関数型言語と圏論の関係性

https://otamusan.github.io/FPandCategory/categoricalfp.pdf

2022 年 11 月 1 日

目次

1 Haskell とデザインパターン

オブジェクト指向言語におけるデザインパターンとは、プログラムの一部を再利用、テストを容易にするための設計手法である。これらは継承やら実装やらのオブジェクト指向言語特有の操作によって抽象化されていて、さらにそれらの操作で閉じている。

関数型プログラミング、特に Haskell ではリストの操作や入出力などの副作用を伴う操作を、抽象的で汎用性があり極力関数型言語の操作で構成される設計によって与えている。これらは言うなればデザインパターンによって言語機能の拡張しているとみなすことができ、こういった設計もここではデザインパターンと呼ぶことにする。

1.1 関数型

Haskell にも一般のプログラミングと同じようにデータ型が存在し、整数値を持つ `Int` 型や `True` と `False` を持つ `Bool` 型などがある。また整数値を受け取って二倍にして返す、つまり $double(x) = 2x$ なる関数 `double` は `Int` から `Int` への関数であり、haskell ではこのように記述する。

```
double :: Int -> Int
double x = 2*x
-- >>> double 4
-- 8
```

ある型 `a` からある型 `b` への関数は、関数の型 `a -> b` に含まれる。すなわち、関数そのものも値とみなすということである。それによって、関数型から関数型への関数なども定義できるようになる。その一例を示そう。

関数 `f :: Int -> Int` を、自身を合成した関数 `f.f :: Int -> Int` へと写す関数 `mul :: (Int -> Int) -> (Int -> Int)` は次のように記述する。

```
mul :: (Int -> Int) -> (Int -> Int)
mul f = f.f
-- >>> (mul double) (5)
-- 20
```

後に説明するが、`.` という記号は前後の関数を合成する操作であり、`f.f` は $f \circ f$ を意味する。すなわちこの計算は

$$(mul(double))(5) = (double \circ double)(5) = double(double(5)) = 20$$

のように行われる。このような関数を引数、戻り値にするような関数を高階関数と呼ぶ。

二つの整数を足し合わせる操作は $add(x, y) = x + y$ のように定義できる。この関数は引数を二つ持つ多変数関数であり、これは Haskell では

```
add :: Int -> Int -> Int
add x y = x + y
-- >>> add 5 6
-- 11
```

のように型を与えて定義することができる。関数の型 `Int -> Int -> Int` は左から二つは引数の型で、一番右が返り値の型となっている。なぜこのような表記をするかというと、`Int -> Int -> Int` は `Int -> (Int -> Int)` の省略であり、$add(x, y)$ を $(add(x))(y)$ とみなしているためである。ゆえに `add(x) :: Int -> Int` は引数に `x` を足し合わせる関数となっている。またこのように想定された数より少ない引数を適用することを部分適用と呼ぶ。

1.2 多相型と多相関数

関数型 `a -> b` は単一の型ではなく、任意の型 `a` , `b` に対して個別に定義できるのであった。このように既存の型から生成できる型を多相型という。また、多相型から多相型への型を考えることもできる。例えば高階関数と多変数関数を用いれば、関数を合成する操作に次のような型を与えることができる。

```
apply :: (Int -> Int) -> Int -> Int
apply f x = f(x)
-- >>> apply double 10
-- 20
```

しかし、引数の関数は `Int` か `Int` への関数に制限されているが、実際は関数がどのような型であっても、適用の操作は行えるはずである。そこで、 `apply` 関数の引数の型を `a` 、返り値の型を `b` に置き換える。

```
apply :: (a -> b) -> a -> b
apply f x = f(x)
-- >>> apply double 10
-- 20
-- >>> apply not False
-- True
```

例のために真理値を反転する関数 `not :: Bool -> Bool` を使用している。

このように関数の引数の型を任意の型 `a` , `b` に置き換えることで、様々な型の値に対応した関数を定義することができ、このような関数を多相関数と呼ぶ。

このような多相関数は基本的に既存の多相関数の組み合わせによって定義される。この例であれば、`apply` の定義に用いた `f(x)` はすでに haskell 側によって多相関数 `($) :: (a -> b) -> a -> b` としてすでに定義されているため、定義することができた。

値の適用が関数として与えられているように、関数の合成も関数で行うことができる。つまり関数 `double` の定義に用いた `.` が関数であり、

`(.) :: (b -> c) -> (a -> b) -> a -> c`

という型が与えられているということである。

このように関数型プログラミングでの処理の記述は変数の仲介を除けば、関数に関数を合成する操作や、関数に値を適用する操作によって行われる。

1.3 基本的なデータ型

任意の型 `a` , `b` に対して多相型である直積型 `(a,b)` は以下の関数で構成される。

```
(,) :: a -> b -> (a, b)
fst :: (a, b) -> a
snd :: (a, b) -> b

-- >>> (,) 4 5
-- (4,5)
-- >>> fst (4,5)
-- 4
-- >>> snd (4,5)
-- 5
```

実行例を見て分かるように、`(a,b)` の値はある型 `a` と `b` の値をそれぞれ持つ。また、型 `a` , `b` の値 `x` , `y` に対して、

$$\texttt{fst (x, y)} = \texttt{x}, \quad \texttt{snd (x, y)} = \texttt{y}$$

が成り立つ。

直積型は二つの型の値を持つが、ゼロ個の型の値を持つ型も存在する。この型を Unit 型 `()` といい、ただ一つの値 `()` を持つ。

任意の型 `a` , `b` に対して多相型である直和型 `Either a b` は以下の関数で構成される。

```
Right :: b -> Either a b
Left :: a -> Either a b
either :: (a -> c) -> (b -> c) -> Either a b -> c

isZero :: Int -> Bool
isZero x = x==0

-- >>> either isZero not (Left 0)
-- True
-- >>> either isZero not (Right True)
-- False
```

`isZero` 関数の `x==0` は両辺が等しいか真理値を返す関数 `(==) :: Int -> Int -> Bool` であり、定義から分かるように `Either a b` は `a` か `b` の値のどちらか一方を持つ。

`either` 関数についてこの例では、`Int -> Bool` と `Bool -> Bool` を受け取っているため、次の引数の型は `Either Int Bool` を受け取る。もし `Either Int Bool` の値が `Int` であれば、`Int -> Bool` 型の `isZero` 関数に適用し、`Bool` であれば `Bool -> Bool` 型の `not` 関数に適用する、という関数である。

