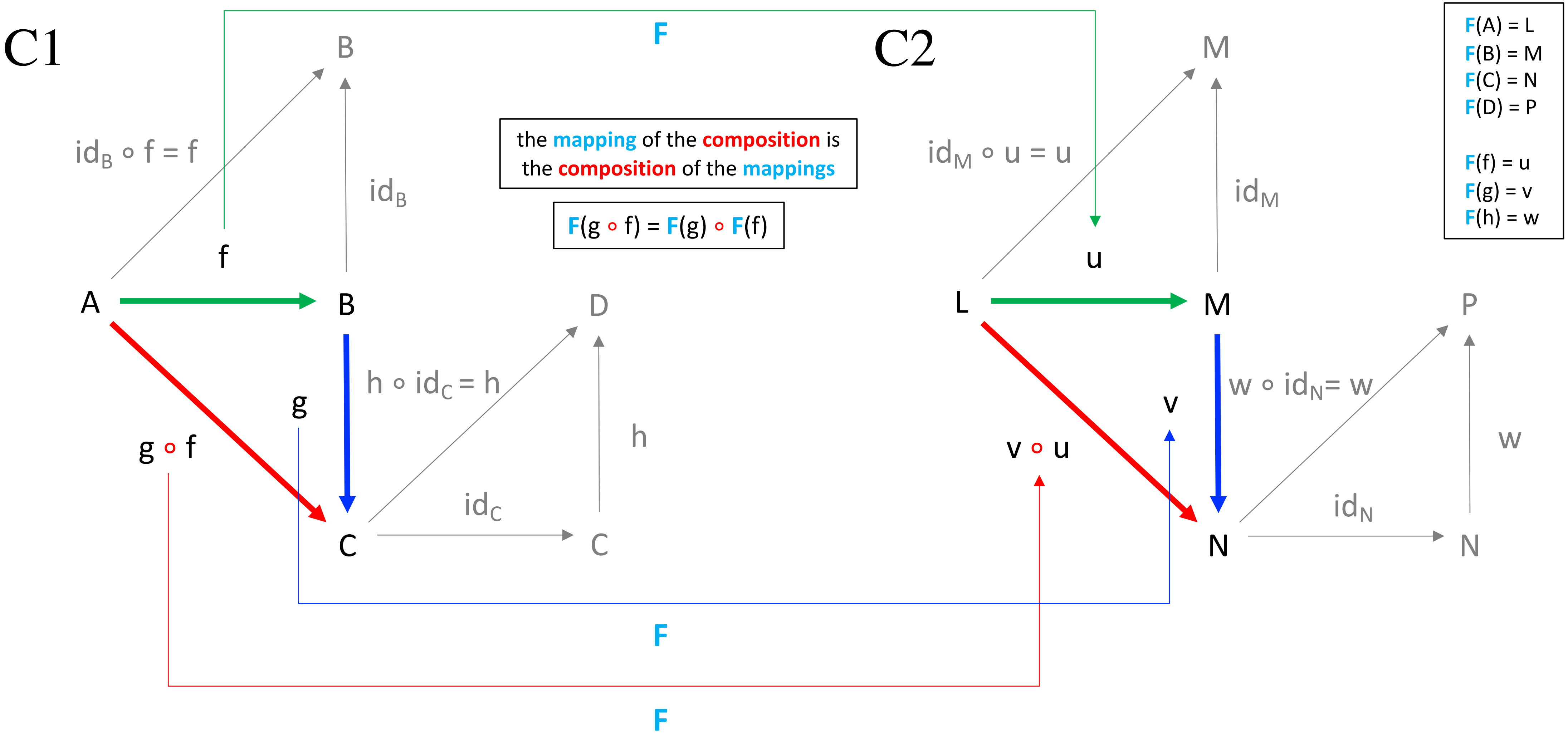
Functor Laws

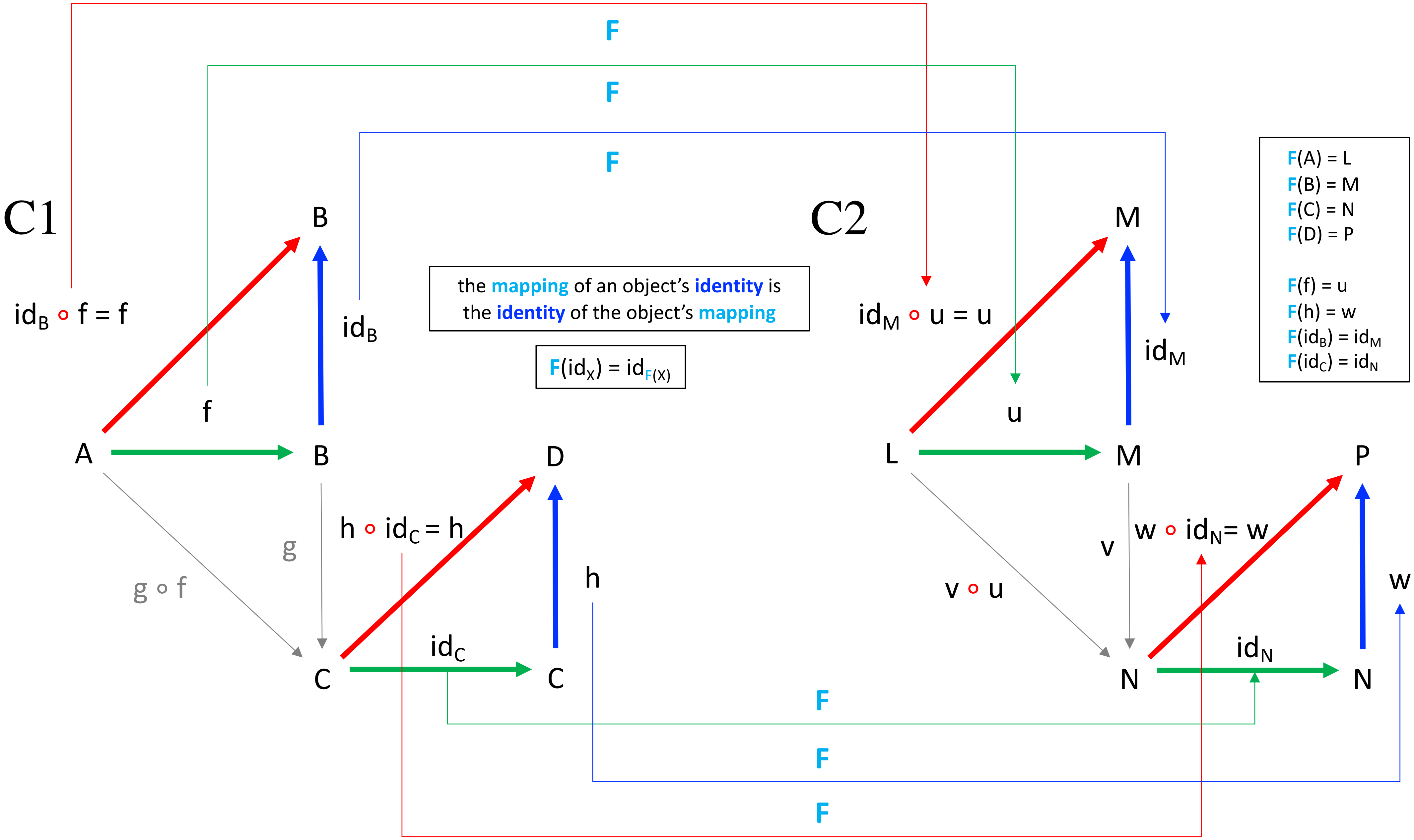
C1 and C2 are categories and ∘ denotes their composition operations. For every category object **X** there is an identity arrow $\mathbf{id_X : X \to X}$ such that for every category arrow $\mathbf{f : A \to B}$ we have $\mathbf{id_B \circ f = f = f \circ id_A}$.

