4

Kleisli Categories

You've seen how to model types and pure functions as a category. I also mentioned that there is a way to model side effects, or non-pure functions, in category theory. Let's have a look at one such example: functions that log or trace their execution. Something that, in an imperative language, would likely be implemented by mutating some global state, as in:

```
string logger;

bool negate(bool b) {
    logger += "Not so! ";
    return !b;
}
```

You know that this is not a pure function, because its memoized version would fail to produce a log. This function has *side effects*.

4

Kleisli Categories

型と純粋関数をカテゴリとしてモデル化する方法について見てきました。また，副作用や非純粋関数をカテゴリ理論でモデル化する方法があることも紹介した．そのような例として、実行のログを取ったり、トレースしたりする関数を見てみよう。命令型言語では、次のようにグローバルな状態を変化させることによって実装されるでしょう。

```
string logger;

bool negate(bool b) {
    logger += "Not so! ";
    return !b;
}
```

これは純粋な関数ではありません。なぜなら、メモ化されたバージョンはログを生成できないからです。この関数には副作用があります。

In modern programming, we try to stay away from global mutable state as much as possible — if only because of the complications of concurrency. And you would never put code like this in a library.

Fortunately for us, it's possible to make this function pure. You just have to pass the log explicitly, in and out. Let's do that by adding a string argument, and pairing regular output with a string that contains the updated log:

```
pair<bool, string> negate(bool b, string logger) {
    return make_pair(!b, logger + "Not so! ");
}
```

This function is pure, it has no side effects, it returns the same pair every time it's called with the same arguments, and it can be memoized if necessary. However, considering the cumulative nature of the log, you'd have to memoize all possible histories that can lead to a given call. There would be a separate memo entry for:

```
negate(true, "It was the best of times. ");
```

and

```
negate(true, "It was the worst of times. ");
```

and so on.

It's also not a very good interface for a library function. The callers are free to ignore the string in the return type, so that's not a huge burden; but they are forced to pass a string as input, which might be inconvenient.

Is there a way to do the same thing less intrusively? Is there a way to separate concerns? In this simple example, the main purpose of the

現代のプログラミングでは、可能な限りグローバルな変更可能状態を避けようとします – 並行処理が複雑になるという理由だけであれば。そして、このようなコードをライブラリに入れることはないでしょう。幸いなことに、この関数を純粋なものにすることは可能です。ログを明示的にインとアウトで渡せばいいだけだ。文字列の引数を追加し、通常の出力と更新されたログを含む文字列を対にすることで、それを実現しましょう。

```
pair <bool , string > negate( bool b, string logger) {
    return make_pair(!b, logger + "Not so! ");
}
```

この関数は純粋で、副作用がなく、同じ引数で呼ばれるたびに同じペアを返し、必要ならメモ化することもできます。しかし、ログの累積的な性質を考慮すると、与えられた呼び出しにつながる可能性のあるすべての履歴をメモする必要があります。のメモ項目が別々に存在することになる。

```
negate(true, "It was the best of times. ");
```

and

```
negate(true, "It was the worst of times. ");
```

といった具合になります。また、ライブラリ関数のインターフェースとしては、あまり良いとは言えません。呼び出し側は戻り値の型に含まれる文字列を自由に無視できるので、大きな負担にはなりません。しかし、入力として文字列を渡さざるを得ないので、不便を感じるかもしれません。同じことをより押し付けがましくなく行う方法はないでしょうか？懸念を分離する方法はないだろうか？この単純な例では、主目的は

function `negate` is to turn one Boolean into another. The logging is secondary. Granted, the message that is logged is specific to the function, but the task of aggregating the messages into one continuous log is a separate concern. We still want the function to produce a string, but we'd like to unburden it from producing a log. So here's the compromise solution:

```
pair<bool, string> negate(bool b) {
    return make_pair(!b, "Not so! ");
}
```

The idea is that the log will be aggregated *between* function calls.

