Haskellize your C++

Enrico Maria

If you want your C++ to look like Haskell, a few libraries might help...

March 24, 2021

Let's take it easy

What is this presentation for?

- Writing some LaTeX after a long time...
- Showing that FP is possible in C++, to some extent
- Getting feedback on what my level is
- Pique your interest for C++ from a Haskell perspective

Starting off with something easy

We need some simple Haskell code that we want to reproduce in C++

- Let's take a simple list...

```
xs :: [Int]
xs = [1..10]
```

- ...and map a function on it using `fmap`:

```
ys = fmap (+3) xs
-- ys has not been computed yet
ys == [4..13] -- True (now ys is computed)
```

Can't you see? I'm the Unix Piper

- How do we write something similar in C++?

```
ys = fmap (+3) [1..10]
print ys
```

Can't you see? I'm the Unix Piper

- How do we write something similar in C++?

```
ys = fmap (+3) [1..10]
print ys
```

- `std::transform`? No, that works with `begin`/`end` iterators.

Can't you see? I'm the Unix Piper

- How do we write something similar in C++?

```
ys = fmap (+3) [1..10]
print ys
```

- `std::transform`? No, that works with `begin`/`end` iterators.
- Range-v3 to the rescue! Ready...

```
#include <range/v3/view/iota.hpp>
#include <range/v3/view/transform.hpp>
namespace rv = ranges::views;
auto constexpr plus3 = [](auto x){ return x + 3; };
```

Can't you see? I'm the Unix Piper

- How do we write something similar in C++?

```
ys = fmap (+3) [1..10]
print ys
```

- `std::transform`? No, that works with `begin`/`end` iterators.
- Range-v3 to the rescue! Ready...

```
#include <range/v3/view/iota.hpp>
#include <range/v3/view/transform.hpp>
namespace rv = ranges::views;
auto constexpr plus3 = [](auto x){ return x + 3; };
```

- Range-v3 to the rescue! Go!

```
auto ys = rv::iota(1,11) | rv::transform(plus3);
std::cout << ys << std::endl;
// prints [4,5,6,7,8,9,10,11,12,13], literally
```

Can't you see? I'm the Unix Piper

```
auto ys = rv::iota(1,11) | rv::transform(plus3);
```

Pros:

Cons:

Can't you see? I'm the Unix Piper

```
auto ys = rv::iota(1,11) | rv::transform(plus3);
```

Pros:

- Clear Linux-like pipe syntax conveying left-to-right flow

Cons:

Can't you see? I'm the Unix Piper

```
auto ys = rv::iota(1,11) | rv::transform(plus3);
```

Pros:

- Clear Linux-like pipe syntax conveying left-to-right flow
- Guess what? It's a cheap lazy view; computation is deferred

Cons:

Can't you see? I'm the Unix Piper

```
auto ys = rv::iota(1,11) | rv::transform(plus3);
```

Pros:

- Clear Linux-like pipe syntax conveying left-to-right flow
- Guess what? It's a cheap lazy view; computation is deferred
- Easy to get a concrete container out of it

```
#include <range/v3/range/conversion.hpp>
namespace r = ranges;
auto ys = rv::iota(1,11) | rv::transform(plus3)
                         | r::to_vector;
```

Cons:

Can't you see? I'm the Unix Piper

```
auto ys = rv::iota(1,11) | rv::transform(plus3);
```

Pros:

- Clear Linux-like pipe syntax conveying left-to-right flow
- Guess what? It's a cheap lazy view; computation is deferred
- Easy to get a concrete container out of it

```
#include <range/v3/range/conversion.hpp>
namespace r = ranges;
auto ys = rv::iota(1,11) | rv::transform(plus3)
                         | r::to_vector;
```

- Supports syntax `rv::transform(xs, f)` (so it's a flipped `fmap`...)