There is a functor F from C1 to C2 (a category homomorphism) which maps each C1 object to a C2 object and maps each C1 arrow to a C2 arrow in such a way that the following two **functor laws** are satisfied (i.e. in such a way that composition and identity are preserved):

1) $F(g \circ f) = F(g) \circ F(f)$ - mapping the composition of two arrows is the same as composing the mapping of the 1st arrow and the mapping of 2nd arrow

2) $F(id_X) = id_{F(X)}$ - the mapping of an object's identity is the same as the identity of the object's mapping




C1
C2
F
F
F
F
F
F
$id_B \circ f = f$
$id_M \circ u = u$
$h \circ id_C = h$
$w \circ id_N = w$
the mapping of an object's identity is
the identity of the object's mapping
$F(id_X) = id_{F(X)}$
$F(A) = L$
$F(B) = M$
$F(C) = N$
$F(D) = P$
$F(f) = u$
$F(h) = w$
$F(id_B) = id_M$
$F(id_C) = id_N$
A
B
C
D
f
g
$g \circ f$
id_B
id_C
h
L
M
N
P
u
v
$v \circ u$
id_M
id_N
w

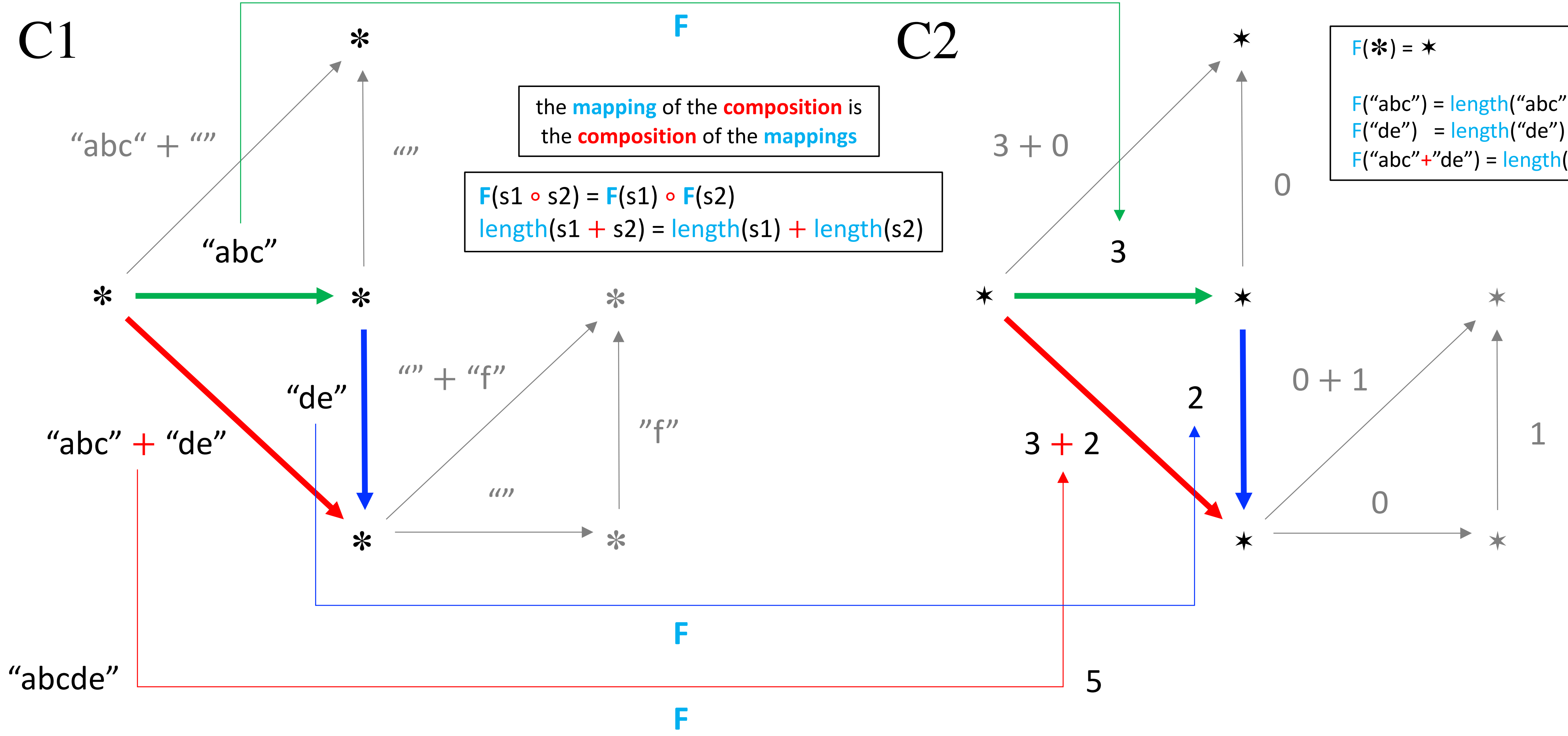
Example: a Functor from one Monoid to another