To see how this can be done, let's switch to a slightly more realistic example. We have one function from string to string that turns lower case characters to upper case:

```
string toUpper(string s) {
    string result;
    int (*toupperp)(int) = &toupper; // toupper is overloaded
    transform(begin(s), end(s), back_inserter(result), toupperp);
    return result;
}
```

and another that splits a string into a vector of strings, breaking it on whitespace boundaries:

```
vector<string> toWords(string s) {
    return words(s);
}
```

The actual work is done in the auxiliary function `words`:

関数negateの主な目的は、あるブール値を別のブール値に変えることです。ログの記録は二の次です。確かに、ログに記録されるメッセージは関数に特有のものですが、メッセージを1つの連続したログに集約するタスクは別の問題です。私たちは、この関数が文字列を生成することを望んでいますが、ログを生成することから解放されたいと思っています。そこで、妥協案として次のようなものがあります。

```
ペア <bool , string> negate( bool b) {…
    return make_pair(!b, "Not so! ");
}
```

このアイデアは、ログが関数呼び出しの間に集約されることです。これがどのように行われるかを見るために、もう少し現実的な例に切り替えてみましょう。小文字を大文字にする文字列から文字列への関数を一つ用意しました。

```
string toUpper(string s) {
    string result;
    int ( * toupperp)( int ) = & toupper; // toupperはオーバーロードされています。
    transform(begin(s), end(s), back_inserter(result), toupperp);
    return result;
}
```

また、文字列を文字列のベクトルに分割し、空白の境界でそれを分割するものもあります。

```
ベクトル < 文字列 > toWords(string s) { {…
    return words(s);
}
```

実際の作業は補助関数 `words` で行われます。

```
vector<string> words(string s) {
    vector<string> result{""};
    for (auto i = begin(s); i != end(s); ++i)
    {
        if (isspace(*i))
            result.push_back("");
        else
            result.back() += *i;
    }
    return result;
}
```

We want to modify the functions `toUpper` and `toWords` so that they piggyback a message string on top of their regular return values.

We will “embellish” the return values of these functions. Let’s do it in a generic way by defining a template `Writer` that encapsulates a pair whose first component is a value of arbitrary type `A` and the second component is a string:

```
template<class A>
using Writer = pair<A, string>;
```

Here are the embellished functions:

ベクトル < 文字列 > 言葉(文字列s) { ----------。
    vector<string> result{""};
    for (auto i = begin(s); i != end(s); ++i)
    {
        if (isspace(*i))
            result.push_back("");
        else
            result.back() += *i;
    }
    return result;
}

関数 `toUpper` と `toWords` を修正して、通常の戻り値の上にメッセージ文字列を載せるようにしたい。

これらの関数の戻り値を「装飾」するのです。一般的な方法として、第一成分が任意の型Aの値、第二成分が文字列であるペアをカプセル化するテンプレートWriterを定義することでそれを実現しましょう。

```
template<class A>
using Writer = pair < A, string > ;
```

以下は、装飾された関数です。

```
Writer<string> toUpper(string s) {
    string result;
    int (*toupperp)(int) = &toupper;
    transform(begin(s), end(s), back_inserter(result), toupperp);
    return make_pair(result, "toUpper ");
}

Writer<vector<string>> toWords(string s) {
    return make_pair(words(s), "toWords ");
}
```

We want to compose these two functions into another embellished function that uppercases a string and splits it into words, all the while producing a log of those actions. Here's how we may do it:

```
Writer<vector<string>> process(string s) {
    auto p1 = toUpper(s);
    auto p2 = toWords(p1.first);
    return make_pair(p2.first, p1.second + p2.second);
}
```

We have accomplished our goal: The aggregation of the log is no longer the concern of the individual functions. They produce their own messages, which are then, externally, concatenated into a larger log.