Cons:

Can't you see? I'm the Unix Piper

```
auto ys = rv::iota(1,11) | rv::transform(plus3);
```

Pros:

- Clear Linux-like pipe syntax conveying left-to-right flow
- Guess what? It's a cheap lazy view; computation is deferred
- Easy to get a concrete container out of it

```
#include <range/v3/range/conversion.hpp>
namespace r = ranges;
auto ys = rv::iota(1,11) | rv::transform(plus3)
                         | r::to_vector;
```

- Supports syntax `rv::transform(xs, f)` (so it's a flipped `fmap`...)

Cons:

- Cannot pass rvalue *containers* through the pipe (ranges are ok)

Can't you see? I'm the Unix Piper

```
auto ys = rv::iota(1,11) | rv::transform(plus3);
```

Pros:

- Clear Linux-like pipe syntax conveying left-to-right flow
- Guess what? It's a cheap lazy view; computation is deferred
- Easy to get a concrete container out of it

```
#include <range/v3/range/conversion.hpp>
namespace r = ranges;
auto ys = rv::iota(1,11) | rv::transform(plus3)
                         | r::to_vector;
```

- Supports syntax `rv::transform(xs, f)` (so it's a flipped `fmap`...)

Cons:

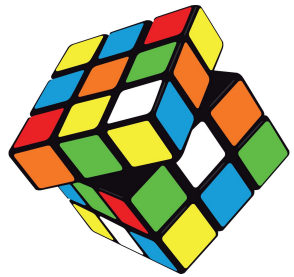
- Cannot pass rvalue *containers* through the pipe (ranges are ok)
- TPOIASI... (in a couple of slides)

Can't you see? I'm the Unix Piper

In case we have doubts that Range-v3 uses lazy views...

```
#include <range/v3/view/take.hpp>
#include <range/v3/view/zip_with.hpp>

auto r1 = rv::iota(1) | rv::take(10);
auto r2 = rv::iota(11); // semi-infinite range

auto divAsDoubles = [](int x, int y){
  return (double)x / y;
}; // we'll do better...

auto r12 = rv::zip_with(divAsDoubles, r1, r2);
std::cout << r12 << std::endl;
// prints [0.0909091, 0.166667, 0.230769, ...
```

TPOIASI (article on Fluent{C++})

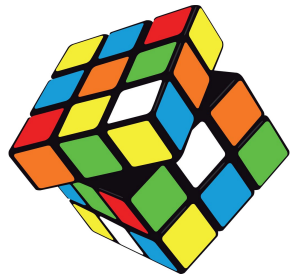


Terrible Problem Of Incrementing A Smart Iterator

- Simple usecase...

```
auto constexpr isMultipleOf4 = [](int x){ return 0 ==
↪   x % 4; };
auto constexpr times2 = [](int x){
    std::cout << "transforming " << x << std::endl;
    return 2*x;
};
auto v = rv::iota(1,6) | rv::transform(times2)
                       | rv::filter(isMultipleOf4);
for (auto i : v) {/* to trigger the computation */}
```

TPOIASI (article on Fluent{C++})



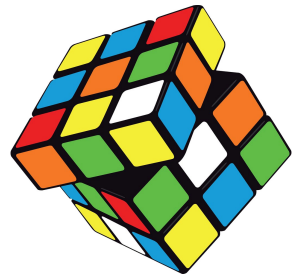
Terrible Problem Of Incrementing A Smart Iterator

- Simple usecase...

```
auto constexpr isMultipleOf4 = [](int x){ return 0 ==
↪   x % 4; };
auto constexpr times2 = [](int x){
    std::cout << "transforming " << x << std::endl;
    return 2*x;
};
auto v = rv::iota(1,6) | rv::transform(times2)
                       | rv::filter(isMultipleOf4);
for (auto i : v) {/* to trigger the computation */}
```

- What's the output?

