Chapter 2

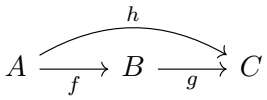
Composition

2.1 Composition

Programming is about composition. Paraphrasing Wittgenstein, one could say: "Of that which cannot be decomposed one should not speak." This is not a prohibition, it's a statement of fact. The process of studying, understanding, and describing is the same as the process of decomposing; and our language reflects this.

The reason we have built the vocabulary of objects and arrows is precisely to express the idea of composition.

Given an arrow f from A to B and an arrow g from B to C, their composition is an arrow that goes directly from A to C. In other words, if there are two arrows, the target of one being the same as the source of the other, we can always compose them to get a third arrow.



In math we denote composition using a little circle

$$h = g \circ f$$

We read this: "h is equal to g after f." The order of composition might seem backward, but this is because we think of functions as taking arguments on the right. In Haskell we replaced the circle with a dot:

```
h = g . f
```

This is every program in a nutshell. In order to accomplish `h`, we decompose it into simpler problems, `f` and `g`. These, in turn, can be decomposed further, and so on.

Now suppose that we were able to decompose g itself into $j \circ k$. We have

$$h = (j \circ k) \circ f$$

We want this decomposition to be the same as

$$h = j \circ (k \circ f)$$

We want to be able to say that we have decomposed h into three simpler problems

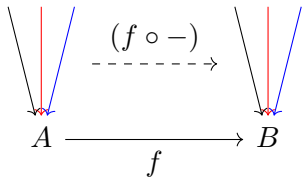
$$h = j \circ k \circ f$$

and not have to keep track which decomposition came first. This is called *associativity* of composition, and we will assume it from now on.

Composition is the source of two mappings of arrows called pre-composition and post-composition.

When an arrow f is post-composed after an arrow h, it produces the arrow $f \circ h$. Of course, f can be post-composed only after arrows whose target is the source of f. Post-composition by f is written as $(f \circ -)$, leaving a hole for h. As Lao Tzu would say, "Usefulness of post-composition comes from what is not there."

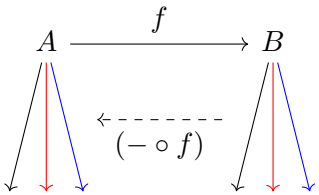
Thus an arrow $f\colon A \to B$ induces a mapping of arrows $(f \circ -)$ that maps arrows which are probing A to arrows which are probing B.



Since objects have no internal structure, when we say that f transforms A to B, this is exactly what we mean.

Post-composition lets us shift focus from one object to another.

Dually, you can pre-compose f, or apply $(- \circ f)$ to arrows originating in B and map them to arrows originating in A.



Pre-composition let us shift the perspective from one observer to another. Notice that the outgoing arrows are mapped in the direction opposite to the shifting arrow f.

Another way of looking at pre- and post-composition is that they are the result of partial application of the two-hole composition operator $(- \circ -)$, in which we pre-fill one hole or the other with an arrow.

In programming, an outgoing arrow is interpreted as extracting data from the source. An incoming arrow is interpreted as producing or constructing the target. Outgoing arrows define the interface, incoming arrows define the constructors.

Do the following exercises to convince yourself that shifts in focus and perspective are composable.

Exercise 2.1.1. *Suppose that you have two arrows, $f\colon A \to B$ and $g\colon B \to C$. Their composition $g \circ f$ induces a mapping of arrows $((g \circ f) \circ -)$. Show that the result is the same if you first apply $(f \circ -)$ and follow it by $(g \circ -)$. Symbolically:*

$$((g \circ f) \circ -) = (g \circ -) \circ (f \circ -)$$

Hint: Pick an arbitrary object X and an arrow $h\colon X \to A$ and see if you get the same result.

Exercise 2.1.2. *Convince yourself that the composition from the previous exercise is associative. Hint: Start with three composable arrows.*

Exercise 2.1.3. *Show that pre-composition $(- \circ f)$ is composable, but the order of composition is reversed:*

$$(- \circ (g \circ f)) = (- \circ f) \circ (- \circ g)$$

2.2 Function application

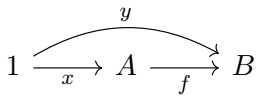
We are ready to write our first program. There is a saying: "A journey of a thousand miles begins with a single step." Our journey is from 1 to B. The single step is an arrow from the terminal object 1 to A. It's an element of A. We can write it as

$$1 \xrightarrow{x} A$$

The rest of the journey is the arrow

$$A \xrightarrow{f} B$$

These two arrows are composable (they share the object A) and their composition is the arrow y from 1 to B. In other words, y is an *element* of B



We can write it as

$$y = f \circ x$$

We used f to map an *element* of A to an *element* of B. Since this is something we do quite often, we call it the *application* of a function f to x, and use the shorthand notation

$$y = fx$$

Let's translate it to Haskell. We start with an element of A (a shorthand for `()->A`)

```
x :: A
```

We declare the function f as an element of the "object of arrows" from A to B

```
f :: A -> B
```

with the understanding (which will be elaborated upon later) that it corresponds to an arrow from `A` to `B`. The result is an element of B

```
y :: B
```

and it is defined as

```
y = f x
```

We call this the application of a function to an argument, but we expressed it purely in terms of function composition. (Note: In other programming languages function application requires the use of parentheses, e.g., `y = f(x)`.)

2.3 Identity

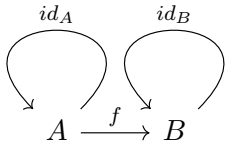
You may think of arrows as representing change: object A becomes object B. An arrow that loops back represents a change in an object itself. But change has its dual: lack of change, inaction or, as Lao Tzu would say *wu wei*.

Every object has a special arrow called the identity, which leaves the object unchanged. It means that, when you compose this arrow with any other arrow, either incoming or outgoing, you get that arrow back. Identity arrow does nothing to an arrow.

An identity arrow on the object A is called id_A. So if we have an arrow $f\colon A \to B$, we can compose it with identities on either side

$$id_B \circ f = f = f \circ id_A$$

or, pictorially:



We can easily check what an identity does to elements. Let's take an element $x\colon 1 \to A$ and compose it with id_A. The result is:

$$id_A \circ x = x$$

which means that identity leaves elements unchanged.

In Haskell, we use the same name `id` for all identity functions (we don't subscript it with the type it's acting on). The above equation, which specifies the action of id on elements, translates directly to

```
id x = x
```

and it becomes the definition of the function `id`.

We've seen before that both the initial object and the terminal object have unique arrows circling back to them. Now we are saying that every object has an identity arrow circling back to it. Remember what we said about uniqueness: If you can find two of those, then they must be equal. We must conclude that these unique looping arrows we talked about must be the identity arrows. We can now label these diagrams:

In logic, identity arrow translates to a tautology. It's a trivial proof that, "if A is true then A is true." It's also called the *identity rule*.

If identity does nothing then why do we care about it? Imagine going on a trip, composing a few arrows, and finding yourself back at the starting point. The question is: Have you done anything, or have you wasted your time? The only way to answer this question is to compare your path with the identity arrow.

Some round trips bring change, others don't.

Exercise 2.3.1. *What does* $(id_A \circ -)$ *do to arrows terminating in* A*? What does* $(- \circ id_A)$ *do to arrows originating from* A*?*