)))) (fmap F headDeltaM x))))))
  ≡⟨ cong (\de → deltaM (mono (mu M de))) (covariant F (fmap F headDeltaM) (mu M) (fmap F headDeltaM x)) ⟩
  deltaM (mono (mu M (((fmap F (mu M)) ∘ (fmap F (fmap F headDeltaM))) (fmap F headDeltaM x))))
  ≡⟨ refl ⟩
  deltaM (mono (mu M (fmap F (mu M) ((fmap F (fmap F headDeltaM) (fmap F headDeltaM x))))))
  ≡⟨ cong (\de → deltaM (mono de)) (association−law M (fmap F (fmap F headDeltaM) (fmap F headDeltaM x))) ⟩
  deltaM (mono (mu M (mu M (fmap F (fmap F headDeltaM) (fmap F headDeltaM x)))))
  ≡⟨ cong (\de → deltaM (mono (mu M de))) (mu−is−nt M headDeltaM (fmap F headDeltaM x)) ⟩
  deltaM (mono (mu M (fmap F (headDeltaM {M = M}) (mu M (fmap F (headDeltaM {M = M}) x)))))
  ≡⟨ refl ⟩
  deltaM (mono (mu M (fmap F (headDeltaM {M = M}) (headDeltaM {M = M} (deltaM (mono (mu M (fmap F (headDeltaM {M = M}) x))))))))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM (mono (mu M (fmap F (headDeltaM {M = M}) x))))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM (mono (mu M (fmap F (headDeltaM {M = M}) (headDeltaM {M = M} (deltaM (mono x)))))))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM−mu (deltaM (mono x)))

  ■

deltaM−association−law {l} {A} {S n} {T} {F} {M} (deltaM (delta x d)) = begin
  deltaM−mu (deltaM−fmap deltaM−mu (deltaM (delta x d)))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM (delta (fmap F deltaM−mu x) (delta−fmap (fmap F deltaM−mu) d)))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F (headDeltaM {M = M}) (headDeltaM {A = DeltaM M A (S (S n))} {M = M} (deltaM (delta (fmap F deltaM−mu x) (delta−fmap (fmap F deltaM−mu) d))))))
         (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (tailDeltaM (deltaM (delta (fmap F deltaM−mu x) (delta−fmap (fmap F deltaM−mu) d))))))))
  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F (headDeltaM {A = A} {M = M}) (fmap F deltaM−mu x)))
            (unDeltaM {A = A} {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F deltaM−mu) d))))))
  ≡⟨ cong (\de → deltaM (delta (mu M de) (unDeltaM {A = A} {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F deltaM−mu) d))))))
        (sym (covariant F deltaM−mu headDeltaM x)) ⟩
  deltaM (delta (mu M (fmap F ((headDeltaM {A = A} {M = M}) ∘  deltaM−mu) x))
            (unDeltaM {A = A} {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F deltaM−mu) d))))))
  ≡⟨ cong (\de → deltaM (delta (mu M de)
                  (unDeltaM {A = A} {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F deltaM−mu) d))))))
        (fmap−headDeltaM−with−deltaM−mu {A = A} {M = M} x) ⟩
  deltaM (delta (mu M (fmap F (((mu M) ∘ (fmap F headDeltaM)) ∘ headDeltaM) x))
            (unDeltaM {A = A} {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM (delta−fmap (fmap F deltaM−mu) d))))))
```