また任意の型 `x` とその値 `y` 、

```
f :: a -> x
g :: b -> x
```

に対して、

$$\texttt{either f g (Left x)} = \texttt{f x}$$
$$\texttt{either f g (Right x)} = \texttt{g x}$$

が成り立つ。

一般に直和を用いて新しい型を定義する場合、`Either`型を用いて定義するのではなく、haskell における多相型を定義する構文によって行われることが多い。また`either`関数もパターンマッチと呼ばれる構文によって行われる。

```
data EitherIB = Value Int | Judge Bool
calc :: EitherIB -> Bool
calc (Value a) = isZero a
calc (Judge a) = not a
-- >>> :t Value
-- Value :: Int -> EitherIB
-- >>> :t Judge
-- Judge :: Bool -> EitherIB
```

この例では、`EitherIB`が`Either Int Bool`に該当し、`Value`が`Left`、`Judge`が`Right`に対応する。

1.4 デザインパターンの例

最後にこれらを応用した Maybe 型を見る。

```
data Maybe a  =  Nothing | Just a
```

`Maybe a`は`Either`型を用いると、`Either () a`と書ける。ここでの`Nothing`は`Unit`型の値を与える関数とみなせるが、値はただ一つであるため単に定数と見て良い。すなわち`Maybe a`はただ一つの値`Nothing`と型`a`の値のどちらかの値を持つ型である。

この`Maybe`の想定している用途として、以下の例を扱う。

```
div10 :: Float -> Float
div10 = (10 /)
-- >>> div10 4
-- 2.5
-- >>> div10 0
-- Infinity
```

`div10`は 10 を与えられた数で割る関数であるが、結果のように零除算については Infinity という値が与えられる。これを素直に計算の失敗と見なし、型によって計算結果が正しく出力されないことを示したい。そこで、

```
div10' :: Float -> Maybe Float
div10' x = if x == 0 then Nothing else Just $ div10 x

-- >>> div10' 4
-- Just 2.5
-- >>> div10' 0
-- Nothing
```

という関数を定義する。これは零除算が発生する場合 `Nothing` を返し、そうでない場合は `Just` によって計算結果を `Maybe` に包んで返している。

これによって、 `Maybe Float` の値を用いて計算を行うときは、必ず `Nothing` と `Just` の場合分けが必要になり、他の言語でいう Null チェックが必要なことを型レベルで示していることになる。

またここではこれ以上語らないが、 `Maybe` 型の挙動はモナドと呼ばれるデザインパターンによって一般化される。これによって特に `a -> Maybe b` 型の関数と `b -> Maybe c` 型の関数の自然な合成を与えることができる。

2 圏論

関数型言語では関数という言葉が使われるように、型を属する値の集合、関数を値の集合から値の集合への写像とみなすことができる。集合を写像の視点から観察する場合、圏論によって一般化することが有効である場合が多い。そのため以降は型と関数の議論を圏論によってモデル化することを考える。

2.1 型と関数の圏

型を対象、関数を射とする圏 $\mathbb{H}$ を定義する。圏の定義の詳細は [CTI]2.0.1 を参照 Haskell には二つの関数が等しいことを判定する一般的な操作は無い。そのため関数の等号を改めて定義する。

定義 2.1.1（関数の外延性） 関数 `f :: a -> b` 、 `g :: a -> b` と型 `a` の任意の値 `x` において

$$\texttt{f x} = \texttt{g x} \iff \texttt{f} = \texttt{g}$$

定義 2.1.2（型と関数の圏）

対象 すべての型の集合を対象集合 $\mathrm{Obj}(\mathbb{H})$ とする。また各型を対象とみなす場合 $A, B, C \cdots$ と表記し、型とみなす場合は `a` , `b` , `c` $\cdots$ とする。また各型の値を圏論の文脈では $a, b, c \cdots$ と表記することにする。

射 対象 A から対象 B への射集合 $\mathbb{H}(A, B)$ を集合と見なした関数型 `a -> b` とする。また各射は $f, g, h \cdots$ と表記する。

射の合成 射の合成を行う写像 $\circ : \mathbb{H}(B, C) \times \mathbb{H}(A, B) \to \mathbb{H}(A, C)$ を任意の射 $f : A \to B,\ g : B \to C$ に対して `f.g` によって定義する。

恒等射の存在　任意の型 `a` に対して恒等関数 `id :: a -> a` は、 `id x = x` と定義できる。この恒等関数を恒等射とし、$id_A : A \to A$ と表記する。

結合律　関数の外延性より、集合の圏における結合律と同様の手法で証明できる。[CTI]2.0.3 を参照

単位元律　結合律と同様に [CTI]2.0.3 を参照

次に型の直積と直和の普遍性との関係を見る。

命題 2.1.3**（$\mathbb{H}$ の積）**　圏 $\mathbb{H}$ は積を持つ

証明 2.1.4　(`(a, b)` , `fst` , `snd`) は型 `a` , `b` における積であることを示せば良い。任意の型 `x` 、任意の関数

```
f :: x -> a
g :: x -> b
```

に対して、

```
h :: x -> (a, b)
h x = (f x, g x)
```

とする。すると明らかに

$$\texttt{fst . h \$ x} = \texttt{f x}$$
$$\texttt{snd . h \$ x} = \texttt{g x}$$

であり、2.1.1 の関数の外延性より

$$\texttt{fst . h} = \texttt{f}$$
$$\texttt{snd . h} = \texttt{g}$$

が成り立ち、 `h` は `f` 、 `g` の射の対であることが分かる。またある関数 `i :: x -> (a, b)` によって与えられた値 `i x :: (a, b)` が

$$\texttt{fst (i x)} = \texttt{f x}$$
$$\texttt{snd (i x)} = \texttt{g x}$$

を満たす時、 `fst` , `snd` の性質から `i x = (f x, g x)` が成り立つ。よって 2.1.1 の関数の外延性より `i` = `h` である。これにより射の対 `h` の一意に存在することを示せた。 よって (`(a, b)` , `fst` , `snd`) は積であり、任意の型 `a` , `b` に対して存在するから $\mathbb{H}$ は積を持つ。