Now imagine a whole program written in this style. It's a nightmare of repetitive, error-prone code. But we are programmers. We know how to deal with repetitive code: we abstract it! This is, however, not your run of the mill abstraction — we have to abstract *function composition* itself. But composition is the essence of category theory, so before we write more code, let's analyze the problem from the categorical point of view.


```
ライター < 文字列 > toUpper(string s) { ----------以下がその例です。
    string result;
    int (*toupperp)(int) = &toupper;
    transform(begin(s), end(s), back_inserter(result), toupperp);
    return make_pair (result, "toUpper ");
}

ライター < ベクトル < 文字列 >> toWords(string s)
    return make_pair(words(s), "toWords ");
}
```

この2つの関数を組み合わせて、文字列を大文字にし、単語に分割し、さらにその動作のログを生成する、もう一つの装飾された関数にしたいと思います。その方法は次のとおりです。

```
ライター < ベクトル < 文字列 >> 処理(文字列 s)
    auto p1 = toUpper(s);
    auto p2 = toWords(p1.first);
    return make_pair(p2.first, p1.second + p2.second);
}
```

ログの集計はもはや個々の関数の関心事ではありません。彼らは自分自身のメッセージを生成し、それを外部で連結してより大きなログにします。さて、このようなスタイルで書かれたプログラム全体を想像してみてください。このようなスタイルで書かれたプログラム全体を想像してみると、繰り返しの多い、エラーを起こしやすい悪夢のようなコードになることでしょう。しかし、私たちはプログラマーです。私たちは、繰り返しの多いコードに対処する方法を知っています。しかし、これは一般的な抽象化ではなく、関数の構成そのものを抽象化しなければならない。しかし、構成はカテゴリ理論の本質である。そこで、さらにコードを書く前に、この問題をカテゴリの観点から分析してみよう。

4.1 The Writer Category

The idea of embellishing the return types of a bunch of functions in order to piggyback some additional functionality turns out to be very fruitful. We'll see many more examples of it. The starting point is our regular category of types and functions. We'll leave the types as objects, but redefine our morphisms to be the embellished functions.

For instance, suppose that we want to embellish the function `isEven` that goes from `int` to `bool`. We turn it into a morphism that is represented by an embellished function. The important point is that this morphism is still considered an arrow between the objects `int` and `bool`, even though the embellished function returns a pair:

```
pair<bool, string> isEven(int n) {
    return make_pair(n % 2 == 0, "isEven ");
}
```

By the laws of a category, we should be able to compose this morphism with another morphism that goes from the object `bool` to whatever. In particular, we should be able to compose it with our earlier `negate`:

```
pair<bool, string> negate(bool b) {
    return make_pair(!b, "Not so! ");
}
```

Obviously, we cannot compose these two morphisms the same way we compose regular functions, because of the input/output mismatch. Their composition should look more like this:

```
pair<bool, string> isOdd(int n) {
    pair<bool, string> p1 = isEven(n);
    pair<bool, string> p2 = negate(p1.first);
```


4.1 The Writer Category

関数の戻り値の型に装飾を施して機能を追加するというアイデアは、非常に実り多いものであることがわかります。これからもっと多くの例を見ていきましょう。出発点は、通常の型と関数のカテゴリです。型はオブジェクトとして残すが、モルヒズムは装飾された関数として再定義する。例えば、intからboolに変換する関数isEvenを装飾したいとする。これをモルヒズムに変換して、装飾された関数で表現する。重要な点は，このモルヒズムは，たとえ装飾された関数がペアを返すとしても，まだオブジェクトintとboolの間の矢印とみなされることです.

```
ペア <bool , string> isEven( int n) {。
    return make_pair(n % 2 == 0, "isEven ");
}
```

カテゴリーの法則によって、私たちはこのモルヒズムをオブジェクトboolからwhateverに行く別のモルヒズムと合成することができるはずである。特に、私たちの以前の否定と合成することができるはずです。

```
pair <bool , string > negate( bool b) { と合成することができます。
    return make_pair(!b, "Not so! ");
}
```

もちろん、この2つのモルヒズムは、入出力の不一致のために、通常の関数の合成と同じようには合成できない。この2つのモルヒズムの合成は、次のようになります。

```
pair <bool , string> isOdd( int n) {。
    pair<bool, string> p1 = isEven(n);
    pair<bool, string> p2 = negate(p1.first);
```

```
    return make_pair(p2.first, p1.second + p2.second);
}
```

So here's the recipe for the composition of two morphisms in this new category we are constructing:

1. Execute the embellished function corresponding to the first morphism
2. Extract the first component of the result pair and pass it to the embellished function corresponding to the second morphism
3. Concatenate the second component (the string) of the first result and the second component (the string) of the second result
4. Return a new pair combining the first component of the final result with the concatenated string.

