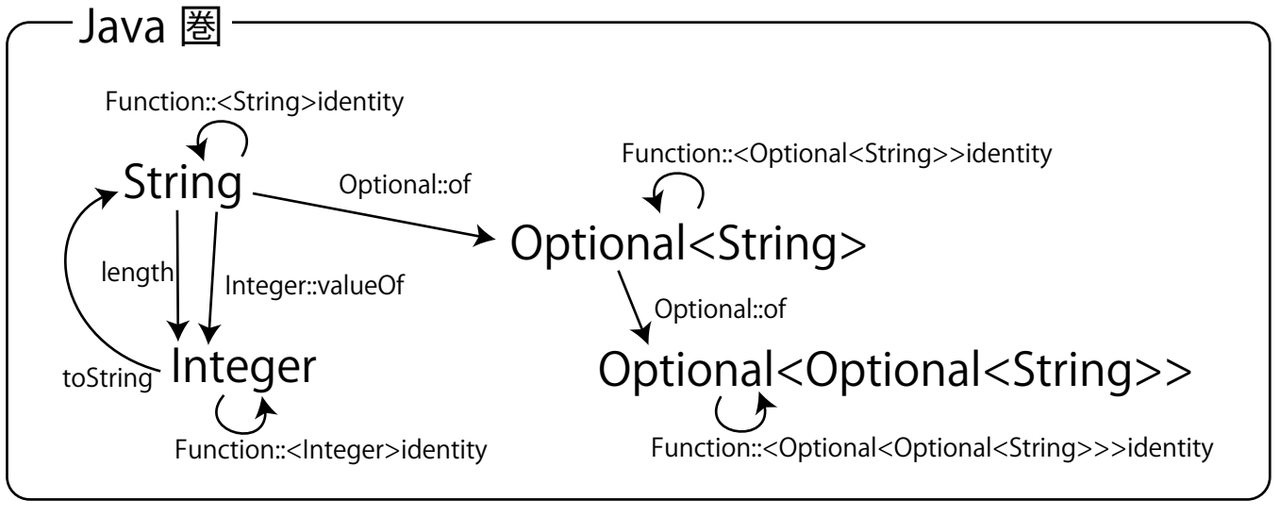
圏の定義 :

1. 対象という集団と、射という集団からなる。
2. 射 f: T→R と g: R→S があった場合、その合成 g・f: T→S が一意に存在する。
3. ・は結合律を満たす。
4. 各対象 T に対して恒等射 I_T が存在し、合成の単位元となる。