命題 2.1.5**（$\mathbb{H}$ の終対象）**　圏 $\mathbb{H}$ は終対象を持つ。

証明 2.1.6　ある型 `x` において

```
f :: x -> ()
f x = ()
```

なる写像を考える。またこのような関数は任意の `x` について考えることができる。

```
g :: x -> ()
g x = ()
```

なる写像を考えると、`f x` = `()` = `g x` が成り立ち、2.1.1 の関数の外延性より `f` = `g` が成り立つ。`f :: x -> ()` なる関数が一意に存在するから `()` は $\mathbb{H}$ における終対象であり、$\mathbb{H}$ は終対象を持つ。

定義 2.1.7（余積の定義） ある対象 A, B に対して $(A+B,\ \iota_A,\ \iota_B)$ が余積であるとは、$\iota_A : A \to A+B,\ \iota_B : B \to A+B$ であり、任意の対象 X と任意の射 $f : A \to X,\ g : B \to X$ に対して

$$[f, g] \circ \iota_A = f,\ [f, g] \circ \iota_B = g$$

であるような $[f, g] : A + B \to X$ が一意に存在するときである。

命題 2.1.8（$\mathbb{H}$ の余積） 圏 $\mathbb{H}$ は余積を持つ、すなわち任意の対象 A, B に対して余積 $A + B$ が存在する。

証明 2.1.9 厳密に証明するのであれば自然同型

$$\mathbb{H}(A + B, -) \cong \mathbb{H}(A, -) \times \mathbb{H}(B, -)$$

を示すべきであるが、ここでは少し妥協して `Either a b` の値が `Left` , `Right` のみによって与えられるとする。

積と同様に (`Either a b` , `Left` , `Right`) が型 `a` 、 `b` における余積であることを示せば良い。任意の型 `x` 、任意の関数

```
f :: a -> x
g :: b -> x
```

に対して、

$$\texttt{(either f g) Left x} = \texttt{f x}$$
$$\texttt{(either f g) Right x} = \texttt{g x}$$

が成り立つのであった。よって、

$$\texttt{(either f g).Left} = \texttt{f}$$
$$\texttt{(either f g).Right} = \texttt{g}$$

であり、

$$\texttt{h.Left} = \texttt{f}$$
$$\texttt{h.Right} = \texttt{g}$$

を満たすような `h :: Either a b -> x` が存在するとする。すると、 `a` の任意の値 `x` と `b` の任意の値 `y` に対して、

$$\texttt{(either f g).Left x} = \texttt{h.Left x}$$
$$\texttt{(either f g).Right y} = \texttt{h.Right y}$$

が成り立つ。`Either a b`の値が`Left`, `Right`のみによって与えられるから、2.1.1 の関数の外延性より`either f g = h`であり、条件を満たす`either f g`が一意に定まることが分かった。

よって (`Either a b`, `Left`, `Right`) は`a`、`b`に対する余積であり、これが任意の型に対して存在するから $\mathbb{H}$ は余積を持つ

2.2 関手と自然変換

Haskell には他のオブジェクト指向言語におけるインターフェイスのような概念がある。これを型クラスといい、それを実装したものはインスタンスと呼ぶ。例えばある型`a`がシリアル化可能、つまり型`a`の値と文字列の相互変換が可能であるには`a -> String`と`String -> a`なるような二つの関数を持っていなければならない。ただし文字列を値とする型を`String`とする。そこでその条件を型クラス`Serializable`として記述する。

```
class Serializable a where
  serialize :: a -> String
  deserialize :: String -> a
```

真理値は単に`True False`を文字列と見なしたり、真理値と見なしたりできシリアル化可能であると述べることができる。そこで以下に示すように型`Bool`を型クラス`Serializable`のインスタンスにする。

```
instance Serializable Bool where
  serialize :: Bool -> String
  serialize x = if x then "True" else "False"
  deserialize :: String -> Bool
  deserialize x = x == "True"
```

ここで注意すべきことは本来シリアル化であれば String から値が完全に復元できるように

`serialize.deserialize x = x`, `deserialize.serialize x = x`

が成り立つように`serialize`と`deserialize`を定義するべきであるが、一般的に等式によってそのような制約を設けることはできない。そのため型クラスにそのような等式が存在する場合は、手動でそれを満たすかどうか調べなければならない。

また多相型は既存の型を新しい型に写す写像の印象が強く、一般の型とは大きく異なるような概念に見える。しかし関数の全体を型とみなすのと同様に、多相型もまた型の一種である。よって単に多相型によって写された型`f a`だけでなく、多相型`f`もインスタンス化が可能である。その例として`Functor`型を示そう。

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

`fmap`の型に`f a`が含まれるため、`f`は多相型であることが要請されることが分かる。また、関数`fmap`はファンクタ則と呼ばれる以下の等式を満たさなければならない。

`fmap id = id`

`fmap (f . g) = fmap f . fmap g`

この `Functor` という型クラスは、多相型が既存の型を新しい型へ写すような操作と見なした場合、型だけでなくその周りの関数も同様に写すことができることを要請する。

またファンクタ則については、新しい関数に写した場合、ある程度元の関数を引き継ぐよう要請するものである。

多相型と一般の型の関係が分かりにくいため、型の型であるカインドを紹介する。型を更に上位の型であるカインドによって分類するわけだが、一般の型に比べて複雑ではない。例えば多相型でない具体的な型 `Int`、`Bool` のカインドは `*` である。多相型 `Maybe` は `* -> *` であり、`*` の具体的な型を `*` の具体的な型に写す関数と見なしている。

`(a,b)` や `a -> b` は二つの任意の型から一つの型を与えていたが、これも多変数関数と同様に `(,) :: * -> * -> *` というようなカインドが与えられている。

```
-- >>> :k Int
-- Int :: *
-- >>> :k Bool
-- Bool :: *
-- >>> :k Maybe
-- Maybe :: * -> *
-- >>> :k (,)
-- (,) :: * -> * -> *
-- >>> :k (->)
-- (->) :: * -> * -> *
```

さて Functor の話に戻ると、この名前の通りこれは圏 $\mathbb{H}$ における自己関手にあたる。関手の定義は [CTI]6.0.1 を参照。

命題 2.2.1　多相型 `f` が `Functor` のインスタンスである時、`f` は $\mathbb{H}$ における自己関手である。

対象関数　`a` を `f a` に写す操作を対象関数とする。

射関数　`fmap :: (a -> b) -> f a -> f b` を射関数とする。
`fmap :: (a -> b) -> (f a -> f b)` とみなすことができるため、2.1.2 の圏 $\mathbb{H}$ の射集合の定義より、$\mathbb{H}(A, B) =$ `a -> b` であったから、`fmap` は射関数とすることができる。