C1 = Monoid(String, +, "")
- **objects**: just one, denoted by ✻
- **arrows**: strings
- **composition operation**: string concatenation
- **identity arrows**: the empty string

C2 = Monoid (Int, +, 0)
- **objects**: just one, denoted by ✶
- **arrows**: integer numbers
- **composition operation**: integer addition
- **identity arrows**: the number zero

A **functor F** from C1 to C2 that maps ✻ to ✶ and maps a string to its length. The functor laws
- $F(g \circ f) = F(g) \circ F(f)$
- $F(id_X) = id_{F(X)}$

become
- length(s1 + s2) = length(s1) + length(s2)
- length("") = 0

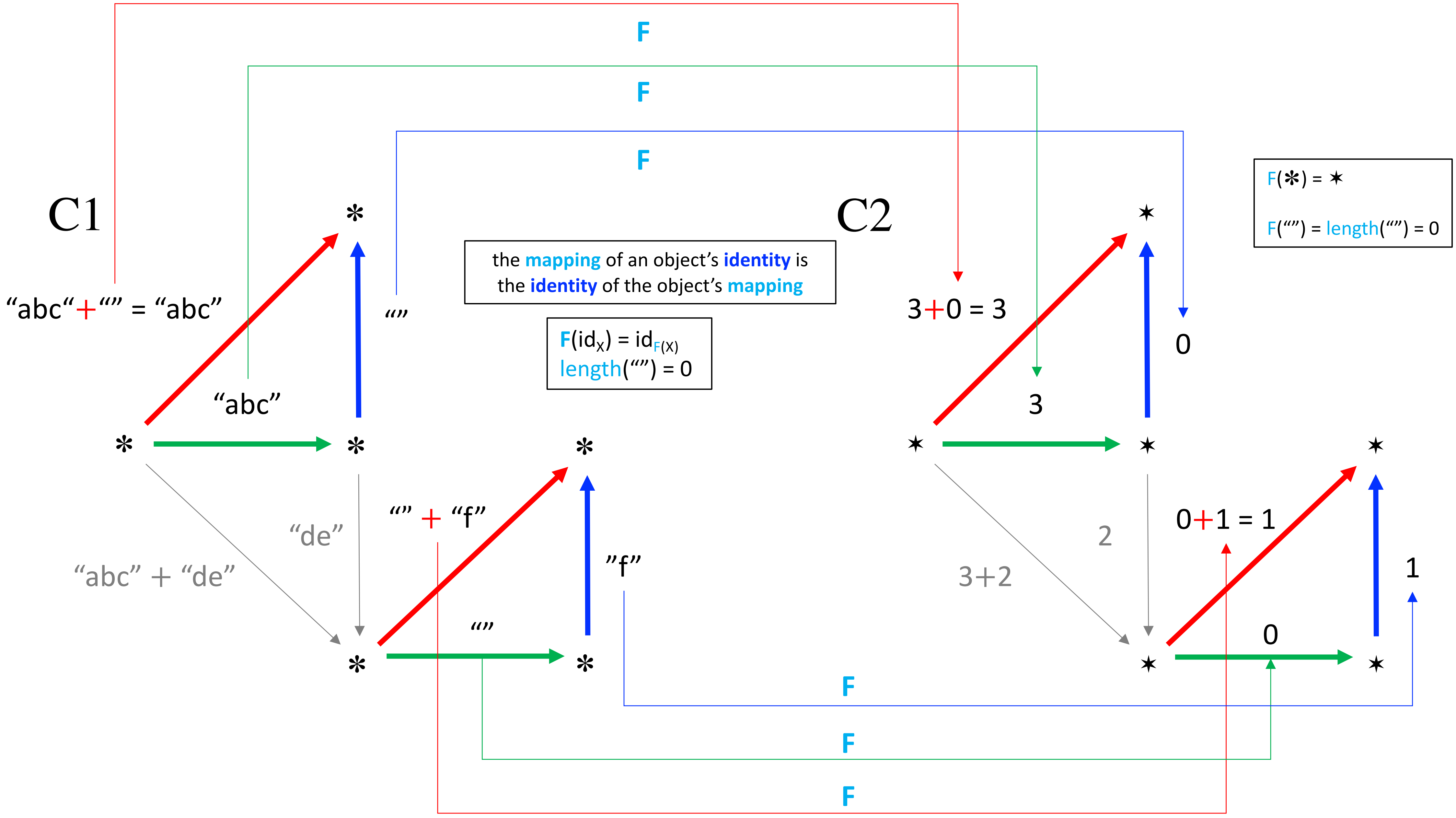


the **mapping** of the **composition** is the **composition** of the **mappings**

$F(s1 \circ s2) = F(s1) \circ F(s2)$
length(s1 + s2) = length(s1) + length(s2)

F(✻) = ✶

F("abc") = length("abc") = 3
F("de") = length("de") = 2
F("abc"+"de") = length("abc"+"de") = 5


C1
C2
F
F
F
F(✻) = ✶
F("") = length("") = 0
"abc"+"" = "abc"
""
"abc"
the mapping of an object's identity is
the identity of the object's mapping
F(id_X) = id_F(X)
length("") = 0
3+0 = 3
0
3
"de"
"" + "f"
"abc" + "de"
"f"
""
2
0+1 = 1
3+2
1
0
F
F
F