If we want to abstract this composition as a higher order function in C++, we have to use a template parameterized by three types corresponding to three objects in our category. It should take two embellished functions that are composable according to our rules, and return a third embellished function:

```
template<class A, class B, class C>
function<Writer<C>(A)> compose(function<Writer<B>(A)> m1,
                               function<Writer<C>(B)> m2)
{
    return [m1, m2](A x) {
        auto p1 = m1(x);
        auto p2 = m2(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
    };
}
```

Now we can go back to our earlier example and implement the composition of `toUpper` and `toWords` using this new template:


```
    return make_pair(p2.first, p1.second + p2.second);
}
```

というわけで、この新しいカテゴリを構成する2つのモルヒズムの合成のレシピは以下のとおりです。

1. 最初のモルヒズムに対応する装飾された関数を実行する 2. 結果のペアの第一成分を取り出し、それを第二のモルヒズムに対応する装飾された関数に渡す 3. 3. 第1結果の第2成分（文字列）と第2結果の第2成分（文字列）を連結する 4. 最終結果の第一成分と連結された文字列を組み合わせた新しいペアを返す。

この合成をC++で高次関数として抽象化するには、カテゴリの3つのオブジェクトに対応する3つの型をパラメータとするテンプレートを使用しなければならない。このテンプレートは、我々のルールに従って合成可能な2つの装飾された関数を取 り、3番目の装飾された関数を返す必要があります。

```
template<class A , class B , class C>.
function<Writer<C>(A)> compose(function<Writer<B>(A)> m1,
                               function<Writer<C>(B)> m2)
{
    return [m1, m2](A x) {
        auto p1 = m1(x);
        auto p2 = m2(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
    };
}
```

さて、先ほどの例に戻り、この新しいテンプレートを使ってtoUpperとtoWordsの合成を実装してみましょう。

```
Writer<vector<string>> process(string s) {
    return compose<string, string, vector<string>>(toUpper, toWords)(s);
}
```

There is still a lot of noise with the passing of types to the compose template. This can be avoided as long as you have a C++14-compliant compiler that supports generalized lambda functions with return type deduction (credit for this code goes to Eric Niebler):

```
auto const compose = [](auto m1, auto m2) {
    return [m1, m2](auto x) {
        auto p1 = m1(x);
        auto p2 = m2(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
    };
};
```

In this new definition, the implementation of process simplifies to:

```
Writer<vector<string>> process(string s) {
    return compose(toUpper, toWords)(s);
}
```

But we are not finished yet. We have defined composition in our new category, but what are the identity morphisms? These are not our regular identity functions! They have to be morphisms from type A back to type A, which means they are embellished functions of the form:

```
Writer<A> identity(A);
```

They have to behave like units with respect to composition. If you look at our definition of composition, you'll see that an identity morphism should pass its argument without change, and only contribute an empty string to the log:


```
Writer < vector < string >> process(string s) {.
    return compose < string, string, vector < string >> (toUpper, toWords)(s);
}
```

composeテンプレートへの型の受け渡しには、まだノイズが多い。これは、戻り値の型控除が可能な一般化ラムダ関数をサポートするC++14準拠のコンパイラであれば回避できます（このコードの謝辞はEric Nieblerに捧げます）。

```
auto const compose = [](auto m1, auto m2) {
    return [m1, m2](auto x) {
        auto p1 = m1(x);
        auto p2 = m2(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
    };
};
```

この新しい定義では、processの実装は次のように単純化されます。

```
Writer < vector < string >> process(文字列 s) { となります。
    return compose(toUpper, toWords)(s);
}
```