恒等射の保存　正しくファンクタ則の `fmap id` $=$ `id` が恒等射の保存である。

射の合成の保存　正しくファンクタ則の `fmap (f . g)` $=$ `fmap f . fmap g` が射の合成の保存である。

これから `Functor` のインスタンスを紹介していくが、ファンクタ則を満たすかどうかはここでは証明しない。興味があれば [CTI]6 章を読んでほしい。

対象 B を対象 $A \times B$ に写す操作は関手 $A \times - : \mathbb{H} \to \mathbb{H}$ であったから、同様に `(,) a :: * -> *` となる多相型も `Functor` のインスタンス化ができる。この `(,) a` という型は、多相型 `(,)` を多変数関数と見なしたときに、型 `a` を部分適用して得られた多相型である。

```
instance Functor ((,) a) where
  fmap f (x,y) = (x, f y)
```

`Functor` の求める型のカインドは `* -> *` であるが、これを `* -> * -> *` に拡張した `Bifunctor` が存在する。

```
class Bifunctor p where
  bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
  bimap f g = first f . second g

  first :: (a -> b) -> p a c -> p b c
  first f = bimap f id
  second :: (b -> c) -> p a b -> p a c
  second = bimap id
```

`Bifunctor` 型クラスでは必要とされる関数にすでに実装がされていて、`first`、`second` は `bimap` を参照、`bimap` は `first`、`second` を参照している。これはどちらか一方を実装すれば、もう片方も自動的に定義されるということである。詳細は述べないが、`second` が `Functor` 型クラスの実装である `fmap` に該当し、`first` が `Functor` で実装しきれなかった方の関手性である。これは正しく圏論における双関手であり、詳しくは [CTI]6.3.1 を参照してほしい。

直積型である `(a, b)` と直和型である `Either a b` も `Bifunctor` のインスタンスであり、

```
instance Bifunctor (,) where
  bimap f g ~(a, b) = (f a, g b)
instance Bifunctor Either where
  bimap f _ (Left a) = Left (f a)
  bimap _ g (Right b) = Right (g b)
```

のように実装されている。直感的には `(a, b)` 型と `Either a b` 型の `a` と `b` の値にそれぞれ関数を適用しているという点で、一種の値の並列計算のように思える。

また、直積と直和が関手的であることから、それらで構成される `Maybe` 型も `Functor` の実装を自然に与えることができる。実際デフォルトの実装は

```
instance Functor Maybe where
  fmap _ Nothing       = Nothing
  fmap f (Just a)      = Just (f a)
```

であるが、`Either` 型によって定義された `Maybe` 型の場合は

```
instance Functor (Either () a) where
  fmap f = bimap id f
```

のように `Either` 型の双関手性を用いて定義できる。`Maybe` 型による `fmap` の用途としては、値 `m :: Maybe a` に対して関数 `f :: a -> b` を適用できるという点にあるだろう。

すなわち `(fmap f) m :: Maybe b` である。`fmap` の定義にある通り、`m` が `Nothing` である時何も行わず、`Just n` であった時 `f` に `n` を適用するという操作を行うが、`fmap` によって面倒な場合分けを自動で行われる点が便利である。

さらに最初に多相型の例として挙げた関数型もまた `Functor` 型のインスタンスである。内容としては [CTI]6.4.5 と同じであるが、

```
instance Functor ((->) r) where
  fmap = (.)
```

と簡潔に定義される

`fmap` の型は `(a -> b) -> f a -> f b` であったから、この場合は `(a -> b) -> (r -> a) -> (r -> b)` となる。よって型は合致している。これを関手の観点から見ると、ある関数 `f :: a -> b` を用いて部分適用を行うと `(.) f :: (r -> a) -> (r -> b)` が得られるが、これは与えられた関数に `f` を合成する関数である。

`(->)` も `(,)` と同様に `* -> * -> *` なるカインドを持つから、`Bifunctor` のインスタンスと考えるかもしれないがそうはならない。特に `first :: (a -> b) -> p a c -> p b c` に該当する関数が存在しないためである。これに `(->)` を当てはめると、`second :: (a -> b) -> (a -> c) -> (b -> c)` となるが、`a -> b` と `a -> c` の関数に対する一般的な合成は特に定義されていない。しかし `Bifunctor` の代わりに `(->)` は `Profunctor` と呼ばれる型クラスのインスタンスである。

次に自然変換について説明する。Haskell において自然変換は単なる多相関数によって表される。すなわち、`Functor` 型クラスのインスタンスである多相型二つの間の多相関数であり、自然性と呼ばれる等式を満たすならば自然変換である。

この自然性というのは、`Functor` である `f, g` とその間の多相関数 `n :: f x -> g x` と関数 `g :: a -> b` に対して、

$$\texttt{fmap g.n} = \texttt{n.fmap g}$$

が成り立つことである。自然変換についても詳細は [CTI]7.0.1 を参照

また Haskell では型 `a` を同じ型 `a` に写すような多相型は構成できないが、これが存在すると見なし `fmap` 関数に恒等関数を割り当てることで `Functor` 型クラスのインスタンスとみなす。

また、[CTI]5.3.6、[CTI]5.3.7 の評価射、余評価射の定義に従って、関数 `ev`、`ce` を定義する

```
ev :: (b -> a, b) -> a
ev p = fst p $ snd p
ce :: a -> (b -> (a, b))
ce = (,)
```

これも [CTI]7.0.9、[CTI]7.0.11 より、型 `a` に対して自然になる。

また自然変換は命題 [CTI]7.5.1 のような普遍性を持ち、それは定義 [CTI]8.0.9 のエンドと呼ばれる対象に一般化されるのであった。これを Haskell で記述すると

```
{-# LANGUAGE RankNTypes #-}
data End p = End {prj :: forall a. p a a}
-- >>> :t End
```

```
-- End :: (forall a. p a a) -> End p
-- >>> :t prj
-- prj :: End p -> p a a
```