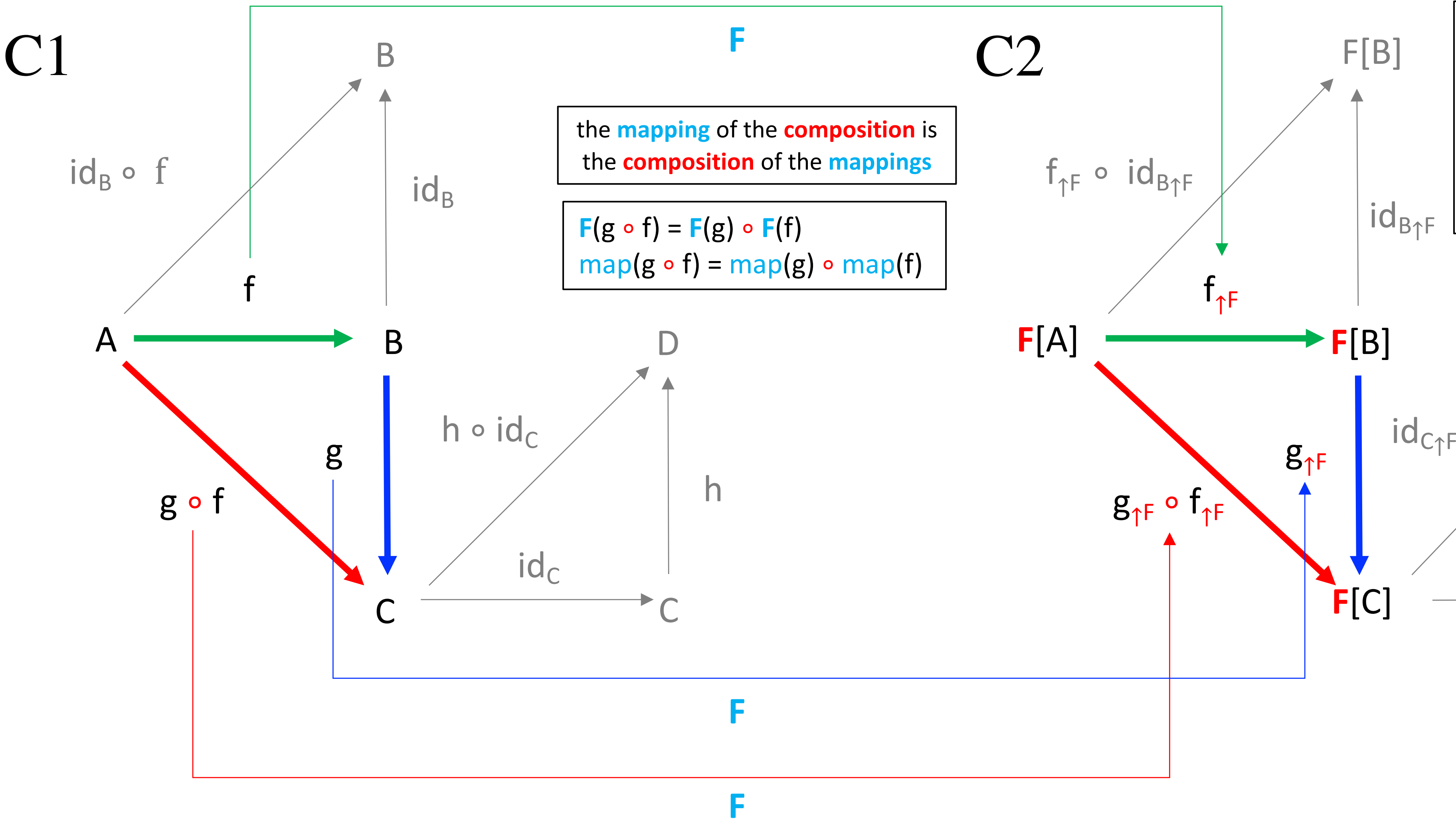
Example: a Functor from the category of 'Scala types and functions' to itself

$f_{\uparrow F}$ is function f lifted into context **F**
F[A] is type A lifted into context **F**
$id_{X_{\uparrow F}}$ is id_X lifted into context **F**

C1 = C2 = Scala types and functions
- **objects**: types
- **arrows**: functions
- **composition operation**: compose function, denoted here by ∘
- **identity arrows**: identity function T => T, denoted here by $\mathbf{id_T}$