```
342 |  ≡⟨ refl ⟩
343 |  deltaM (delta (mu M (fmap F (((mu M) ∘ (fmap F headDeltaM)) ∘ headDeltaM) x))
344 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM−fmap deltaM−mu (deltaM d))))))
345 |  ≡⟨ cong (\de → deltaM (delta (mu M (fmap F (((mu M) ∘ (fmap F headDeltaM)) ∘ headDeltaM) x))
346 |                        (unDeltaM {M = M} (deltaM−mu de))))
347 |        (sym (deltaM−covariant tailDeltaM deltaM−mu (deltaM d))) ⟩
348 |  deltaM (delta (mu M (fmap F (((mu M) ∘ (fmap F headDeltaM)) ∘ headDeltaM) x))
349 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap (tailDeltaM ∘ deltaM−mu) (deltaM d)))))
350 |  ≡⟨ cong (\de → deltaM (delta (mu M (fmap F (((mu M) ∘ (fmap F headDeltaM)) ∘ headDeltaM) x))
351 |                        (unDeltaM {M = M} (deltaM−mu de))))
352 |        (fmap−tailDeltaM−with−deltaM−mu (deltaM d))  ⟩
353 |  deltaM (delta (mu M (fmap F (((mu M) ∘ (fmap F headDeltaM)) ∘ headDeltaM) x))
354 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap tailDeltaM)) ∘ tailDeltaM) (
    |     deltaM d)))))
355 |  ≡⟨ cong (\de → deltaM (delta (mu M de)
356 |                     (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap tailDeltaM)) ∘
    |     tailDeltaM) (deltaM d))))))
357 |       (covariant F headDeltaM ((mu M) ∘ (fmap F headDeltaM)) x) ⟩
358 |  deltaM (delta (mu M (((fmap F ((mu M) ∘ (fmap F headDeltaM))) ∘ (fmap F headDeltaM)) x))
359 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap tailDeltaM)) ∘ tailDeltaM) (
    |     deltaM d)))))
360 |  ≡⟨ refl ⟩
361 |  deltaM (delta (mu M (((fmap F ((mu M) ∘ (fmap F headDeltaM))) (fmap F headDeltaM x))))
362 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap tailDeltaM)) ∘ tailDeltaM) (
    |     deltaM d)))))
363 |  ≡⟨ cong (\de → deltaM (delta (mu M de)
364 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap tailDeltaM)) ∘ tailDeltaM) (
    |     deltaM d))))))
365 |       (covariant F (fmap F headDeltaM)  (mu M) (fmap F headDeltaM x)) ⟩
366 |
367 |  deltaM (delta (mu M ((((fmap F (mu M)) ∘ (fmap F (fmap F headDeltaM))) (fmap F headDeltaM x))))
368 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap tailDeltaM)) ∘ tailDeltaM) (
    |     deltaM d)))))
369 |  ≡⟨ refl ⟩
370 |  deltaM (delta (mu M (fmap F (mu M) (fmap F (fmap F headDeltaM) (fmap F headDeltaM x))))
371 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap tailDeltaM)) ∘ tailDeltaM) (
    |     deltaM d)))))
372 |  ≡⟨ cong (\de → deltaM (delta de (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap
    |     tailDeltaM)) ∘ tailDeltaM) (deltaM d))))))
373 |       (association−law M (fmap F (fmap F headDeltaM) (fmap F headDeltaM x))) ⟩
374 |  deltaM (delta (mu M (mu M (fmap F (fmap F headDeltaM) (fmap F headDeltaM x))))
375 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap tailDeltaM)) ∘ tailDeltaM) (
    |     deltaM d)))))
376 |  ≡⟨ cong (\de → deltaM (delta (mu M de) (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap
    |     tailDeltaM)) ∘ tailDeltaM) (deltaM d))))))
377 |       (mu−is−nt M headDeltaM (fmap F headDeltaM x)) ⟩
378 |  deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x))))
379 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap ((deltaM−mu ∘ (deltaM−fmap tailDeltaM)) ∘ tailDeltaM) (
    |     deltaM d)))))
380 |  ≡⟨ cong (\de →  deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x)))) (unDeltaM {M = M} (deltaM−
    |     mu de))))
381 |       (deltaM−covariant (deltaM−mu ∘ (deltaM−fmap tailDeltaM)) tailDeltaM (deltaM d)) ⟩
382 |  deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x))))
383 |            (unDeltaM {M = M} (deltaM−mu (((deltaM−fmap (deltaM−mu ∘ (deltaM−fmap tailDeltaM))  ∘ (deltaM−fmap
    |     tailDeltaM)) (deltaM d))))))
384 |  ≡⟨ refl ⟩
385 |  deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x))))
386 |            (unDeltaM {M = M} (deltaM−mu (((deltaM−fmap (deltaM−mu ∘ (deltaM−fmap tailDeltaM)) (deltaM−fmap
    |     tailDeltaM (deltaM d)))))))
387 |  ≡⟨ cong (\de → deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x)))) (unDeltaM {M = M} (deltaM−mu
    |      de))))
388 |       (deltaM−covariant deltaM−mu (deltaM−fmap tailDeltaM) (deltaM−fmap tailDeltaM (deltaM d)))  ⟩
389 |  deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x))))
390 |            (unDeltaM {M = M} (deltaM−mu (((deltaM−fmap deltaM−mu) ∘ (deltaM−fmap (deltaM−fmap tailDeltaM))) (
    |     deltaM−fmap tailDeltaM (deltaM d))))))
391 |  ≡⟨ refl ⟩
392 |  deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x))))
393 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap deltaM−mu (deltaM−fmap (deltaM−fmap tailDeltaM) (deltaM
    |     −fmap tailDeltaM (deltaM d)))))))
394 |  ≡⟨ cong (\de → deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x)))) (unDeltaM {M = M} de)))
395 |       (deltaM−association−law (deltaM−fmap (deltaM−fmap tailDeltaM) (deltaM−fmap tailDeltaM (deltaM d)))) ⟩
396 |  deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x))))
397 |            (unDeltaM {M = M} (deltaM−mu (deltaM−mu (deltaM−fmap (deltaM−fmap tailDeltaM) (deltaM−fmap
    |     tailDeltaM (deltaM d)))))))
398 |
399 |  ≡⟨ cong (\de → deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x))))
400 |                        (unDeltaM {M = M} (deltaM−mu de))))
401 |       (sym (deltaM−mu−is−nt tailDeltaM (deltaM−fmap tailDeltaM (deltaM d)))) ⟩
402 |  deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x))))
403 |            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))
    |     )))))
404 |  ≡⟨ cong (\de → deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x))))
```