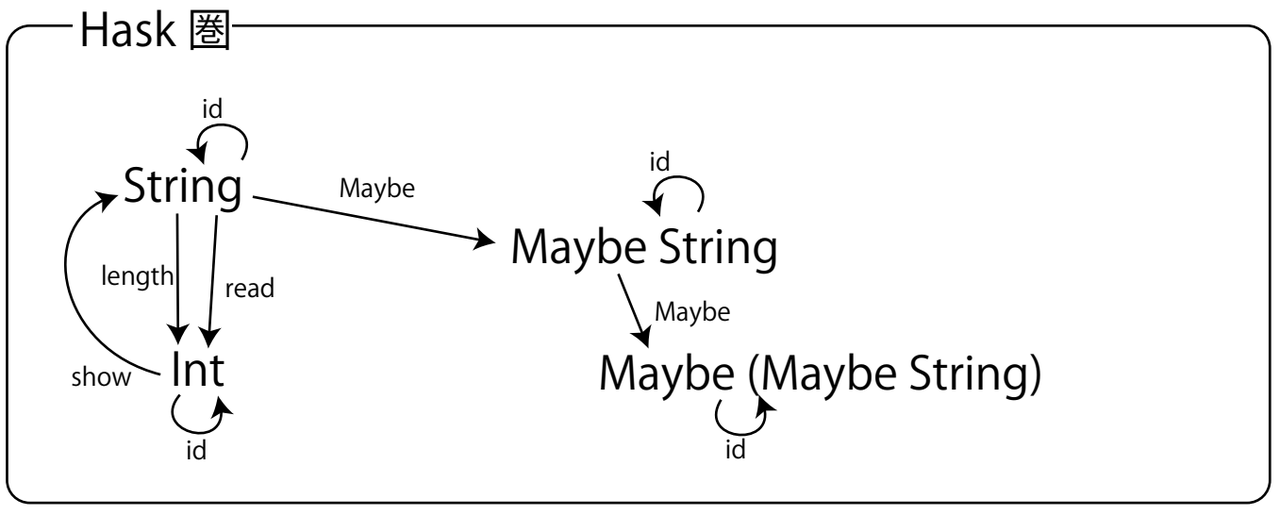
対象は型であり、射は関数である。

射の合成は関数の合成。

恒等射は Function::<T>identity

射の合成は省略。

本当は String から Optional<Optional<String>> への射 Optional::of・Optional::of などがある。



対象は型であり、射は関数である。

射の合成は関数の合成。

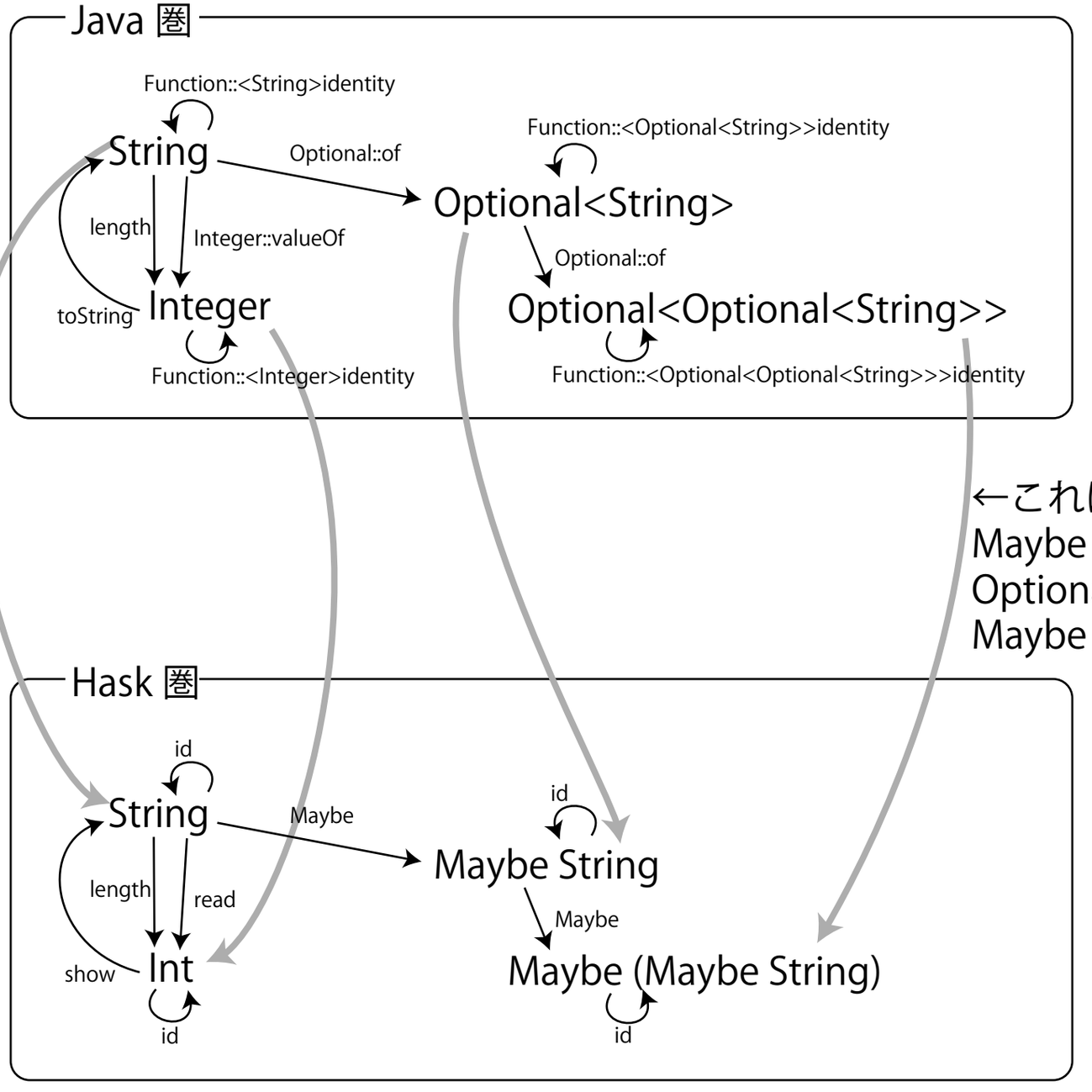
恒等射は id

射の合成は省略。

本当は String から Maybe (Maybe String) への射 Maybe . Maybe などがある。

圏 C から圏 D への関手 F の定義

1. 圏 C の任意の対象 X に対して圏 D の対象 F (X) を一意に定める。
2. 圏 C の任意の射 f: T→R に対して圏 D の射 F(f): F(T)→F(R) を一意に定める。
3. F(g・f) = F(g)・F(f) が成り立つ