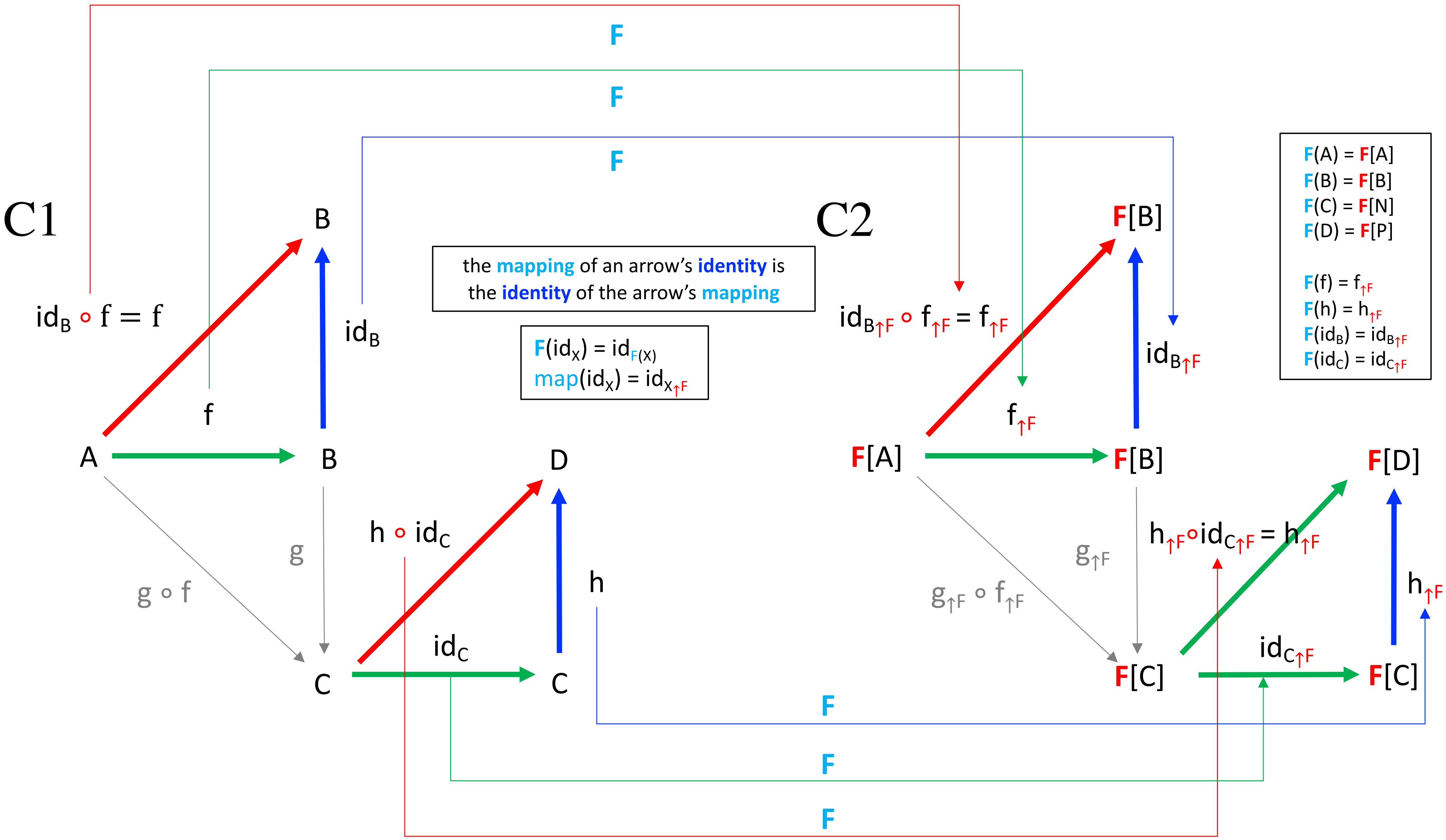
A functor F from C1 to C2 consisting of
- a type constructor **F** that maps type A to **F**[A]
- a map function from function f:A=>B to function $f_{\uparrow F}$:**F**[A] => **F**[B]

So **F**(g ∘ f) = **F**(g) ∘ **F**(f) becomes map(g ∘ f) = map(g) ∘ map(f)
and **F**(id_X) = $id_{F(X)}$ becomes map(id_X) = $id_{X_{\uparrow F}}$



F(A) = **F**[A]
F(B) = **F**[B]
F(C) = **F**[C]

F(f:A=>B) = map(f) = $f_{\uparrow F}$:**F**[A]=>**F**[B]
F(g:B=>C) = map(g) = $g_{\uparrow F}$:**F**[B]=>**F**[C]
F(g∘f:A=>C) = map(g∘f) = $g_{\uparrow F}$∘$f_{\uparrow F}$:**F**[A]=>**F**[C]



C1
C2
F
F
F
B
A
B
C
D
C
$id_B \circ f = f$
f
id_B
g
$g \circ f$
$h \circ id_C$
id_C
h
the mapping of an arrow's identity is
the identity of the arrow's mapping
$F(id_X) = id_{F(X)}$
$map(id_X) = id_{X\uparrow F}$
F[B]
F[A]
F[B]
F[C]
F[D]
F[C]
$id_{B\uparrow F} \circ f_{\uparrow F} = f_{\uparrow F}$
$f_{\uparrow F}$
$id_{B\uparrow F}$
$g_{\uparrow F}$
$g_{\uparrow F} \circ f_{\uparrow F}$
$h_{\uparrow F} \circ id_{C\uparrow F} = h_{\uparrow F}$
$id_{C\uparrow F}$
$h_{\uparrow F}$
F(A) = F[A]
F(B) = F[B]
F(C) = F[N]
F(D) = F[P]
$F(f) = f_{\uparrow F}$
$F(h) = h_{\uparrow F}$
$F(id_B) = id_{B\uparrow F}$
$F(id_C) = id_{C\uparrow F}$
F
F
F

Making the Scala example more concrete with an actual type constructor: Option

```
F(String) = Option[String]
F(Char) = Option[Char]
F(Int) = Option[Int]

F(f:String=>Char) = map(f) = f↑F: Option[String] => Option[Char]
F(g:Char=>Int) = map(g) = g↑F: Option[Char] => Option[Int]
F(g∘f:A=>C) = map(g∘f) = g∘f↑F: Option[String] => Option[Int]
```

```scala
val f: String => Char = x => x.head
val g: Char => Int = x => x.toInt
```