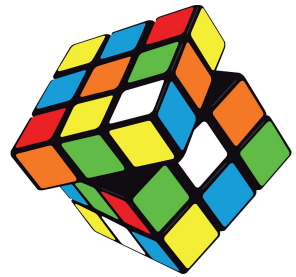
TPOIASI



- Here it is:

```
transforming 1
transforming 2
transforming 2 ← duplicate!
transforming 3
transforming 4
transforming 4 ← duplicate!
transforming 5
```

TPOIASI

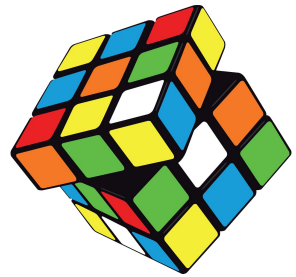


- Here it is:

```
transforming 1
transforming 2
transforming 2 ← duplicate!
transforming 3
transforming 4
transforming 4 ← duplicate!
transforming 5
```

- Why? Long story short:
  - `filter`'s `operator++` calls `transform`'s `operator*`
  - `filter`'s `operator*` calls, again, `transform`'s `operator*`

TPOIASI



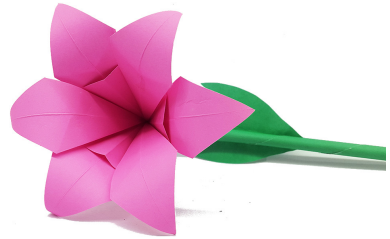
- Here it is:

```
transforming 1
transforming 2
transforming 2 ← duplicate!
transforming 3
transforming 4
transforming 4 ← duplicate!
transforming 5
```

- Why? Long story short:
  - `filter`'s `operator++` calls `transform`'s `operator*`
  - `filter`'s `operator*` calls, again, `transform`'s `operator*`
- Solution: caching the result of the `transform`

```
#include <range/v3/view/cache1.hpp>
auto v = rv::iota(1,6) | rv::transform(times2)
                       | rv::cache1
                       | rv::filter(isMultipleOf4);
```

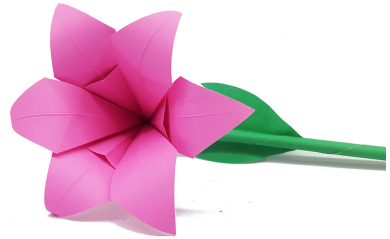
Meet Miss Hana



- The Hana way of `transforming`

```
#include <boost/hana/transform.hpp>
std::vector<int> v = rv::iota(1)
                   | rv::take(10)
                   | r::to_vector;

auto w = hana::transform(v, times3); // flipped fmap
```

Meet Miss Hana



- The Hana way of `transforming`

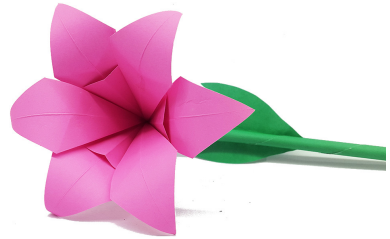
```
#include <boost/hana/transform.hpp>
std::vector<int> v = rv::iota(1)
                   | rv::take(10)
                   | r::to_vector;

auto w = hana::transform(v, times3); // flipped fmap
```

- . . . hmmm. . . actually to get the above work, we have to copy the `Functor` instance of `std::vector` from

```
#include <boost/hana/ext/std/vector.hpp>
```

which is commented out (Jason Rice on Gitter: *« I can only guess it is unfinished. Maybe it needed more tests.»*)

Meet Miss Hana



- The Hana way of `transforming`

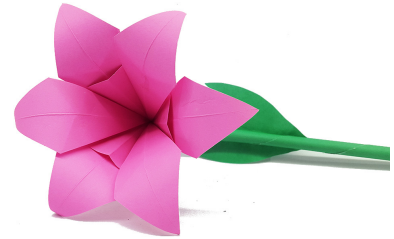
```
#include <boost/hana/transform.hpp>
std::vector<int> v = rv::iota(1)
                   | rv::take(10)
                   | r::to_vector;

auto w = hana::transform(v, times3); // flipped fmap
```