射の対応は省略

本当は Java の length と Haskell の length などが対応する。

自己関手でない

Java 圏から Hask 圏への関手 F

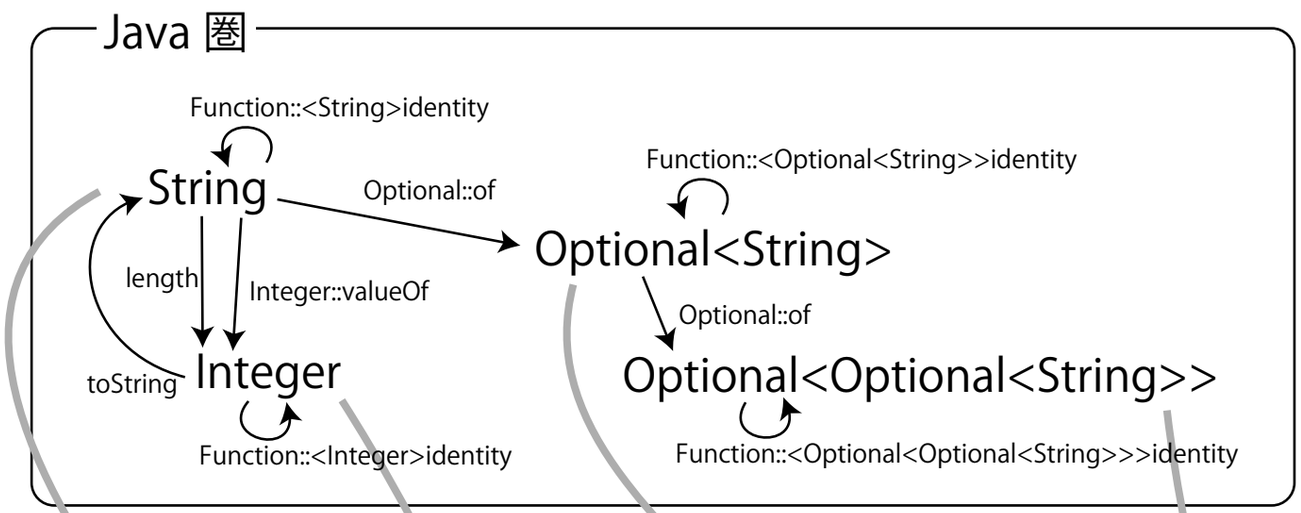
1. Java の任意の型 X に対して
Haskell の型 F (X) を一意に定める。
2. Java の任意の関数 f: T→R に対して
Haskell の関数 F(f): F(T)→F(R) を一意に定める。
3. F(g・f) = F(g)・F(f) が成り立つ

←これは Optional<Optional<String>> 型と
Maybe (Maybe String) 型の対応関係を表した矢印であり、
Optional<Optional<String>> の値から
Maybe (Maybe String) の値への関数ではない。

Optional<String> に対して対応する型
Maybe String があればよく、
Optional<String> の値から Maybe String の値
への関数は必要ない。

圏 C から圏 D への関手 F の定義

1. 圏 C の任意の対象 X に対して圏 D の対象 F (X) を一意に定める。
2. 圏 C の任意の射 f: T→R に対して圏 D の射 F(f): F(T)→F(R) を一意に定める。
3. F(g・f) = F(g)・F(f)