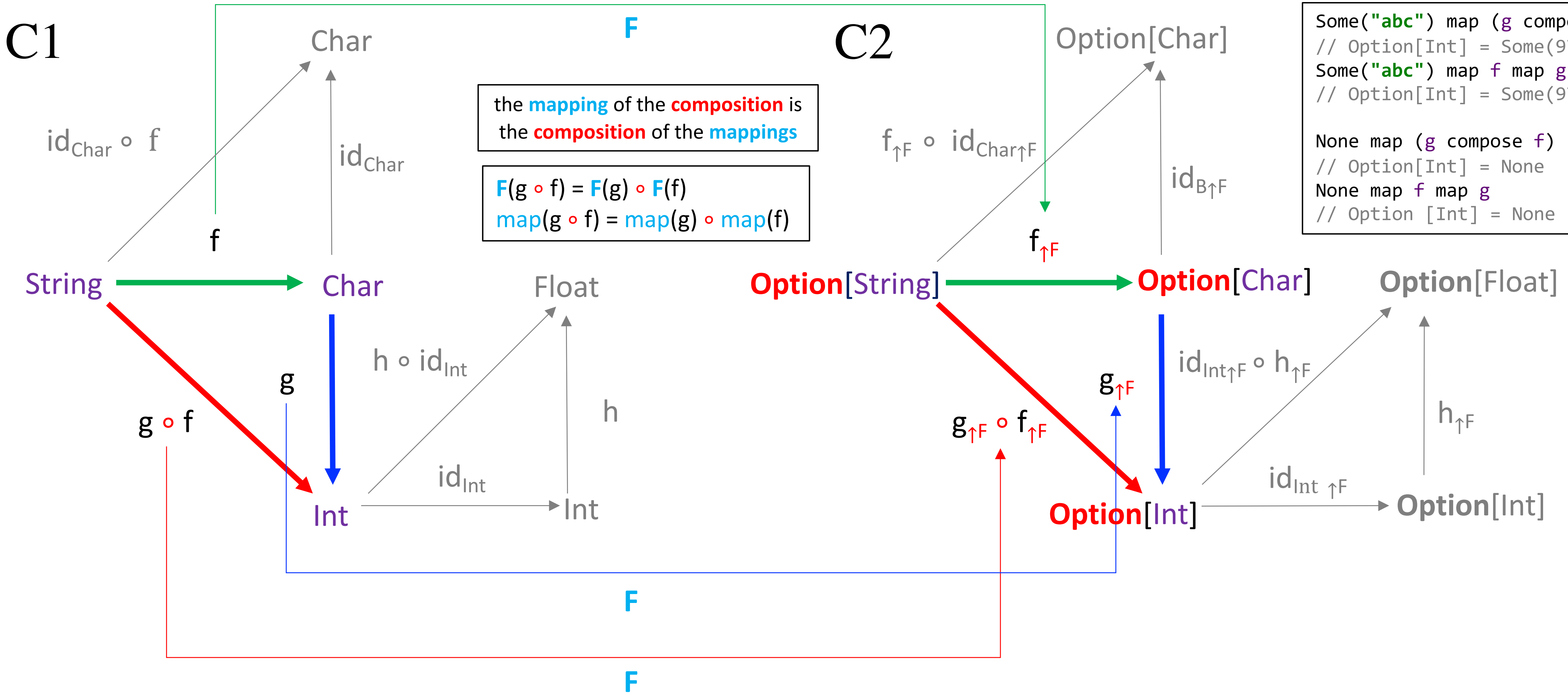
```scala
f("abc")
// Char = a
g('a')
// Int = 97
(g compose f)("abc")
// Int = 97
```

```scala
Some("abc") map (g compose f)
// Option[Int] = Some(97)
Some("abc") map f map g
// Option[Int] = Some(97)

None map (g compose f)
// Option[Int] = None
None map f map g
// Option [Int] = None
```

the mapping of the composition is the composition of the mappings

$F(g \circ f) = F(g) \circ F(f)$
$map(g \circ f) = map(g) \circ map(f)$



```
F(String) = Option[String]
F(Char) = Option[Char]
F(Int) = Option[Int]
```

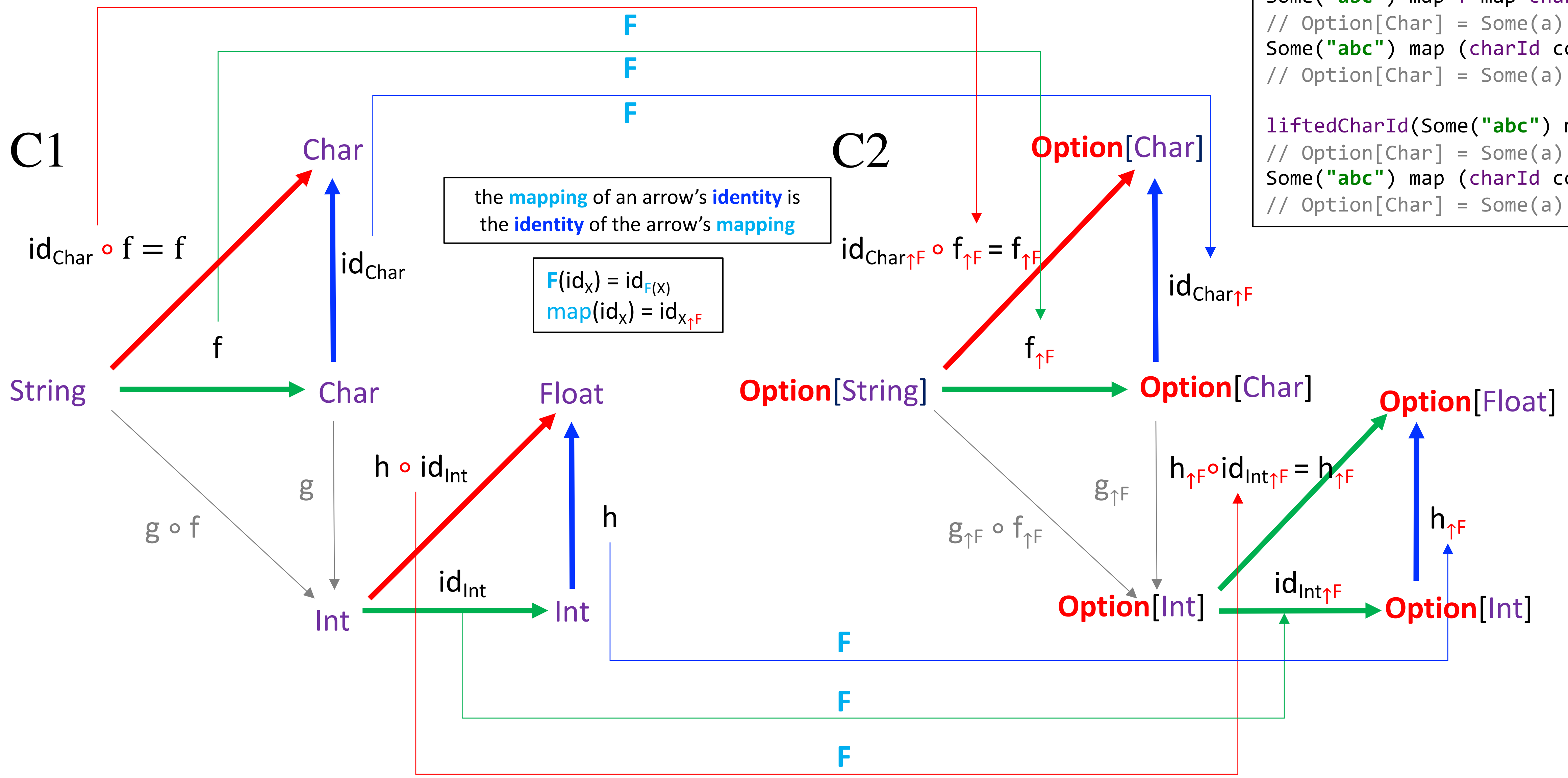
$F(f) = f_{\uparrow F}$
$F(h) = h_{\uparrow F}$
$F(id_{Int}) = id_{Int\uparrow F}$
$F(id_{Char}) = id_{Char\uparrow F}$

```scala
val f: String => Char = x => x.head
val charId: Char => Char = x => x
val liftedCharId: Option[Char] => Option[Char] = x => x
```

```scala
Some("abc") map f map charId
// Option[Char] = Some(a)
liftedCharId(Some("abc") map f)
// Option[Char] = Some(a)

Some("abc") map f map charId
// Option[Char] = Some(a)
Some("abc") map (charId compose f)
// Option[Char] = Some(a)

liftedCharId(Some("abc") map f)
// Option[Char] = Some(a)
Some("abc") map (charId compose f)
// Option[Char] = Some(a)
```