- . . . hmmm. . . actually to get the above work, we have to copy the `Functor` instance of `std::vector` from

```
#include <boost/hana/ext/std/vector.hpp>
```

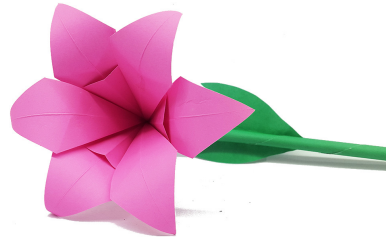
  which is commented out (Jason Rice on Gitter: *« I can only guess it is unfinished. Maybe it needed more tests.»*)
- What about `Applicative` and `Monad`, by the way? . . .

Meet Miss Hana



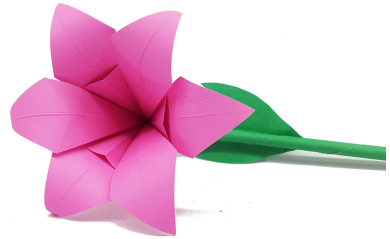
- What about `Applicative` for `std::vector`?

Meet Miss Hana



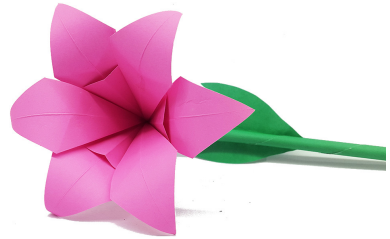
- What about `Applicative` for `std::vector`?
- Haskell's `pure` $\approx$ `std::vector`'s constructor,

Meet Miss Hana



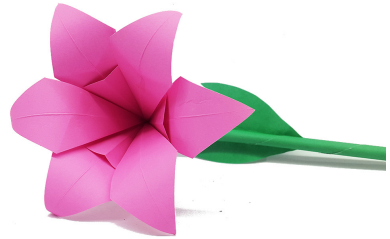
- What about `Applicative` for `std::vector`?
- Haskell's `pure` $\approx$ `std::vector`'s constructor,
- Haskell's `(<*>)` $\approx$???

Meet Miss Hana



- What about `Applicative` for `std::vector`?
- Haskell's `pure` $\approx$ `std::vector`'s constructor,
- Haskell's `(<*>)` $\approx$???
- We have a problem. How do we put different functions (with "only" the signature in common) into a `std::vector`?

Meet Miss Hana



- What about `Applicative` for `std::vector`?
- Haskell's `pure` ≈ `std::vector`'s constructor,
- Haskell's `(<*>)` ≈ ???
- We have a problem. How do we put different functions (with "only" the signature in common) into a `std::vector`?
- Honestly, I have no idea, but the following topics come to my mind:
  - type erasure
  - `std::function` (does it use type erasure?...)

More difficult

- Let's take a simple list of lists

```
xss :: [[Int]]
xss = [[1,2,3],[4],[5,6]]
```

More difficult

- Let's take a simple list of lists

```
xss :: [[Int]]
xss = [[1,2,3],[4],[5,6]]
```

- How do we map a function, e.g. `(+3)`, on it?

More difficult

- Let's take a simple list of lists

```
xss :: [[Int]]
xss = [[1,2,3],[4],[5,6]]
```

- How do we map a function, e.g. `(+3)`, on it?
- Simple, we use `fmap . fmap`!

```
yss = (fmap . fmap) (+3) xss
yss == [[4,5,6],[7],[8,9]]
```

More difficult

What do we need for "this" syntax to do the same in C++?

```
(fmap . fmap) (+3) xss
```

Remember that

```
fmap :: (a -> b) -> f a -> f b
```

right `fmap` gets `a -> b` and feeds `f a -> f b` to left `fmap`.