このように定義できる。ただし、ファンクタ則や自然性と同じようにエンドの持つ普遍性は記述できない。またこの定義では`p a b`が共変関手、反変関手として振る舞うことを記述できていないが、これは`p a b`が前に述べた`Profunctor`型クラスのインスタンスであるように制約を持たせれば良い。また自然変換の全体はエンドであったから、`p a a`=`f a -> g a`=`(Mor f g) a a`としてエンドによって定義することができる。

```
data Nat f g a b = Nat{component :: f a -> g b}

toMaybe :: Either b a -> Maybe a
toMaybe = either (const Nothing) Just

toMaybeNat :: End (Nat (Either b) Maybe)
toMaybeNat = End (Nat toMaybe)

-- >>> :t component (prj toMaybeNat)
-- component (prj toMaybeNat) :: Either b1 b2 -> Maybe b2
```

この例では多相関数`toMaybe`をエンド`toMaybeNat`に変換している。また

```
mor.prj :: End (Mor f g) -> f b -> g b
```

によって元の多相関数へと復元することもできる。

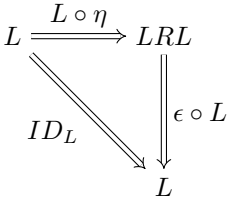
2.3 関数型とべき随伴

定義 2.3.1（随伴関手） ある二つの関手 $L : \mathbb{C} \to \mathbb{D}$、$R : \mathbb{D} \to \mathbb{C}$ が随伴関手であるとは、以下の性質を満たす時である。

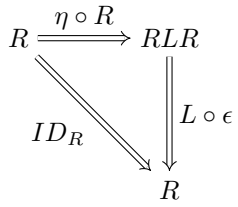
単位と余単位 ある自然変換 $\eta : Id_{\mathbb{C}} \Rightarrow R \circ L$ と $\epsilon : L \circ R \Rightarrow Id_{\mathbb{D}}$ が存在する。また η を単位、ϵ を余単位と呼ぶことにする。

三角恒等式 自然変換の二等式

$$(\epsilon \circ L) \cdot (L \circ \eta) = ID_L$$



$$(R \circ \epsilon) \cdot (\eta \circ R) = ID_R$$



が成り立つ。

また $L : \mathbb{C} \to \mathbb{D}$ が左随伴関手、$R : \mathbb{D} \to \mathbb{C}$ が対応する右随伴関手である時、$L \dashv R$ と表記する

命題 2.3.2**（べき随伴）** `(, b)` $\dashv$ `(->) b` である。

ただし、`(, b)` は `Functor` 型クラスのインスタンスではないが `Bifunctor` 型クラスの `first` 関数を `fmap` 関数とすることで `Functor` のインスタンスとみなす。

証明 2.3.3 単位を多相関数 `ce`、余単位を `ev` とする。これらの多相関数や多相型が関手、自然変換であることは確認したから三角恒等式を満たすことを調べれば良い。1 つ目の等式は

$$\texttt{ev.(first ce)} = \texttt{id}$$

と表される。しかし添字が省略されてわかりにくいため、各々の関数の型を示しておく。

```
ce :: a -> (b -> (a, b))
fst ce :: (a, b) -> ((b -> (a, b)), b)
ev.(first ce) :: (a, b) -> (a, b)
```

これに `(a, b)` 型の任意の値 `(x, y)` を適用すると、

$$\begin{aligned}\texttt{(ev.(first ce))(x, y)} &= \texttt{ev(first ce (x,y))} \\ &= \texttt{ev((,) x, y)} \\ &= \texttt{(,) x y} \\ &= \texttt{(x, y)}\end{aligned}$$

となり、`ev.(first ce)` = `id` が成り立つ。同様に 2 つ目の等式は

$$\texttt{(fmap ev).ce} = \texttt{id}$$

である。各関数の型は以下のようになっていて、

```
ev :: (b -> a, b) -> a
fmap ev :: (b -> (b -> a, b)) -> (b -> a)
(fmap ev).ce :: (b -> a) -> (b -> a)
```

`(b -> a)` 型の任意の値 f を適用すると、

$$\begin{aligned} \texttt{((fmap ev).ce)f} &= \texttt{(fmap ev)(ce f)} \\ &= \texttt{(fmap ev)((,) f)} \\ &= \texttt{ev.((,) f)} \end{aligned}$$

ここで型 `b` の任意の値 `y` に対して、

$$\begin{aligned} \texttt{(ev.((,) f)) y} &= \texttt{ev(f, y)} \\ &= \texttt{f(y)} \end{aligned}$$

であるから、`ev.((,) f)` = `f` である。よって `(fmap ev).ce` = `id` となる。

定義 2.3.4**（カルテジアン閉圏）**　圏 $\mathbb{C}$ がカルテジアン閉圏 (ccc) であるとは、積と終対象を持ち、積関手 $(- \times B) : \mathbb{C} \to \mathbb{C}$ に対して右随伴となるような関手 $(-)^B : \mathbb{C} \to \mathbb{C}$ を持つということである。

命題 2.3.5　圏 $\mathbb{H}$ は ccc である。

証明 2.3.6　命題 2.1.3、2.1.5、2.3.2 より、圏 $\mathbb{H}$ は ccc である。

また随伴関手の同値な定義より、

命題 2.3.7**（随伴の射集合同型）**

$$\mathbb{H}(A \times B, C) \cong \mathbb{H}(A, C^B)$$

であり、A, B, C に対して自然である。

B に対して自然であることは、単位、余単位が B について超自然と呼ばれる一般化された自然性を持つことによって示される。またカルテジアン閉圏の性質より、上の命題は射集合ではなく冪で述べることができるようになる

命題 2.3.8

$$C^{A \times B} \cong (C^B)^A$$

であり、A, B, C に対して自然である。

証明 2.3.9　米田の原理より、$A \cong B \iff \mathbb{C}(-, A) \cong \mathbb{C}(-, B)$ であるから、$\mathbb{H}(X, C^{A \times B}) \cong \mathbb{H}(X, (C^B)^A)$ であり X に対して自然であることを示せば良い。

$$\begin{aligned} \mathbb{H}(X, C^{A \times B}) &\cong \mathbb{H}(X \times (A \times B), C) && \text{(随伴の射集合同型)} \\ &\cong \mathbb{H}((X \times A) \times B, C) && \text{(積の結合則)} \\ &\cong \mathbb{H}(X \times A, C^B) && \text{(随伴の射集合同型)} \\ &\cong \mathbb{H}(X, (C^B)^A) && \text{(随伴の射集合同型)} \end{aligned}$$