```
                        (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM de)))))
        (sym (deconstruct−id (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))) ⟩
  deltaM (delta (mu M (fmap F headDeltaM (mu M (fmap F headDeltaM x))))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM
              (deltaM (unDeltaM {A = DeltaM M A (S (S n))} {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))
     )))))

  ≡⟨ refl ⟩
  deltaM (delta (mu M (fmap F headDeltaM (headDeltaM {M = M} ((deltaM (delta (mu M (fmap F headDeltaM x))
                    (unDeltaM {A = DeltaM M A (S (S n))} {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))))
     ))))
            (unDeltaM {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (tailDeltaM ((deltaM (delta (mu M (fmap F
     headDeltaM x))
                    (unDeltaM {A = DeltaM M A (S (S n))} {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))))
     ))))))

  ≡⟨ refl ⟩
  deltaM−mu (deltaM (delta (mu M (fmap F headDeltaM x))
                    (unDeltaM {A = DeltaM M A (S (S n))} {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (deltaM d))))))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM (delta (mu M (fmap F headDeltaM (headDeltaM {M = M} (deltaM (delta x d)))))
                    (unDeltaM {A = DeltaM M A (S (S n))} {M = M} (deltaM−mu (deltaM−fmap tailDeltaM (tailDeltaM (
     deltaM (delta x d))))))))
  ≡⟨ refl ⟩
  deltaM−mu (deltaM−mu (deltaM (delta x d)))
  ■



deltaM−is−monad : {l : Level} {A : Set l} {n : Nat}
             {T : Set l → Set l} {F : Functor T} {M : Monad T F} →
             Monad {l} (\A → DeltaM M A (S n)) (deltaM−is−functor {l} {n})
deltaM−is−monad {l} {A} {n} {T} {F} {M} =
            record { mu    = deltaM−mu
                   ; eta   = deltaM−eta
                   ; eta−is−nt = deltaM−eta−is−nt
                   ; mu−is−nt = (\f x → (sym (deltaM−mu−is−nt f x)))
                   ; association−law = deltaM−association−law
                   ; left−unity−law  = deltaM−left−unity−law
                   ; right−unity−law = (\x → (sym (deltaM−right−unity−law x)))
                   }
```