Scala Functor abstraction – ‘map method implementation’ and ‘functor laws in action’ for Option

The scala Functor abstraction typically looks like this:

```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B):F[B]
}
```

Examples of the functor laws in action are easier to follow if instead of using the customary signature of **map**, we rearrange it by first swapping its two parameters and then uncurrying the second parameter:

```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B):F[B]
}
```

customary signature

swapping of **fa** and **f** parameters

```
trait Functor[F[_]] {
  def map[A,B](f: A => B)(fa: F[A]):F[B]
}
```

after swapping parameters

uncurrying of **fa** parameter

```
trait Functor[F[_]] {
  def map[A,B](f: A => B):F[A] => F[B]
}
```

after uncurrying the second parameter

```scala
trait Functor[F[_]] {
  def map[A,B](f: A => B):F[A] => F[B]
}

val optionF = new Functor[Option]{
  def map[A,B](f: A => B):Option[A] => Option[B] = {
    case Some(a) => Some(f(a))
    case None => None
  }
}
```

Example of Functor abstraction and Functor laws in action, using the more convenient **map** signature.

The signature is more convenient in that it allows us to compose the mappings by literally performing their (function) composition.

the mapping of the composition is the composition of the mappings	$F(g \circ f) = F(g) \circ F(f)$ $map(g \circ f) = map(g) \circ map(f)$

```scala
val increment:Int=>Int = x => x + 1
val twice:Int=>Int = x => 2 * x
val arrow1 = increment
val arrow2 = twice
```

```scala
val mappingOfArrowComposition = optionF.map(arrow1 compose arrow2)
val compositionOfArrowMappings = optionF.map(arrow1) compose optionF.map(arrow2)
// mapping the composition of two arrows is the same as mapping the arrows and composing them
assert(mappingOfArrowComposition(Some(3)) == compositionOfArrowMappings(Some(3)))
assert(mappingOfArrowComposition(None) == compositionOfArrowMappings(None))
```

the mapping of an object's identity is the identity of the object's mapping	$F(id_X) = id_{F(X)}$ $map(id_X) = id_{X\uparrow F}$

```scala
val increment:Int=>Int = x => x + 1
val arrow = increment
val arrowOutputIdentity:Int=>Int = x => x
val arrowMappingOutputIdentity:Option[Int]=>Option[Int]=x=>x
```

```scala
val arrowMapping:Option[Int]=>Option[Int] = optionF.map(arrow)
val identityMapping:Option[Int]=>Option[Int] = optionF.map(arrowOutputIdentity)
// mapping the identity of an arrow's output type is the same as mapping
// the arrow and taking the identity of the result's output type: they have
// the same effect when composed with the mapping of the arrow
assert((identityMapping compose arrowMapping)(Some(3)) == (arrowMappingOutputIdentity compose arrowMapping)(Some(3)))
// mapping an arrow and the identity of its output type and composing them is the same
// as mapping the composition of the arrow with the identity of its output type
assert((identityMapping compose arrowMapping)(Some(3)) == optionF.map(arrowOutputIdentity compose arrow)(Some(3)))
// mapping an arrow and composing it with the identity of the result's output type is the same
// as mapping the composition of the arrow with the identity of its output type
assert((arrowMappingOutputIdentity compose arrowMapping)(Some(3)) == optionF.map(arrowOutputIdentity compose arrow)(Some(3)))
```

```scala
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B):F[B]
}

val optionF = new Functor[Option] {
  def map[A,B](fa:Option[A])(f: A => B): Option[B] =
    fa match {
      case Some(a) => Some(f(a))
      case None => None
    }
}
```

Example of Functor abstraction and Functor laws in action, using the customary **map** signature.

The signature is less convenient in that rather than allowing us to compose the mappings by literally performing their (function) composition, it requires us to do so by chaining map invocations and calling functions.