計算に使用した同型は X, A, B, C において自然であるから、$C^{A \times B} \cong (C^B)^A$ が自然同型になる。

またこの命題における積を終対象に置き換えたものは以下のようになる。

命題 2.3.10　$A^1 \cong A$ であり A に対して自然

証明 2.3.11

$$\begin{aligned} \mathbb{H}(X, A^1) &\cong \mathbb{H}(X \times 1, A) && \text{(随伴の射集合同型)} \\ &\cong \mathbb{H}(X, A) && \text{(積の単位元則)} \end{aligned}$$

計算に使用した同型は X, A に対して自然であるから、$A^1 \cong A$ であり A に対して自然

また右随伴関手は連続であるから直ちに以下の同型が成り立つ。

命題 2.3.12

$$(A \times B)^C \cong A^C \times B^C$$

$$1^A \cong A$$

また詳細は省くが、$\mathbb{H}$ は「対称」モノイダル閉圏であるから、

$$A^{(-)} \dashv (A \times -)$$

が成り立つ。重要なことは $A^{(-)} : \mathbb{H}^{\mathrm{op}} \to \mathbb{H}$ が左随伴関手になり、余連続であるということである。これによって次の命題が成り立つ。

命題 2.3.13

$$C^{A+B} \cong C^A \times C^B$$

最後にこれらの同型の同型射を Haskell で定義する。

```
curry :: ((a, b) -> c) -> a -> b -> c
curry = (.coeval).fmap

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry = (eval.).first

eval :: (b -> a, b) -> a
eval p = fst p $ snd p
coeval :: a -> (b -> (a, b))
coeval = (,)
```

べき随伴の射集合同型の同型射 `curry` と `uncurry` は以前定義した、べき随伴における単位余単位によって定義できる。また `(.coeval)` の `.` は関数の合成ではなく、関数型をファンクタと見なした時の `fmap` として用いている。

```
outProduct :: (a -> (b, c)) -> (a -> b, a -> c)
outProduct f = (fst.f, snd.f)
```

```
inProduct :: (a -> b, a -> c) -> a -> (b, c)
inProduct (f, g) a = (f a, g a)

outEither :: (Either a b -> c) -> (a -> c, b -> c)
outEither f = (f.Left, f.Right)
inEither :: (a -> c , b -> c) -> Either a b -> c
inEither (f, g) = either f g
```

また左右の冪随伴における余積、積の保存の同型もこのように定義できる。

3 Optics

3.1 Lens と Prism

Optics の例として Lens、Prism を挙げる。

Lens はオブジェクト指向におけるフィールドと、それに対するセッター、ゲッターを一般化した概念である。例えばある型 S が型 A を内部に保持する時、A に関する操作としては二つの関数 $set : S \times A \to S$、$get : S \to A$ が考えられる。直感的には set は S の値 s と A の値 a を受け取って、s の内部の A の値を a に置き換えた、新しい S の値を返す関数である。また get は S の値 s の内部の A の値を返す関数である。

例として `Person` 型の値が型の直積によって `String` 型の値を持つとする。この `Person` の値の中の文字列を set、get を用いて記述する

```
type Person = (String, Int)

set :: (Person, String) -> Person
set (p, s) = (s, snd p)
get :: Person -> String
get = fst

person :: Person
person = ("A",20)
-- >>> get person
-- "A"
-- >>> set (person, "B")
-- ("B",20)
```

ここでは直積型を用いて S の値が A の値を持つことを明示的に示したが、一般の場合ではそうは行かない。そのため set, get 関数による等式で間接的に示す。

$$\pi_A = get \circ set : S \times A \to A$$

$$id_S = set \circ \langle id_S, get \rangle : S \to S \times A$$

$$set \circ (set \times id_A) = put \circ (\pi_S \circ \pi_{S \times A} \times \pi_A) : (S \times A) \times A \to S$$

1つ目の等式は値をその中の A の値によって set で置き換えても S の値は変わらないことを示す。2つ目は set によって与えた a の値が、get によって取り出せることを示し、3つ目はある s に a を置き換えた時、その前の a が上書きされることを示している。

次に Lens の一種の双対である Prism を紹介する。これは部分型としての包含関係を表す概念であり、二つの関数 $review : A \to S$、$match : S \to S + A$ で構成される。この場合 A から S への値にアップキャストする操作は $review$ 関数により容易に行えて、S から A への値のダウンキャストは可能であれば A の値、不可能であればそのまま S の値を返す操作とみなせる。例えば、整数の内の奇数のみを持つ型 `Odd` を考え、`Integer` 型との包含関係を $review$、$match$ で記述する

```
newtype Odd = Odd{toInt :: Integer} deriving Show
match :: Integer -> Either Integer Odd
match x = if odd x
  then Right (Odd x)
  else Left x
review :: Odd -> Integer
review = toInt

-- >>> match 4
-- Left 4
-- >>> match 3
-- Right (Odd {toInt = 3})
-- >>> review (Odd 5)
-- 5
```

Lens と同様に Prism でも、包含関係を間接的に示すのに $review$, $match$ 関数を用いた等式で示す。また等式における $[id_S, review]$ は積における射の対の双対である。

$$match \circ review = \iota_A : A \to S + A$$

$$[id_S, review] \circ match = id_S : S \to S$$

1つ目の等式は A の値 a をアップキャストをしてからダウンキャストを行うと、$S + A$ における A の方の値が必ず得られて、それが a を $S + A$ に入射したものと一致することを示している。2つ目の等式は S の値 s が $match$ によってダウンキャストが成功した場合、更にアップキャストを行なった値は s と一致し、ダウンキャストが失敗した場合も s と一致することを示している。

ここで Lens の $set : S \times A \to S$ と $get : S \to A$ の組の全体を $Lens(S, A)$、Prism の $review : A \to S$ と $match : S \to S + A$ の組の全体を $Prism(S, A)$ と表記することにしよう。これは Haskell においてはそのような多相型を定義することである。

```
data Lens s a = Lens{get :: s -> a , set :: (s, a) -> s}
data Prism s a = Prism{review :: a -> s , match :: s -> Either s a}
-- >>> :t Lens
-- Lens :: (s -> a) -> ((s, a) -> s) -> Lens s a
```

```
-- >>> :t Prism
-- Prism :: (a -> s) -> (s -> Either s a) -> Prism s a
```