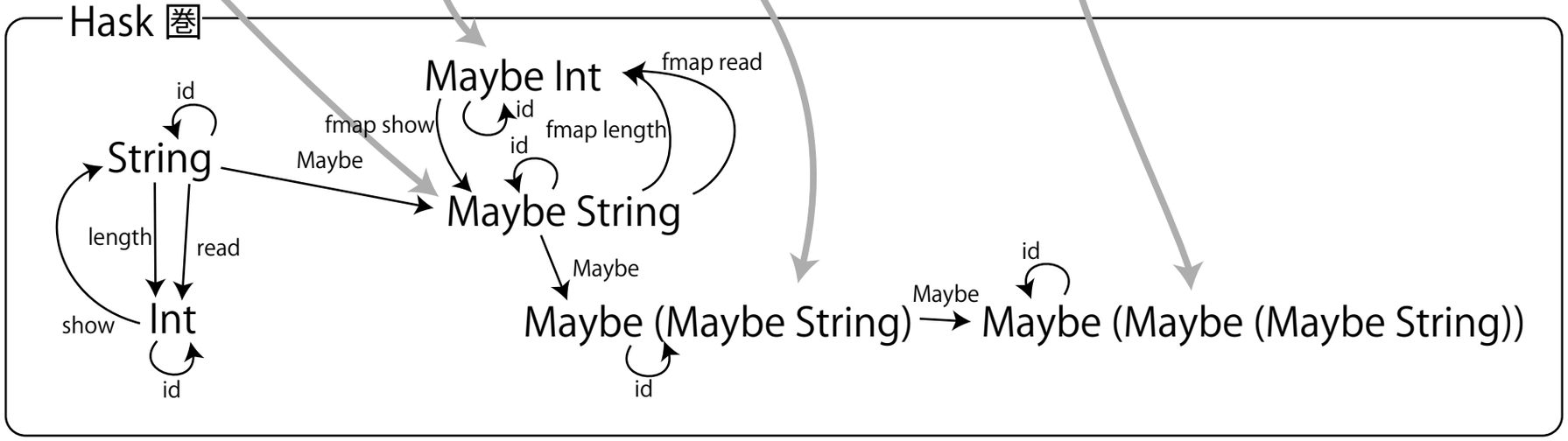


自己関手でない

Java 圏から Hask 圏への別の関手 F'

1. Java の任意の型 X に対して
Haskell の型 F '(X) を一意に定める。
2. Java の任意の関数 f: T→R に対して
Haskell の関数 F' (f): F' (T)→F' (R) を一意に定める。
3. F(g・f) = F(g)・F(f)

Java の String を Haskell の Maybe String に対応させてもよい！



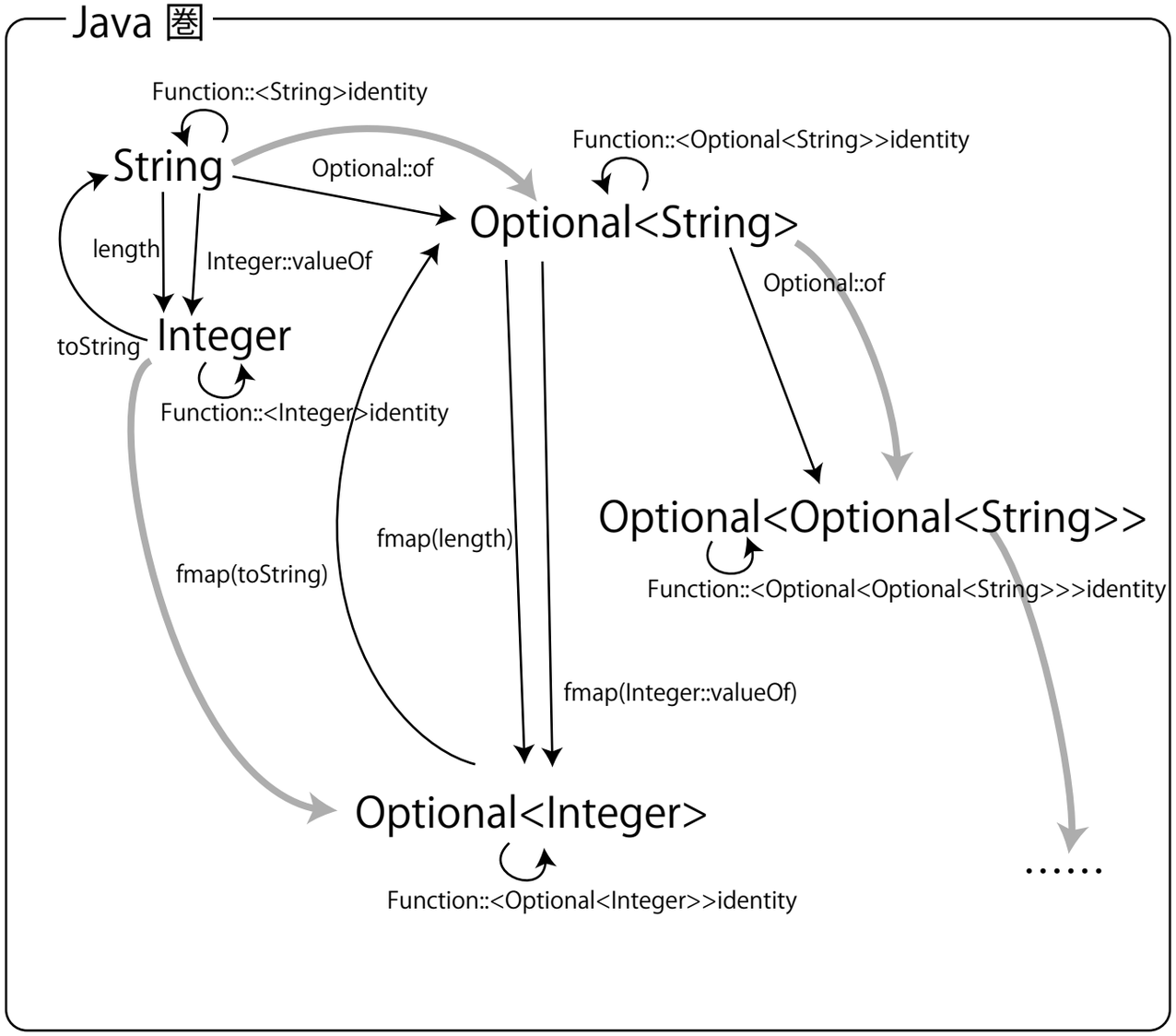
Java の型に対して
対応する型が Haskell に
あればよく、
Haskell の型の中に
Java の型に対応しない
型があってもよい。