しかし、まだ終わりではありません。新しいカテゴリでは合成を定義したが、恒等式モルヒズムは何だろうか？これは通常の恒等式関数ではありません。これは A 型から A 型に戻るモルヒズムでなければならない。つまり、次のような形式の装飾された関数である。

```
Writer<A> identity(A);
```

また、合成に関してもユニットのように振る舞わなければなりません。合成の定義を見ると、恒等式モルヒズムはその引数を変更せずに渡し、ログに空の文字列を寄与するだけであることがわかるでしょう。

```
template<class A> Writer<A> identity(A x) {
    return make_pair(x, "");
}
```

You can easily convince yourself that the category we have just defined is indeed a legitimate category. In particular, our composition is trivially associative. If you follow what's happening with the first component of each pair, it's just a regular function composition, which is associative. The second components are being concatenated, and concatenation is also associative.

An astute reader may notice that it would be easy to generalize this construction to any monoid, not just the string monoid. We would use `mappend` inside `compose` and `mempty` inside `identity` (in place of `+` and `""`). There really is no reason to limit ourselves to logging just strings. A good library writer should be able to identify the bare minimum of constraints that make the library work — here the logging library's only requirement is that the log have monoidal properties.

4.2 Writer in Haskell

The same thing in Haskell is a little more terse, and we also get a lot more help from the compiler. Let's start by defining the `Writer` type:

```
type Writer a = (a, String)
```

Here I'm just defining a type alias, an equivalent of a `typedef` (or `using`) in C++. The type `Writer` is parameterized by a type variable `a` and is equivalent to a pair of `a` and `String`. The syntax for pairs is minimal: just two items in parentheses, separated by a comma.


```
template<class A> Writer < A > identity(A x) {
    return make_pair(x, "");
}
```

今定義したカテゴリが本当に正当なカテゴリであることは簡単に納得できるだろう。特に、我々の構成は自明な連想性を持っている。各ペアの第1成分で何が起こっているかを追ってみると、それは単なる普通の関数合成で、それは連想的である。2番目のコンポーネントは連結され、連結もまた連想的である。賢明な読者は、この構成を文字列モノイドだけでなく、任意のモノイドに簡単に一般化できることにお気づきでしょう。composeの内部にはmappendを、identityの内部にはmemptyを（+や""の代わりに）使うのです。文字列のロギングに限定する理由はありません。良いライブラリの作者は、ライブラリを動作させるための最小限の制約を特定することができるはずです。ここでは、ロギングライブラリの唯一の要件は、ログがモノイダルプロパティを持つということです。

4.2 Writer in Haskell

Haskellで同じことをすると、もう少し簡潔で、コンパイラからもっと多くの助けを得ることができます。まず、Writer 型を定義することから始めましょう。

```
type Writer a = (a, String)
```

これは C++ の typedef（または using ）に相当するもので、型の別名を定義しているだけです。Writer 型は型変数 a でパラメタ化され，a と String のペアに相当します．ペアの構文は最小限のもので、2つの項目を括弧で囲み、コンマで区切っただけです。

Our morphisms are functions from an arbitrary type to some `Writer` type:

```
a -> Writer b
```

We'll declare the composition as a funny infix operator, sometimes called the "fish":

```
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
```

It's a function of two arguments, each being a function on its own, and returning a function. The first argument is of the type `(a -> Writer b)`, the second is `(b -> Writer c)`, and the result is `(a -> Writer c)`.

Here's the definition of this infix operator — the two arguments `m1` and `m2` appearing on either side of the fishy symbol:

```
m1 >=> m2 = \x ->
    let (y, s1) = m1 x
        (z, s2) = m2 y
    in (z, s1 ++ s2)
```

The result is a lambda function of one argument `x`. The lambda is written as a backslash — think of it as the Greek letter λ with an amputated leg.

The `let` expression lets you declare auxiliary variables. Here the result of the call to `m1` is pattern matched to a pair of variables `(y, s1)`; and the result of the call to `m2`, with the argument `y` from the first pattern, is matched to `(z, s2)`.

It is common in Haskell to pattern match pairs, rather than use accessors, as we did in C++. Other than that there is a pretty straightforward correspondence between the two implementations.