また $Lens(S, A)$ における get 関数を $get_{S,A}$ と表記する。また set 関数や Prism においても同様に表記する。

これによって Lens の合成 $\circ : Lens(S, A) \times Lens(T, S) \to Lens(T, A)$ が定義を行う。

定義 3.1.1

$$\circ : Lens(S, A) \times Lens(T, S) \to Lens(T, A)$$

なる関数 $\circ$ を任意の $set_{T,S} : T \times S \to T, get_{T,S} : T \to S,\ set_{S,A} : S \times A \to S,\ get_{S,A} : S \to A$ に対して

$$get_{T,A} = get_{S,A} \circ get_{T,S}$$
$$set_{T,A} = set_{T,S} \circ (id_T \times set) \circ a_{TSA} \circ (\langle id_T, get_{T,S} \rangle \times id_A)$$

と定義する。

$$T \times A \xrightarrow{\langle id_T, get_{T,S} \rangle \times id_A} (T \times S) \times A \xrightarrow{a_{TSA}} T \times (S \times A) \xrightarrow{id_T \times set_{S,A}} T \times S \xrightarrow{set_{T,S}} T$$

$set_{T,A}$ の定義が複雑かもしれないが、単に T から S を取り出し、S に A を与えてその S をまた T に戻しているだけである。

また Lens の合成において恒等射のような働きをする Lens も定義できる。

定義 3.1.2**（恒等** Lens**）**

$$get_{S,S} = id_S$$
$$set_{S,S} = \pi_{L,S \times S}$$

Prism においても Lens と同様に合成が定義できる。

3.2 Optics への一般化

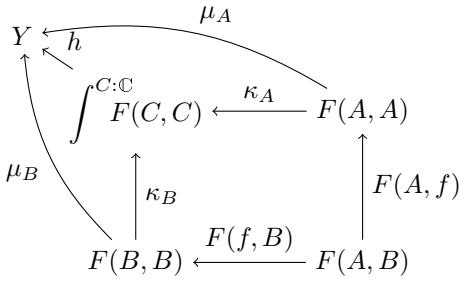
Optics を構成するための圏論の概念としてコエンドを定義する。

定義 3.2.1**（コエンド）**　ある圏 $\mathbb{C}, \mathbb{D}$ と、関手 $F : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{D}$ に対するコエンド $(\int^{C:\mathbb{C}} F(C, C), \kappa)$ を以下のように構成する。

余楔　余楔と呼ばれる組 (Y, μ) を、圏 $\mathbb{D}$ のある対象 Y と、圏 $\mathbb{C}$ の任意の対象 X に対して $\mu_C : Y \to F(X, X)$ なる射が存在し、圏 $\mathbb{C}$ の任意の射 $f : B \to A$ に対して $\mu_A \circ F(A, f) = \mu_B \circ F(f, B)$ が成り立つような μ によって構成する。

$$\begin{array}{ccc} \mathrm{Y} & \xleftarrow{\mu_A} & F(A, A) \\ \uparrow{\scriptstyle \mu_B} & & \uparrow{\scriptstyle F(A, f)} \\ F(B, B) & \xleftarrow{F(f, B)} & F(A, B) \end{array}$$

普遍性　ある余楔 $(\int^{C:\mathbb{C}} F(C,C), \kappa)$ が F に対してコエンドであるとは、余楔 (Y, μ) が存在して、任意の対象 X において $\mu = h \circ \kappa_X$ が成り立つような $h : \int^{C:\mathbb{C}} F(C,C) \to Y$ が一意に存在する時である。また、κ を余積の場合と紛らわしくない場合は入射と呼ぶことにする。



コエンドの普遍性は $\int^{C:\mathbb{C}} F(C,C)$ が圏 C の任意の対象に対する $F(C,C)$ の余積であることを示していて、$F(A,B)$ の元に対し f を A と B のどちらに適用しても入射によって示されるということである。

余積という部分をもう少し詳しく見ると、$F(A,A)$ から Y への射の族 μ は同様の射の族 κ によって単なる射 h へと分解される。同様に κ が他の単なる射によって分解されるかといえば、コエンドの普遍性からそのような射の族は κ 以外には存在しないと言える。そういった意味で $(\int^{C:\mathbb{C}} F(C,C), \kappa)$ は余楔の中で、射の族としての最小限の性質のみを持っていると考えられる。

コエンドも Haskell で定義できる

```
{-# LANGUAGE ExistentialQuantification #-}
data Coend p = forall a. Coend (p a a)
-- >>> :t Coend
-- Coend :: p a a -> Coend p
```

`Coend` の値コンストラクタを見ると、`p a a` から `Coend p` へ写る過程で型引数 `a` の情報を忘れてしまっている。そのため一度 `Coend` に写してしまうと、元の型に復元することはできなくなり、一般的には一切の操作が行えなくなってしまう。そういった意味でこれは任意の型 `a` で添字付けられた `p a a` の直和であるように思える。

コエンドに限らずこういった定義の手法を存在量化と呼ぶが、実用の面では最低限操作が行えることを保証するために `a` の型に制約を加えることが多い。

次にコエンドとエンドの関係性についての重要な命題を示す。

命題 3.2.2

$$\mathbb{Set}(\int^{C:\mathbb{C}} T(C,C), X) \cong \int_{C:\mathbb{C}} \mathbb{Set}(T(C,C), X)$$

であり、X, T に対して自然。

証明 3.2.3　反変 Hom 関手は余連続であるから余極限を保つ。またエンドは極限によって定義できるように、コエンドも余極限によって定義できる。これによってコエンドは反変 Hom 関手によって保たれるが、反変性よりコエンドの双対であるエンドとして保たれる。

定義 3.2.4（Optics）　対称モノイダル圏 $\mathbb{C}$ における組 (S, S')、(A, A') の間の Optic を

$$Optic_{\mathbb{C}}(S, S', A, A') = \int^{M:\mathbb{C}} \mathbb{C}(S, M \otimes A) \times \mathbb{C}(M \otimes A', S')$$

と定義する。

この定義におけるコエンドの普遍性は、$\mathbb{C}(S, M \otimes A) \times \mathbb{C}(M \otimes A', S')$ の元 $\langle l, r \rangle$ に対し、

$$\kappa_N((f \otimes A) \circ l, r) = \kappa_M(l, r(f \otimes A'))$$

が成り立つような κ を入射とする $\mathbb{C}(S, M \otimes A) \times \mathbb{C}(M \otimes A', S')$ の余積である。次に Optics が Lens、Prism の一般化であることを示したいが、そのために余米田の補題が必要であるためここで示す。