Therefore we need:

More difficult

What do we need for "this" syntax to do the same in C++?

```
(fmap . fmap) (+3) xss
```

Remember that

```
fmap :: (a -> b) -> f a -> f b
```

right `fmap` gets `a -> b` and feeds `f a -> f b` to left `fmap`.

Therefore we need:

- an `fmap`-like function,

More difficult

What do we need for "this" syntax to do the same in C++?

```
(fmap . fmap) (+3) xss
```

Remember that

```
fmap :: (a -> b) -> f a -> f b
```

right `fmap` gets `a -> b` and feeds `f a -> f b` to left `fmap`.

Therefore we need:

- an `fmap`-like function,
- since we do have `hana::transform`, we need also a `flip` function,

More difficult

What do we need for "this" syntax to do the same in C++?

```
(fmap . fmap) (+3) xss
```

Remember that

```
fmap :: (a -> b) -> f a -> f b
```

right `fmap` gets `a -> b` and feeds `f a -> f b` to left `fmap`.

Therefore we need:

- an `fmap`-like function,
- since we do have `hana::transform`, we need also a `flip` function,
- a `(.)`-like operator for function composition,

More difficult

What do we need for "this" syntax to do the same in C++?

```
(fmap . fmap) (+3) xss
```

Remember that

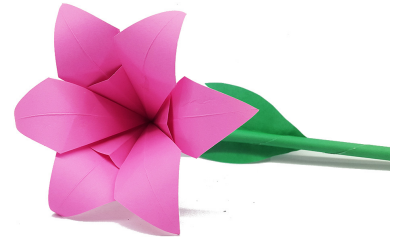
```
fmap :: (a -> b) -> f a -> f b
```

right `fmap` gets `a -> b` and feeds `f a -> f b` to left `fmap`.

Therefore we need:

- an `fmap`-like function,
- since we do have `hana::transform`, we need also a `flip` function,
- a `(.)`-like operator for function composition,
- currying to appropriately compose non-unary functions (such as `fmap`).

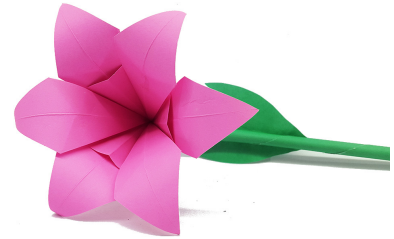
Hana to the rescue!



Needs:

What does Boost.Hana offer of what we need?

Hana to the rescue!

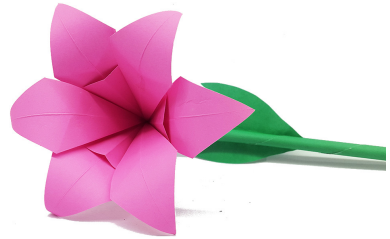


Needs:

- an `fmap`-like function,

What does Boost.Hana offer of what we need?

Hana to the rescue!



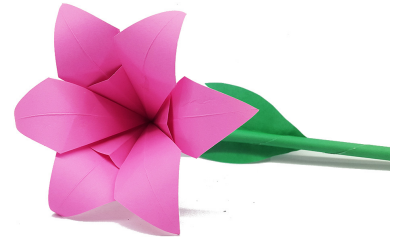
Needs:

- an `fmap`-like function,

What does Boost.Hana offer of what we need?

- `#include <boost/hana/transform.hpp>`

Hana to the rescue!



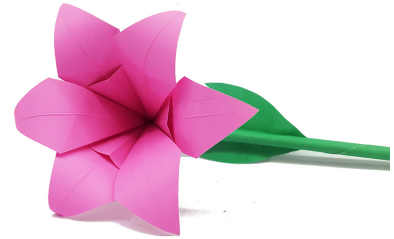
Needs:

- an `fmap`-like function,
- since we do have `hana::transform`, we need also a `flip` function,

What does Boost.Hana offer of what we need?

- `#include <boost/hana/transform.hpp>`

Hana to the rescue!