モルヒズムは，任意の型からあるWriter型への関数です．

```
a -> Writer b
```

合成は、時々「魚」と呼ばれるおかしな infix 演算子として宣言します。

```
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
```

これは二つの引数からなる関数で、それぞれがそれ自体で関数であり、関数を返す。最初の引数は (a -> Writer b) で、2番目の引数は (b -> Writer c) で、その結果は (a -> Writer c) です。この infix 演算子の定義を以下に示します。フィッシュシンボルの両側に現れる2つの引数 m1 と m2 は、このようなものです。

```
m1 >=> m2 = \x ->
    let (y, s1) = m1 x
        (z, s2) = m2 y
    in (z, s1 ++ s2)
```

ラムダはバックスラッシュで記述され、ギリシャ文字のλの足を切断したようなものだと考えてください。let式を使うと、補助変数を宣言することができます。ここでは，m1 の呼び出しの結果が変数のペア (y, s1) にパターンマッチされ， m2 の呼び出しの結果が，最初のパターンから引数 y をとって (z, s2) にマッチされます．Haskellでは、C++でやったようなアクセサを使うのではなく、ペアをパターンマッチするのが一般的です。それ以外の点では、2つの実装の間には非常にわかりやすい対応があります。

The overall value of the `let` expression is specified in its `in` clause: here it's a pair whose first component is `z` and the second component is the concatenation of two strings, `s1++s2`.

I will also define the identity morphism for our category, but for reasons that will become clear much later, I will call it `return`.

```
return :: a -> Writer a
return x = (x, "")
```

For completeness, let's have the Haskell versions of the embellished functions `upCase` and `toWords`:

```
upCase :: String -> Writer String
upCase s = (map toUpper s, "upCase ")

toWords :: String -> Writer [String]
toWords s = (words s, "toWords ")
```

The function `map` corresponds to the C++ `transform`. It applies the character function `toUpper` to the string `s`. The auxiliary function `words` is defined in the standard Prelude library.

Finally, the composition of the two functions is accomplished with the help of the fish operator:

```
process :: String -> Writer [String]
process = upCase >=> toWords
```

4.3 Kleisli Categories

You might have guessed that I haven't invented this category on the spot. It's an example of the so called Kleisli category — a category based

let式の全体の値はそのin節で指定されます。ここでは、第1成分がzで、第2成分が2つの文字列の連結、s1++s2であるペアになっています。また、このカテゴリの同一性モルヒズムを定義しますが、ずっと後で明らかになるであろう 理由から、これを return と呼ぶことにします。

```
return :: a -> Writer a
return x = (x, "")
```

完全を期すために、Haskellバージョンの装飾された関数 upCase と toWords を用意しましょう ：

```
upCase :: String -> Writer String (文字列)
upCase s = (map toUpper s, "upCase ")

toWords :: String -> Writer [ String ]である。
toWords s = (words s, "toWords ")
```

関数mapは、C++の変換に相当する。補助関数 words は標準の Prelude ライブラリで定義されています。最後に、2つの関数の合成は、fish 演算子の助けを借りて行われます。

```
process :: String -> Writer [ String ]。
process = upCase >=> toWords
```

4.3 Kleisli Categories

このカテゴリは私が即興で作ったものではないことにお気づきでしょうか。これはいわゆるクライスリーカテゴリーの一例です。

on a monad. We are not ready to discuss monads yet, but I wanted to give you a taste of what they can do. For our limited purposes, a Kleisli category has, as objects, the types of the underlying programming language. Morphisms from type *A* to type *B* are functions that go from *A* to a type derived from *B* using the particular embellishment. Each Kleisli category defines its own way of composing such morphisms, as well as the identity morphisms with respect to that composition. (Later we'll see that the imprecise term "embellishment" corresponds to the notion of an endofunctor in a category.)