the mapping of the composition is the composition of the mappings	$F(g \circ f) = F(g) \circ F(f)$ $map(g \circ f) = map(g) \circ map(f)$

```scala
var mappingOfArrowComposition = optionF.map(Some(3))(arrow1 compose arrow2)
var compositionOfArrowMappings = optionF.map(optionF.map(Some(3))(arrow2))(arrow1)
// mapping the composition of two arrows is the same as mapping the arrows and composing them
assert(mappingOfArrowComposition == compositionOfArrowMappings)
mappingOfArrowComposition = optionF.map(None)(arrow1 compose arrow2)
compositionOfArrowMappings = optionF.map(optionF.map(None)(arrow2))(arrow1)
assert(mappingOfArrowComposition == compositionOfArrowMappings)
```

```scala
val increment:Int=>Int = x => x + 1
val twice:Int=>Int = x => 2 * x
val arrow1 = increment
val arrow2 = twice
```

the mapping of an arrow's identity is the identity of the arrow's mapping	$F(id_X) = id_{F(X)}$ $map(id_X) = id_{X\uparrow F}$

```scala
val appliedArrowMapping = optionF.map(Some(3))(arrow)
// mapping the identity of an arrow's output type is the same as mapping
// the arrow and taking the identity of the result's output type: they have
// the same effect when composed with the mapping of the arrow
assert(optionF.map(appliedArrowMapping)(arrowOutputIdentity) == arrowMappingOutputIdentity(appliedArrowMapping))
// mapping an arrow and the identity of its output type and composing them is the same
// as mapping the composition of the arrow with the identity of its output type
assert(optionF.map(appliedArrowMapping)(arrowOutputIdentity) == optionF.map(Some(3))(arrowOutputIdentity compose arrow))
// mapping an arrow and composing it with the identity of the result's output type is the same
// as mapping the composition of the arrow with the identity of its output type
assert(arrowMappingOutputIdentity(appliedArrowMapping) == optionF.map(Some(3))(arrowOutputIdentity compose arrow))
```

```scala
val increment:Int=>Int = x => x + 1
val arrow = increment
val arrowOutputIdentity:Int=>Int = x => x
val arrowMappingOutputIdentity:Option[Int]=>Option[Int] = x=>x
```

The Functor Laws mean that a Functor's map is structure preserving

"Only the elements of the structure are modified by `map`; the shape or structure itself is left intact."

11.1.1 Functor laws

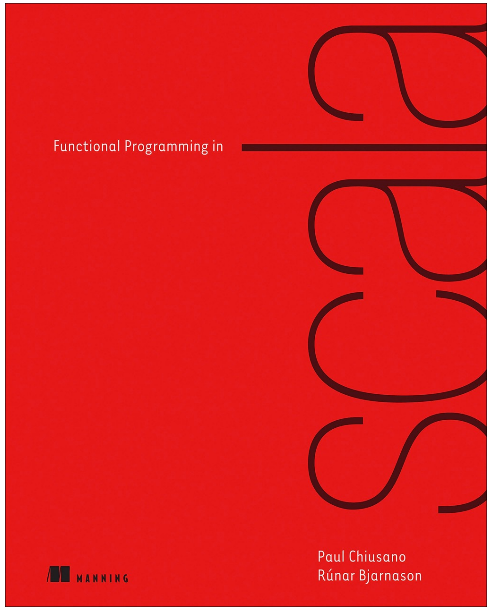
Whenever we create an abstraction like `Functor`, we should consider not only what abstract methods it should have, but which laws we expect to hold for the implementations. The laws you stipulate for an abstraction are entirely up to you, and of course Scala won't enforce any of these laws.

…

For `Functor`, we'll stipulate the familiar law we first introduced in chapter 7 for our `Par` data type:

```
map(x)(a => a) == x
```

In other words, **mapping over a structure `x` with the identity function should itself be an identity**. This law is quite natural, and we noticed later in part 2 that this law was satisfied by the `map` functions of other types besides `Par`. **This law** (and its corollaries given by parametricity) **capture the requirement that `map(x)` "preserves the structure" of `x`. Implementations satisfying this law are restricted from doing strange things like throwing exceptions, removing the first element of a `List`, converting a `Some` to `None`, and so on. Only the elements of the structure are modified by `map`; the shape or structure itself is left intact**. Note that this law holds for `List`, `Option`, `Par`, `Gen`, and most other data types that define `map`!



by Paul Chiusano and
Runar Bjarnason

https://twitter.com/runarorama

https://twitter.com/pchiusano

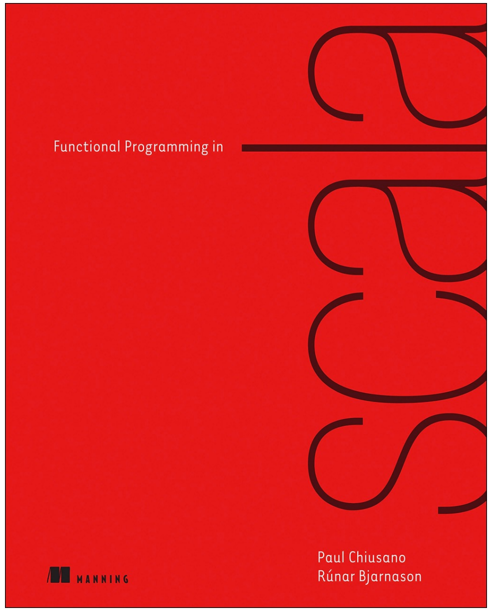
A Functor’s composition law is derivable from its identity law as a free theorem

`map(y)(id) == y`

…To get some insight into what this new law is saying, **let’s think about what map can’t do**. It can’t, say, throw an exception and crash the computation before applying the function to the result (can you see why this violates the law?). **All it can do is apply the function f to the result of y, which of course leaves y unaffected when that function is id.**[11]

Even more interestingly, given `map(y)(id) == y`
…
it must be true that `map(unit(x))(f) == unit(f(x))` . Since **we get this second law or theorem for free**, simply because of the parametricity of map , **it’s sometimes called a free theorem**.[12]

EXERCISE 7.7
Hard: Given `map(y)(id) == y`, it’s **a free theorem** that `map(map(y)(g))(f) == map(y)(f compose g)`. (**This is sometimes called map fusion, and it can be used as an optimization—rather than spawning a separate parallel computation to compute the second mapping, we can fold it into the first mapping**.)[13] Can you prove it? You may want to read the paper “Theorems for Free!” (http://mng.bz/Z9f1) to better understand the “trick” of **free theorems**.
…
[11] We say that **map is required to be structure-preserving** in that **it doesn’t alter the structure** of the parallel computation, **only the value “inside”** the computation.
[12] The idea of free theorems was introduced by Philip Wadler in the classic paper “Theorems for Free!” (http://mng.bz/Z9f1).



by Paul Chiusano and
Runar Bjarnason

https://twitter.com/**runarorama**

https://twitter.com/**pchiusano**