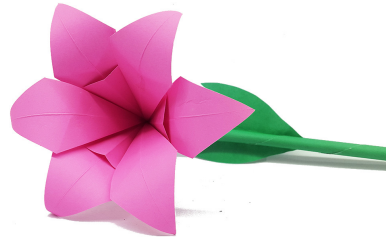
Needs:

- an `fmap`-like function,
- since we do have `hana::transform`, we need also a `flip` function,

What does Boost.Hana offer of what we need?

- `#include <boost/hana/transform.hpp>`
- `#include <boost/hana/functional/flip.hpp>`

Hana to the rescue!



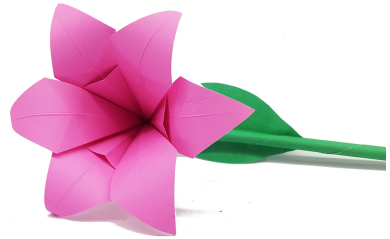
Needs:

- an `fmap`-like function,
- since we do have `hana::transform`, we need also a `flip` function,
- a `(.)`-like operator for function composition,

What does Boost.Hana offer of what we need?

- `#include <boost/hana/transform.hpp>`
- `#include <boost/hana/functional/flip.hpp>`

Hana to the rescue!



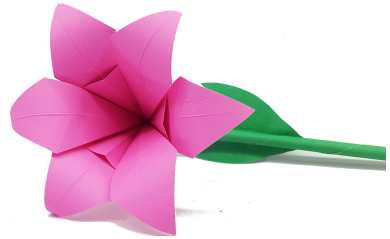
Needs:

- an `fmap`-like function,
- since we do have `hana::transform`, we need also a `flip` function,
- a `(.)`-like operator for function composition,

What does Boost.Hana offer of what we need?

- `#include <boost/hana/transform.hpp>`
- `#include <boost/hana/functional/flip.hpp>`
- `#include <boost/hana/functional/compose.hpp>`

Hana to the rescue!



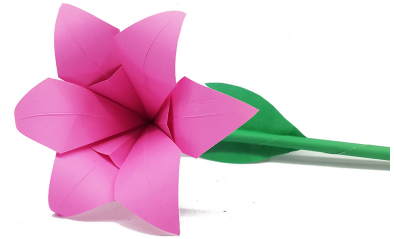
Needs:

- an `fmap`-like function,
- since we do have `hana::transform`, we need also a `flip` function,
- a `(.)`-like operator for function composition,
- currying to appropriately compose non-unary functions (such as `fmap`).

What does Boost.Hana offer of what we need?

- `#include <boost/hana/transform.hpp>`
- `#include <boost/hana/functional/flip.hpp>`
- `#include <boost/hana/functional/compose.hpp>`

Hana to the rescue!



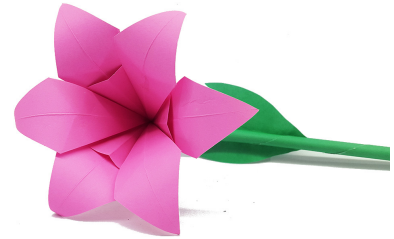
Needs:

- an `fmap`-like function,
- since we do have `hana::transform`, we need also a `flip` function,
- a `(.)`-like operator for function composition,
- currying to appropriately compose non-unary functions (such as `fmap`).

What does Boost.Hana offer of what we need?

- `#include <boost/hana/transform.hpp>`
- `#include <boost/hana/functional/flip.hpp>`
- `#include <boost/hana/functional/compose.hpp>`
- `#include <boost/hana/functional/curry.hpp>`

Hana to the rescue!



Needs:

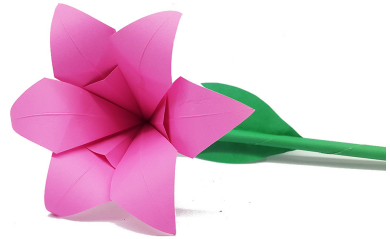
- an `fmap`-like function,
- since we do have `hana::transform`, we need also a `flip` function,
- a `(.)`-like operator for function composition,
- currying to appropriately compose non-unary functions (such as `fmap`).