定義 3.2.5（余米田の補題）　任意の圏 $\mathbb{C}$ と対象 A と関手 $F : \mathbb{C} \to \mathbb{S}\mathrm{et}$ において

$$FA \cong \int^{C:\mathbb{C}} FC \times \mathbb{C}(C, A)$$

またここでの積分記号はエンドではなくコエンドである。

証明 3.2.6　米田の原理より、

$$A \cong B \iff \mathbb{C}(A, -) \cong \mathbb{C}(B, -)$$

であるから、$\mathbb{S}\mathrm{et}(FA, X) \cong \mathbb{S}\mathrm{et}(\int^{C:\mathbb{C}} FC \times \mathbb{C}(C, A), X)$ が X に対して自然に成り立つことを示せば良い。またコエンドは関手であるから、同型を保つ。

$$\begin{aligned}
\mathbb{S}\mathrm{et}(\int^{C:\mathbb{C}} FC \times \mathbb{C}(C, A), X) &\cong \int_{C:\mathbb{C}} \mathbb{S}\mathrm{et}(FC \times \mathbb{C}(C, A), X) && \text{(反変 Hom 関手の余極限の保存)} \\
&\cong \int_{C:\mathbb{C}} \mathbb{S}\mathrm{et}(\mathbb{C}(C, A) \times FC, X) && \text{(積の交換)} \\
&\cong \int_{C:\mathbb{C}} \mathbb{S}\mathrm{et}(\mathbb{C}(C, A), \mathbb{S}\mathrm{et}(FC, X)) && \text{(べき随伴の射集合同型)} \\
&\cong \mathbb{S}\mathrm{et}(FA, X) && \text{(反変米田の補題)}
\end{aligned}$$

よってである。計算のために使用した同型はすべて X, F, A に対して自然であるから、$FA \cong \int^{C:\mathbb{C}} FC \times \mathbb{C}(C, A)$ が成り立ち、F, A に対して自然である。

また同様の方法で F に反変関手を取る場合も証明できる。すなわち、任意の関手 $F : \mathbb{C}^{\mathrm{op}} \to \mathbb{S}\mathrm{et}$ と対象 A に対して

$$FA \cong \int^{C:\mathbb{C}} FC \times \mathbb{C}(A, C)$$

が成り立つ。

余米田の補題によれば、FA が $FC \times \mathbb{C}(C, A)$ を与える関手のコエンドとなるということであるが、具体的な入射の構成は不明である。余楔を用いた証明も思いつくが、$\mathbb{H}$ が余積を持つことの証明のようにコエンドの元を列挙できないため、この方法では難しい。そのため厳密な証明では無いが直接同型射を構成して証明しよう。

命題 3.2.7　$\int^{C:\mathbb{C}} FC \times \mathbb{C}(C, A)$ の任意の元 x は入射 κ によって与えられるとする。すなわち $x = \kappa_B(b, f)$ となるようなある対象 B とある元 b、ある射 $f : B \to A$ の対 $\langle b, f \rangle$ が一意に存在するとする。

証明 3.2.8　同型射 $\phi : FA \to \int^{C:\mathbb{C}} FC \times \mathbb{C}(C, A)$、$\phi^{-1} : \int^{C:\mathbb{C}} FC \times \mathbb{C}(C, A) \to FA$ を以下のように定義する

$$\begin{aligned} \phi(a) &= \kappa_A(a, id_A) \\ \phi^{-1}(x) &= \phi^{-1}(\kappa_B(b, f)) = Ffb \end{aligned}$$

すると、

$$\begin{aligned} \phi \circ \phi^{-1}(\kappa_B(b, f)) &= \phi(Ffb) \\ &= \kappa_A(Ffb, id_A) \\ &= \kappa_B(b, f) && \text{(余楔の定義)} \\ \phi^{-1} \circ \phi(a) &= \phi^{-1}(\kappa_A(a, id_A)) \\ &= a \end{aligned}$$

となるから、$\phi \circ \phi^{-1} = id,\ \phi^{-1} \circ \phi = id_{FA}$ が成り立ち $FA \cong \int^{C:\mathbb{C}} FC \times \mathbb{C}(A, C)$ となる。

同型射を具体的な構成によって定義できたため、Haskell でも定義できるようになった。

```
{-# LANGUAGE ExistentialQuantification #-}
data Coend p = forall a. Coend (p a a)

data CoyonedaRaw f a c b = CoyonedaRaw (f c) (b -> a)
type Coyoneda f a = Coend (CoyonedaRaw f a)

toCoyoneda :: f a -> Coyoneda f a
toCoyoneda x = Coend (CoyonedaRaw x id)

fromCoyoneda ::Functor f => Coyoneda f a -> f a
fromCoyoneda (Coend(CoyonedaRaw x y)) = fmap y x
```

命題 3.2.9　$Optic(S, S', A, A')_{\mathbb{C},\times}$ を Optic の定義のテンソル積を積に置き換えたものとする。すると、

$$Optic(S, S, A, A)_{\mathbb{C},\times} = Lens(S, A)$$

である。

証明 3.2.10

$$\begin{aligned} \int^{M:\mathbb{C}} \mathbb{C}(S, M \times A) \times \mathbb{C}(M \times A, S) &\cong \int^{M:\mathbb{C}} \mathbb{C}(S, M) \times \mathbb{C}(S, A) \times \mathbb{C}(M \otimes A, S) && \text{(共変 Hom 関手の積の保存)} \\ &\cong \mathbb{C}(S, A) \times \mathbb{C}(S \times A, S) && \text{(余米田の補題)} \end{aligned}$$

同様に Prism も Optic で表せる。

命題 3.2.11

$$Optic(S,S,A,A)_{\mathbb{C},+} = Prism(S,A)$$

である。

証明 3.2.12

$$\begin{aligned}\int^{M:\mathbb{C}} \mathbb{C}(S, M+A) \times \mathbb{C}(M+A, S) &\cong \int^{M:\mathbb{C}} \mathbb{C}(S, M+A) \times \mathbb{C}(M, S) \times \mathbb{C}(A, S) && \text{(反変 Hom 関手の余積の保存)} \\ &\cong \mathbb{C}(S, S+A) \times \mathbb{C}(A, S) && \text{(余米田の補題)}\end{aligned}$$