射の対応は省略

本当は Java の length と Haskell の fmap length などが対応する。

圏 C から圏 D への関手 F の定義

1. 圏 C の任意の対象 X に対して圏 D の対象 F (X) を一意に定める。
2. 圏 C の任意の射 f: T→R に対して圏 D の射 F(f): F(T)→F(R) を一意に定める。
3. F(g・f) = F(g)・F(f)



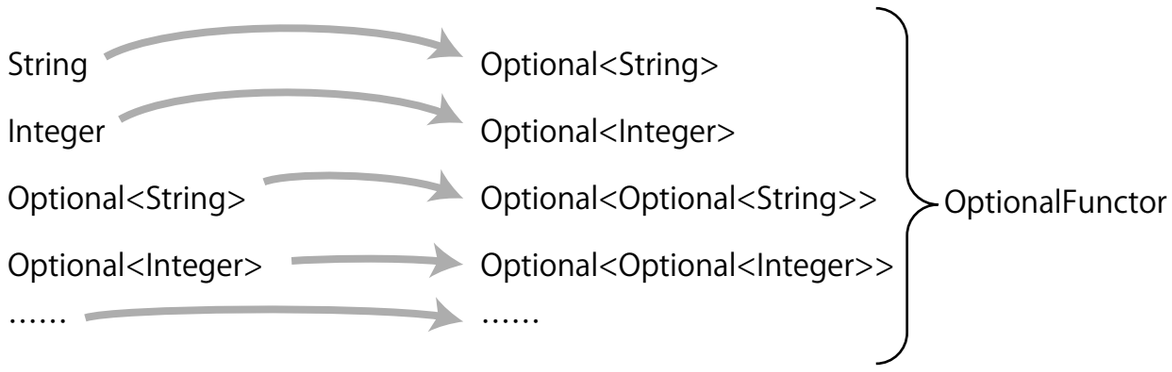

```
Function<Optional<T>, Optional<R>> fmap(Function<T, R> f) {
  ...
}
```

Java 圏から Java 圏への**自己**関手 F

1. Java の任意の型 X に対して
Java の型 F (X) を一意に定める。
2. Java の任意の関数 f: T→R に対して
Java の関数 F(f): F(T)→F(R) を一意に定める。
3. F(g・f) = F(g)・F(f)

