C++ now

2026

Functional Composition Beyond Monads

Jonathan Müller

Functional composition

Composing functions

Composition Given `f(T) -> U` and `g(U) -> V`, the composition is `h(T) -> V` with `h(x) = g(f(x))`.

Composing functions

Composition Given `f(T) -> U` and `g(U) -> V`, the composition is `h(T) -> V` with `h(x) = g(f(x))`.

```
User    lookup_user(UserID id); // f
Address get_address(User user); // g
```

Composing functions

Composition Given `f(T) -> U` and `g(U) -> V`, the composition is `h(T) -> V` with `h(x) = g(f(x))`.

```
User    lookup_user(UserID id); // f
Address get_address(User user); // g

Address lookup_address(UserId id) { // h
    return get_address(lookup_user(id));
}
```

A composition operator

```
auto operator|(auto&& f, auto&& g) {
    return [f = FWD(f), g = FWD(g)](auto&&... x) {
        return g(f(FWD(x)...));
    };
}
```

A composition operator

```
auto operator|(auto&& f, auto&& g) {
    return [f = FWD(f), g = FWD(g)](auto&&... x) {
        return g(f(FWD(x)...));
    };
}

auto lookup_address = lookup_user | get_address;
```

A composition operator

```
auto operator|(auto&& f, auto&& g) {
    return [f = FWD(f), g = FWD(g)](auto&&... x) {
        return g(f(FWD(x)...));
    };
}

auto lookup_address = lookup_user | get_address;
```

Overload set? ADL?

A better composition operator

```
template <typename F>
struct fn : F {
    using F::F;

    friend auto operator|(auto&& f, auto&& g) {
        return fn([f = FWD(f), g = FWD(g)](auto&& x) {
            return g(f(FWD(x)));
        });
    }
};

#define FN(...) fn([](auto&&... x) -> decltype(__VA_ARGS__(FWD(x)...)) { \
    return __VA_ARGS__(FWD(x)...); \
})
```

A better composition operator

```
template <typename F>
struct fn : F {
    using F::F;

    friend auto operator|(auto&& f, auto&& g) {
        return fn([f = FWD(f), g = FWD(g)](auto&& x) {
            return g(f(FWD(x)));
        });
    }
};

#define FN(...) fn([](auto&&... x) -> decltype(__VA_ARGS__(FWD(x)...)) { \
    return __VA_ARGS__(FWD(x)...); \
})

auto lookup_address = FN(lookup_user) | FN(get_address);
```

What's the point?

```
unsubscribe_button.on_click([&]{
    auto user_id = get_current_user_id();
    auto user = lookup_user(user_id);
    auto address = get_address(user);
    subscription_list.unsubscribe(address);
});
```

Statement-oriented composition.

What's the point?

```
unsubscribe_button.on_click([&]{
    auto user_id = get_current_user_id();
    auto user = lookup_user(user_id);
    auto address = get_address(user);
    subscription_list.unsubscribe(address);
});
```

Statement-oriented composition.

```
unsubscribe_button.on_click(
    FN(get_current_user_id) | FN(lookup_user) | FN(get_address)
    | std::bind_front(
        &SubscriptionList::unsubscribe,
        std::ref(subscription_list)
    )
);
```

Using the composition operator.

Aside: Point-free style

"Pointed" style

```
lookup_address :: UserId -> Address
lookup_address id = get_address (lookup_user id)
```

Aside: Point-free style

"Pointed" style

```
lookup_address :: UserId -> Address
lookup_address id = get_address (lookup_user id)
```

"Point-free" style

```
lookup_address :: UserId -> Address
lookup_address = get_address . lookup_user
```

Thank you for listening!

What if the intermediate functions can fail?

```
std::optional<User> lookup_user(UserID id); // f
Address             get_address(User user); // g
```

What if the intermediate functions can fail?

```
std::optional<User> lookup_user(UserID id); // f
Address             get_address(User user); // g

auto lookup_address = FN(lookup_user) | FN(get_address);
auto address = lookup_address(id); // error
```

Composition shape mismatch

How do I compose `f(T) -> Box<U>` and `g(U) -> V`?

- `f` does not give a `U` but some “boxed” `Box<U>`
- `g` accepts a `U`

Composition shape mismatch

How do I compose `f(T) -> Box<U>` and `g(U) -> V`?

- `f` does not give a `U` but some "boxed" `Box<U>`
- `g` accepts a `U`

Need some way to compose data in a box!