The particular monad that I used as the basis of the category in this post is called the *writer monad* and it's used for logging or tracing the execution of functions. It's also an example of a more general mechanism for embedding effects in pure computations. You've seen previously that we could model programming-language types and functions in the category of sets (disregarding bottoms, as usual). Here we have extended this model to a slightly different category, a category where morphisms are represented by embellished functions, and their composition does more than just pass the output of one function to the input of another. We have one more degree of freedom to play with: the composition itself. It turns out that this is exactly the degree of freedom which makes it possible to give simple denotational semantics to programs that in imperative languages are traditionally implemented using side effects.

4.4 Challenge

A function that is not defined for all possible values of its argument is called a partial function. It's not really a function in the mathematical sense, so it doesn't fit the standard categorical mold. It can, however, be represented by a function that returns an embellished type `optional`:

モナドに基づくカテゴリです。まだモナドについて議論する段階ではないのですが、モナドに何ができるかを味わっていただきたかったのです。この限られた目的のために、クライスレ・カテゴリーは、基礎となるプログラミング言語の型 をオブジェクトとして持っています。型 ŭ から型 Lu_lD435 へのモルヒズムは Lu_lD435 から特定の装飾を使用して派生した型に行く関数である。各クライスリ範疇は、そのようなモルヒズムを構成する独自の方法と、その構成に関する恒等モルヒズムを定義している。(後で、「装飾」という不正確な用語が、カテゴリーにおけるエンドファンクタの概念に対応することを見ることになる）。この投稿でカテゴリの基礎として使った特定のモナドはライターモナドと呼ばれ、関数の実行をロギングしたりトレースしたりするのに使われるものである。これはまた、純粋な計算の中に効果を埋め込むための、より一般的なメカニズムの例でもあります。プログラミング言語の型と関数を集合のカテゴリでモデル化できることは、以前に見たとおりだ（いつものように底は無視する）。ここでは、このモデルを少し異なるカテゴリに拡張した。カテゴリでは、モルヒズムは装飾された関数で表され、その合成は、ある関数の出力を別の関数の入力に渡すだけでなく、それ以上のことを行う。我々は、もう1つの自由度、すなわち合成そのものを弄ることができるのである。この自由度によって、命令型言語では伝統的に副作用を使って実装される プログラムに、簡単な意味づけをすることができるようになるのである。

4.4 Challenge

引数のすべての値に対して定義されていない関数を部分関数と呼びます。これは数学的な意味での関数ではないので、標準的なカテゴリ型には当てはまら ない。しかし、これは、オプションの装飾された型を返す関数で表現することができます。

```
template<class A> class optional {
    bool _isValid;
    A _value;
public:
    optional()    : _isValid(false) {}
    optional(A v) : _isValid(true), _value(v) {}
    bool isValid() const { return _isValid; }
    A value() const { return _value; }
};
```

For example, here's the implementation of the embellished function `safe_root`:

```
optional<double> safe_root(double x) {
    if (x >= 0) return optional<double>{sqrt(x)};
    else return optional<double>{};
}
```

Here's the challenge:

1. Construct the Kleisli category for partial functions (define composition and identity).
2. Implement the embellished function `safe_reciprocal` that returns a valid reciprocal of its argument, if it's different from zero.
3. Compose the functions `safe_root` and `safe_reciprocal` to implement `safe_root_reciprocal` that calculates `sqrt(1/x)` whenever possible.


```
template<class A> class optional {
    bool _isValid;
    A _value;
public:
    optional()    : _isValid(false) {}
    optional(A v) : _isValid(true), _value(v) {}
    bool isValid() const { return _isValid; }.
    A value() const { return _value; }.
};
```

たとえば、以下のように、装飾された関数safe_rootの実装は、以下のとおりです。

```
オプションの <double> safe_root( double x) {。
    if (x >= 0) return optional<double>{sqrt(x)};
    else return オプションの <double> {};
}
```

Here's the challenge:

1．部分関数のクライスリカテゴリを構築せよ（合成と恒等式を定義）。2．2．引数がゼロでなければ、その有効な逆数を返す関数 safe_reciprocal を実装する。3．3．関数 safe_root と safe_reciprocal を合成し、可能な限り sqrt(1／x) を計算する safe_root_reciproca l を実装する。