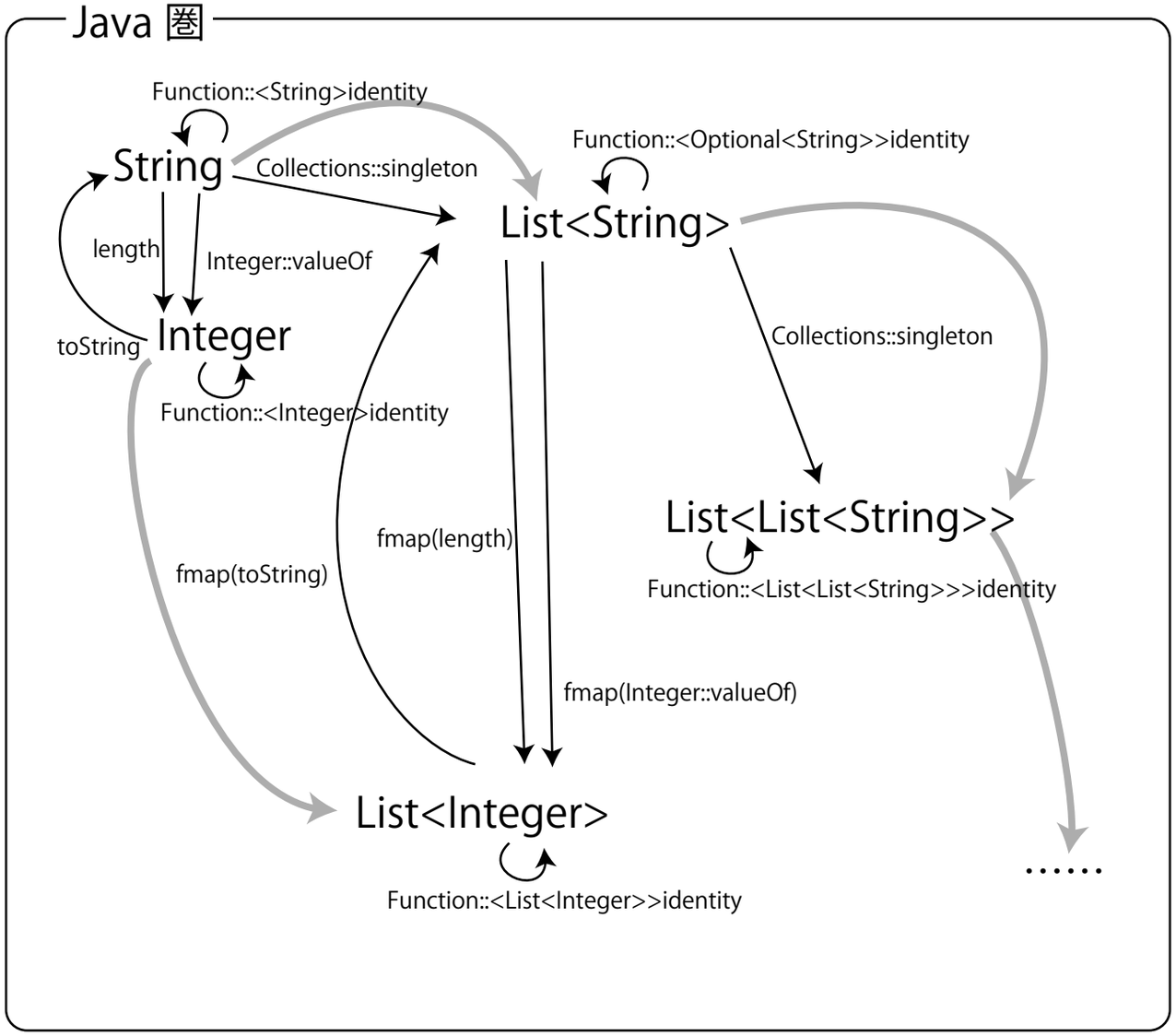
Java の String を Java の Optional<String> に対応させてもよい！

この灰色の矢印達を OptionalFunctor と呼ぶことにする。
(Optional クラスと区別するため)



圏 C から圏 D への関手 F の定義

1. 圏 C の任意の対象 X に対して圏 D の対象 F (X) を一意に定める。
2. 圏 C の任意の射 f: T→R に対して圏 D の射 F(f): F(T)→F(R) を一意に定める。
3. F(g・f) = F(g)・F(f)



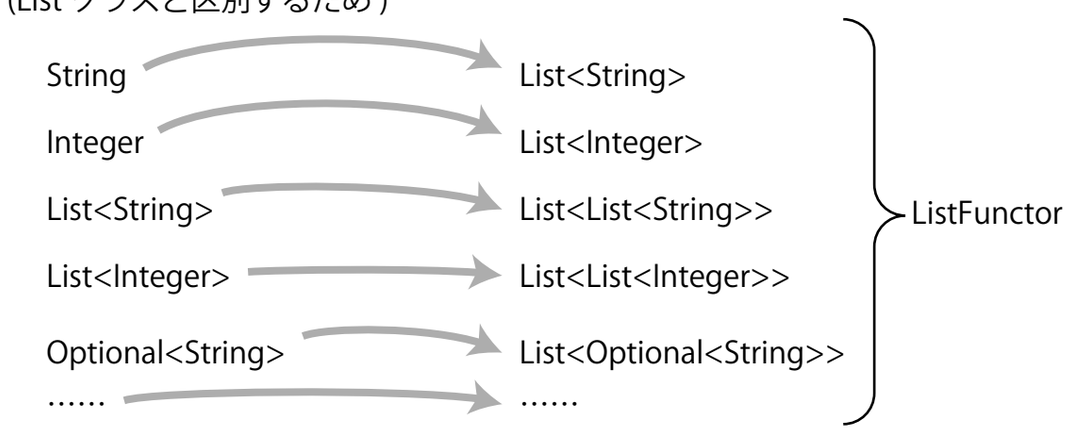

```
Function<List<T>, List<R>> fmap(Function<T, R> f) {
  ...
}
```

Java 圏から Java 圏への**自己**関手 F

1. Java の任意の型 X に対して
Java の型 F (X) を一意に定める。
2. Java の任意の関数 f: T→R に対して
Haskell の関数 F(f): F(T)→F(R) を一意に定める。
3. F(g・f) = F(g)・F(f)