Functors and their variants

Optional Value might not be there.

```
std::optional<User>          lookup_user(UserId id);
std::expected<User, Error> lookup_user(UserId id);
```

Boxes

Optional Value might not be there.

```
std::optional<User>          lookup_user(UserId id);
std::expected<User, Error> lookup_user(UserId id);
```

Range We have more than one value.

```
std::vector<User> lookup_user(UserId id);
```

Boxes

Optional Value might not be there.

```
std::optional<User>        lookup_user(UserId id);
std::expected<User, Error> lookup_user(UserId id);
```

Range We have more than one value.

```
std::vector<User> lookup_user(UserId id);
```

Sender We will have a value once the background work is done.

```
std::execution::task<User> lookup_user(UserId id);
```

Boxes

Optional Value might not be there.

```
std::optional<User>          lookup_user(UserId id);
std::expected<User, Error> lookup_user(UserId id);
```

Range We have more than one value.

```
std::vector<User> lookup_user(UserId id);
```

Sender We will have a value once the background work is done.

```
std::execution::task<User> lookup_user(UserId id);
```

All of the above We might have values once the background work is done.

```
std::execution::task<std::expected<std::vector<User>, Error>>
lookup_user(UserId id);
```

More esoteric boxes

Reader A box that will give you a value if you give it some context.

```
template <typename T, typename Context>
using Reader = std::function<T(Context)>;
```

More esoteric boxes

Reader A box that will give you a value if you give it some context.

```
template <typename T, typename Context>
using Reader = std::function<T(Context)>;
```

Writer A box that will give you a value and accumulate some log.

```
template <typename T, typename Log>
using Writer = std::function<std::pair<T, Log>()>;
```

Composing boxes

```
Box<User> lookup_user(UserId id); // f
Address   get_address(User user); // g
```

What is the return type of the composition?

```
??? lookup_address(UserId id); // h
```

Unwrapping boxes

```
T unwrap(Box<T> box);
```

Unwrapping boxes

```
T unwrap(Box<T> box);
```

We cannot get rid of the box!

Unwrapping boxes

```
T unwrap(Box<T> box);
```

We cannot get rid of the box!

Optional What if we don't have a value?

Unwrapping boxes

```
T unwrap(Box<T> box);
```

We cannot get rid of the box!

Optional What if we don't have a value?

Range What if we have more than one value?

Unwrapping boxes

```
T unwrap(Box<T> box);
```

We cannot get rid of the box!

Optional What if we don't have a value?

Range What if we have more than one value?

Sender We don't want to block waiting until we have a value.

Unwrapping boxes

```
T unwrap(Box<T> box);
```

We cannot get rid of the box!

Optional What if we don't have a value?

Range What if we have more than one value?

Sender We don't want to block waiting until we have a value.

Reader We don't have the context to get the value.

Unwrapping boxes

```
T unwrap(Box<T> box);
```

We cannot get rid of the box!

Optional What if we don't have a value?

Range What if we have more than one value?

Sender We don't want to block waiting until we have a value.

Reader We don't have the context to get the value.

Writer We don't want to discard the log.

Composing boxes

```
Box<User> lookup_user(UserId id); // f
Address   get_address(User user); // g
```

We have to keep the box!

```
Box<Address> lookup_address(UserId id); // h
```

Composing boxes

```
Box<User> lookup_user(UserId id); // f
Address   get_address(User user); // g
```

We have to keep the box!

```
Box<Address> lookup_address(UserId id); // h
```

How to implement it?

Composing optional

```
std::optional<Address> lookup_address(UserId id) {
    auto user = lookup_user(id);
    return user ? std::make_optional(get_address(*user)) : std::nullopt;
}
```

Composing optional

```
std::optional<Address> lookup_address(UserId id) {
    auto user = lookup_user(id);
    return user.transform(&get_address);
}
```

Composing ranges

```
std::vector<Address> lookup_address(UserId id) {
    std::vector<Address> addresses;
    for (const auto& user : lookup_user(id))
        addresses.push_back(get_address(user));
    return addresses;
}
```

Composing ranges

```
auto lookup_address(UserId id) {
    return lookup_user(id) | std::views::transform(&get_address);
}
```

Composing senders

```
std::execution::task<Address> lookup_address(UserId id) {
    auto user = co_await lookup_user(id);
    co_return get_address(user);
}
```

Composing senders

```
auto lookup_address(UserId id) {
    return lookup_user(id) | std::execution::then(&get_address);
}
```

Composing boxes

While we can't unwrap the box, we can call a function on the boxed value!

Composing boxes

While we can't unwrap the box, we can call a function on the boxed value!