What does Boost.Hana offer of what we need?

- `#include <boost/hana/transform.hpp>`
- `#include <boost/hana/functional/flip.hpp>`
- `#include <boost/hana/functional/compose.hpp>`
- `#include <boost/hana/functional/curry.hpp>`

Simply all!

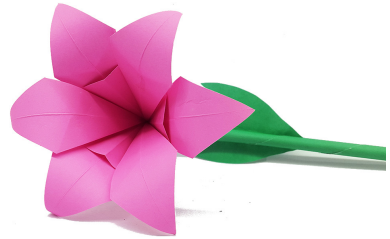
Hana to the rescue!



- Natural solution: imitate Haskell's `fmap`

```
using namespace boost::hana;
std::vector<std::vector<int>> xss{{1,2,3},{4},{5,6}};

auto constexpr fmap = curry<2>(flip(transform));
auto yss = compose(fmap,fmap)(plus3)(xss);
// compose(f, g)(x, y...) == f(g(x), y...)
```

Hana to the rescue!



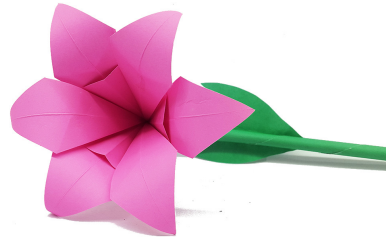
- Natural solution: imitate Haskell's `fmap`

```
using namespace boost::hana;
std::vector<std::vector<int>> xss{{1,2,3},{4},{5,6}};

auto constexpr fmap = curry<2>(flip(transform));
auto yss = compose(fmap,fmap)(plus3)(xss);
// compose(f, g)(x, y...) == f(g(x), y...)
```

- Actually, `on` is more powerful and nicer

```
#include <boost/hana/functional/on.hpp>
auto yss = (fmap ^on^ fmap)(plus3)(xss);
// on(f, g)(x...) == f(g(x)...)
```

Hana to the rescue!



- Natural solution: imitate Haskell's `fmap`

```
using namespace boost::hana;
std::vector<std::vector<int>> xss{{1,2,3},{4},{5,6}};

auto constexpr fmap = curry<2>(flip(transform));
auto yss = compose(fmap,fmap)(plus3)(xss);
// compose(f, g)(x, y...) == f(g(x), y...)
```

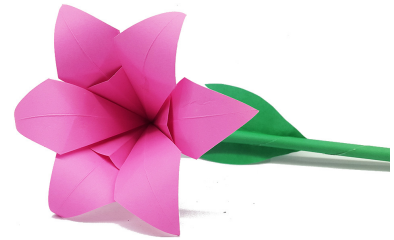
- Actually, `on` is more powerful and nicer

```
#include <boost/hana/functional/on.hpp>
auto yss = (fmap ^on^ fmap)(plus3)(xss);
// on(f, g)(x...) == f(g(x)...)
```

- Compare with Haskell:

```
(fmap . fmap) (+3) xss
```

More from Hana



Hana also offers partial function application:

```
#include <boost/hana/functional/partial.hpp>
#include <boost/hana/functional/reverse_partial.hpp>
```

so these

```
auto constexpr plus3 = [](auto x){ return x + 3; };
auto divAsDoubles = [](int x, int y){
  return (double)x / y;
};
auto r12 = rv::zip_with(divAsDoubles, r1, r2);
```

can be rewritten as this

```
auto constexpr plus3 = partial(std::plus<>{}, 3);
auto constexpr div = std::divides<>{};
auto constexpr mult = curry<2>(std::multiplies<>{});
auto r12 = rv::zip_with(div ^on^ mult(1.0), r1, r2);
```

I hope you enjoyed

Thank you!