同様に Java の String を Java の List<String> に対応させてもよい！

この灰色の矢印達を ListFunctor と呼ぶことにする。
(List クラスと区別するため)



圏 C から圏 D への関手 F, G に対して、F から G への自然変換の定義

1. 圏 C の任意の対象 T に対して、圏 D の射 θ_T: F(T)→G(T) を一意に定める。
2. 圏 C の任意の射 f: A→B に対して、$\theta_B \cdot F(f) = G(f) \cdot \theta_A$ が成り立つ。

OptionalFunctor から ListFunctor への自然変換

Java の任意の型 T に対して、Java の関数 Optional<T>→List<T> を一意に定める。

例 :

```
<T> List<T> optionalToList(Optinal(Optional<T> o) {
  if (o.isPresent()) {
    return Collections.singleton(o.get());
  } else {
    return Collections.emptyList();
  }
}
```

Optional.fmap や List.fmap があると仮定して、

```
optionalToList(Optional.fmap(f).apply(o))
    == List.fmap(f).apply(optionalToList(o))
```

※
関手は値から値への関数ではなかったが、
自然変換は Optinal<T> の値から List<T> の値への関数である。

なお、Optional<String> から List<String> への関数と
Optional<Integer> から List<Integer> への関数は別でもよく、
正確には「自然変換は関数の集まりである」と言う。

Java 圏の自己関手

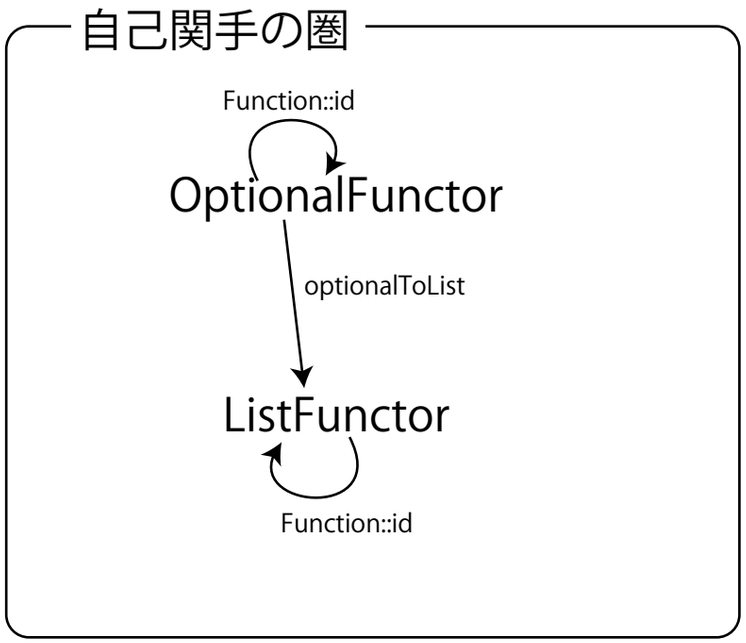
Java の型 T → Java の型 F<T>

T から R への Java の関数 → F<T> から F<R> への Java の関数

Java 圏の自己関手間の自然変換
F<T> の値から G<T> の値への Java の関数 (の集まり)

圏の定義 :

1. 対象という集団と、射という集団からなる。
2. 射 f: T→R と g: R→S があった場合、その合成 g・f: T→S が一意に存在する。
3. ・は結合律を満たす。
4. 各対象 T に対して恒等射 I_T が存在し、合成の単位元となる。



対象は関手
射は自然変換

ここでの矢印は、OptionalFunctor から ListFunctor への関数ではなく、
OptionalFunctor から ListFunctor への間の自然変換。
つまり、Optional<T> の値から List<T> の値へのジェネリックな関数。

