# Scalable program architectures

Haskell design patterns differ from mainstream design patterns in one important way:

- **Conventional architecture:** Combine a several components together of type `A` to generate a "network" or "topology" of type `B`

- **Haskell architecture:** Combine several components together of type `A` to generate a new component of the same type `A`, indistinguishable in character from its substituent parts

This distinction affects how the two architectural styles evolve as code bases grow.

The conventional architecture requires layering abstraction on top of abstraction:

> Oh no, these `B`s are not connectable, so let's make a network of `B`s and call that a `C`.
>
> Well, I want to assemble several `C`s, so let's make a network of `C`s and call that a `D`
>
> ...

Wash, rinse, and repeat until you have an unmanageable tower of abstractions.

With a Haskell-style architecture, you don't need to keep layering on abstractions to preserve combinability. When you combine things together the result is still itself combinable. You don't distinguish between components and networks of components.

In fact, this principle should be familiar to anybody who knows basic arithmetic. When you combine a bunch of numbers together you get back a number:

```
3 + 4 + 9 = 16
```

Zero or more numbers go in and exactly one number comes out. The resulting number is itself combinable. You don't have to learn about "web"s of numbers or "web"s of "web"s of numbers.

If elementary school children can master this principle, then perhaps we can, too. How can we make programming more like addition?

Well, addition is simple because we have `(+)` and `0`. `(+)` ensures that we can always convert **more than** one number into exactly number:

```
(+) :: Int -> Int -> Int
```

... and `0` ensures that we can always convert **less than** one number into exactly one number by providing a suitable default:

```
0 :: Int
```

This will look familiar to Haskell programmers: these type signatures resemble the methods of the `Monoid` type class:

```
class Monoid m where
    -- `mappend` is analogous to `(+)`
    mappend :: m -> m -> m

    -- `mempty` is analogous to `0`
    mempty  :: m
```

In other words, the `Monoid` type class is the canonical example of this Haskell architectural style. We use `mappend` and `mempty` to combine 0 or more `m`s into exactly 1 `m`. The resulting `m` is still combinable.

Not every Haskell abstraction implements `Monoid`, nor do they have to because category theory takes this basic `Monoid` idea and generalizes it to more powerful domains. Each generalization retains the same basic principle of preserving combinability.

For example, a `Category` is just a typed monoid, where not all combinations type-check:

```
class Category cat where
    -- `(.)` is analogous to `(+)`
    (.) :: cat b c -> cat a b -> cat a c
    -- `id` is analogous to `0`
    id  :: cat a a
```

... a `Monad` is like a monoid where we combine functors "vertically":

```
-- Slightly modified from the original type class
class Functor m => Monad m where
    -- `join` is analogous to `(+)`
    join :: m (m a) -> m a

    -- `return` is analogous to `0`
    return :: a -> m a
```

... and an `Applicative` is like a monoid where we combine functors "horizontally":

```
-- Greatly modified, but equivalent to, the original type class
class Functor f => Applicative f where
    -- `mult` is is analogous to `(+)`
    mult :: f a -> f b -> f (a, b)

    -- `unit` is analogous to `0`
    unit :: f ()
```

Category theory is full of generalized patterns like these, all of which try to preserve that basic intuition we had for addition. We convert **more than** one *thing* into exactly one *thing* using something that resembles addition and we convert **less than** one *thing* into exactly one *thing* using something that resembles zero. Once you learn to think in terms of these patterns, programming becomes as simple as basic arithmetic: combinable components go in and exactly one combinable component comes out.

These abstractions scale limitlessly because they always preserve combinability, therefore we never need to layer further abstractions on top. This is one reason why you should learn Haskell: you learn how to build flat architectures.