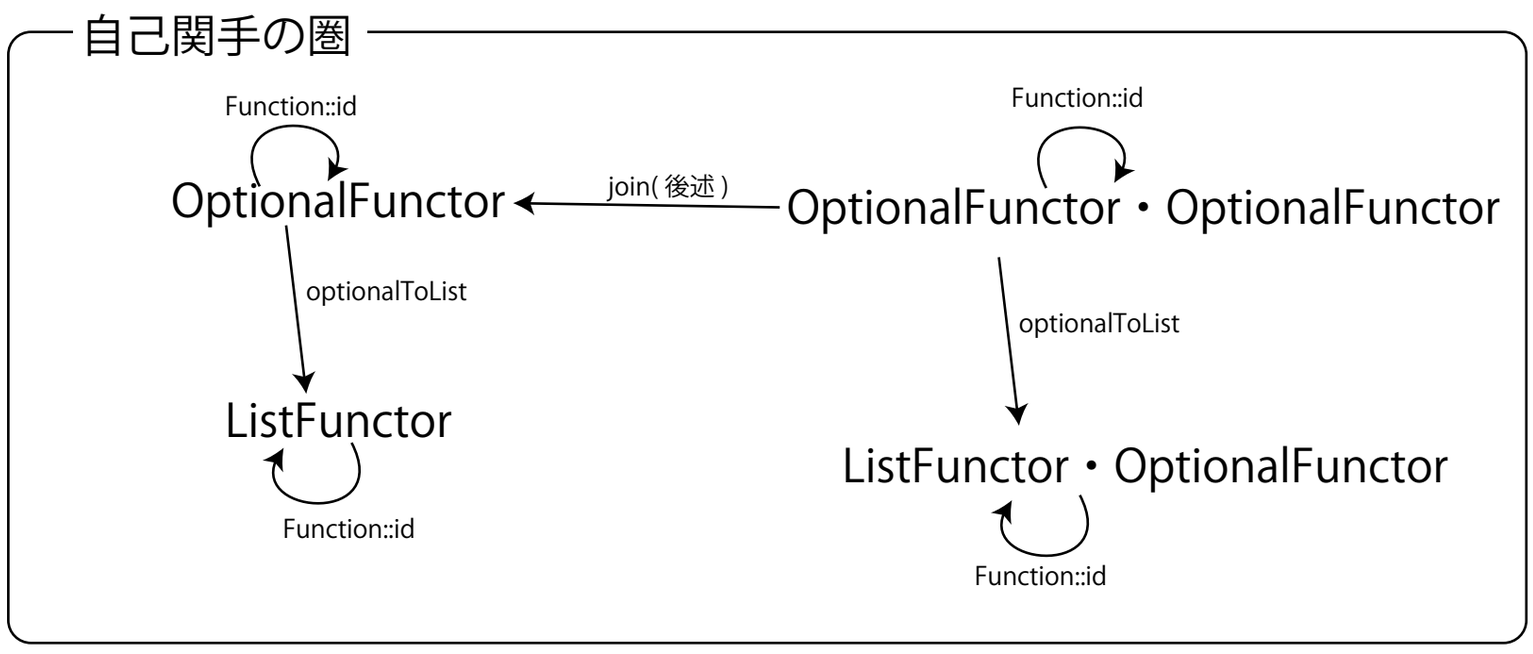
関手の合成

OptionalFunctor は型 T を Optional<T> に対応させる。

ListFunctor は型 T を List<T> に対応させる。

その 2 つのを合成した関手 ListFunctor・OptionalFunctor は
型 T を List<Optional<T>> に対応させる。

同様に OptionalFunctor・OptionalFunctor は
型 T を Optional<Optional<T>> に対応させる。



自己関手の圏におけるモノイド対象

自己関手 F が次の式を満たすとき F は自己関手の圏におけるモノイド対象であるという。

Haskell における Monoid とは異なるので忘れよう

モノイド圏とも異なるよ！

1. F・F→F という自然変換 μ が存在する。

例 : OptionalFunctor・OptionalFunctor→OptionalFunctor という自然変換が存在する。
つまり任意の型 T に対して Optional<Optional<T>>→Optional<T> という関数が存在する。

```
<T> Optional<T> join(Optional<Optional<T>> oo) {
  if (oo.isPresent()) {
    return oo.get();
  } else {
    return Optional.<T>empty();
  }
}
```

2. I→F という自然変換 η が存在する。ただし I は対象 X を対象 X に対応付ける関手 (恒等関手)。

例 : I→OptionalFunctor という自然変換が存在する。
つまり任意の型に対して T→Optional<T> という関数が存在する。

```
<T> Optional<T> unit(T v) {
  return Optional.of(v);
}
```

3. μ と η がいわゆるモナド則を満たす。

ところでこの条件は F がモナドである条件である。

つまり、**モナドは自己関手の圏におけるモノイド対象だよ !!**

どうでもよい知識

「モナドは単なる自己関手の圏におけるモノイド対象だよ。何か問題でも ?」の元ネタは
『不完全にしておよそ正しくないプログラミング言語小史』だが、そのさらに元ネタは
“Categories for the Working Mathematician” (邦題『圏論の基礎』) である。

“a monad in X is just a monoid in the category of endofunctors of X, with product × replaces by composition of endofunctors and unit set by the identity endofunctor.”

参考訳 (『圏論の基礎』が手元にないので独自訳)
「X におけるモナドとは、(集合の圏のモノイドに対して)× を自己関手の合成に置き換え、
単位集合を単位自己関手に置き換えた、X に対する自己関手の圏の単なるモノイド対象だよ」