Functor A type that can be mapped over

- A type constructor `Box<T>`
- A way to apply a function `T -> U` to a box `Box<T>` to get a box `Box<U>`

```
template <typename T, typename U>
Box<U> fmap(std::function<U(T)> fn, Box<T> box);
```

Composing boxes

While we can't unwrap the box, we can call a function on the boxed value!

Functor A type that can be mapped over

- A type constructor `Box<T>`
- A way to apply a function `T -> U` to a box `Box<T>` to get a box `Box<U>`

```
template <typename T, typename U>
Box<U> fmap(std::function<U(T)> fn, Box<T> box);
```

e.g.:

- `std::optional<T>` with `std::optional<T>::transform`
- `std::expected<T, E>` with `std::expected<T, E>::transform` (fixed `E`)
- A range of `T` with `std::views::transform`
- A sender of `T` with `std::execution::then`

Aside: Currying

```
fmap :: (T -> U) -> Box T -> Box U
```

Given a function `T -> U` and a boxed `Box T`, we get a boxed `Box U`.

Aside: Currying

```
fmap :: (T -> U) -> Box T -> Box U
```

Given a function `T -> U` and a boxed `Box T`, we get a boxed `Box U`.

```
fmap :: (T -> U) -> (Box T -> Box U)
```

Given a function `T -> U`, we "lift" it to a function `Box T -> Box U`.

Supporting functors in our composition operator

```
template <typename F>
struct fmap {
    F _fn;

    auto operator()(optional_like auto&& box) const {
        return FWD(box).transform(_fn);
    }
    auto operator()(std::ranges::range auto&& rng) const {
        return FWD(rng) | std::views::transform(_fn);
    }
    auto operator()(std::execution::sender auto&& snd) const {
        return FWD(snd) | std::execution::then(_fn);
    }
};
```

Supporting functors in our composition operator

```
template <typename F>
struct fmap {
    F _fn;

    auto operator()(optional_like auto&& box) const {
        return FWD(box).transform(_fn);
    }
    auto operator()(std::ranges::range auto&& rng) const {
        return FWD(rng) | std::views::transform(_fn);
    }
    auto operator()(std::execution::sender auto&& snd) const {
        return FWD(snd) | std::execution::then(_fn);
    }
};

auto lookup_address = lookup_user | fmap(FN(get_address));
```

N-ary composition of functions

```
Address lookup_address(UserId id);
Invoice lookup_invoice(UserId id);

Mail prepare_invoice(UserId id) {
    auto address = lookup_address(id);
    auto invoice = lookup_invoice(id);
    return Mail(address, invoice);
}
```

N-ary composition of boxes

```
Box<Address> lookup_address(UserId id);
Box<Invoice> lookup_invoice(UserId id);

Box<Mail> prepare_invoice(UserId id) {
    auto address = lookup_address(id);
    auto invoice = lookup_invoice(id);
    return ???;
}
```

Throwing fmap at the problem

```
Box<Address> lookup_address(UserId id);
Box<Invoice> lookup_invoice(UserId id);

Box<Mail> prepare_invoice(UserId id) {
    auto address = lookup_address(id);
    auto invoice = lookup_invoice(id);

    return fmap([&](Address address) {
        return fmap([&](Invoice invoice) {
            return Mail(address, invoice);
        })(invoice);
    })(address);
}
```

Throwing fmap at the problem

```
Box<Address> lookup_address(UserId id);
Box<Invoice> lookup_invoice(UserId id);

Box<Mail> prepare_invoice(UserId id) {
    auto address = lookup_address(id);
    auto invoice = lookup_invoice(id);

    return fmap([&](Address address) {
        return fmap([&](Invoice invoice) {
            return Mail(address, invoice);
        })(invoice);
    })(address);
}
```

Result is a `Box<Box<Mail>>`!

N-ary composition of boxes

We need a variadic version of `fmap`.

N-ary composition of boxes

We need a variadic version of `fmap`.

Applicative A functor that can apply a mapping to multiple boxes

- A type constructor `Box<T>`
- A way to apply a function `(T1, T2, ...) -> U` to multiple boxes `Box<T1>`, `Box<T2>`, ... to get a box `Box<U>`

```
template <typename U, typename ... Ts>
Box<U> fmap(std::function<U(Ts...)> fn, Box<Ts>... boxes);
```

N-ary composition of boxes

We need a variadic version of `fmap`.

Applicative A functor that can apply a mapping to multiple boxes

- A type constructor `Box<T>`
- A way to apply a function `(T1, T2, ...) -> U` to multiple boxes `Box<T1>`, `Box<T2>`, ... to get a box `Box<U>`

```
template <typename U, typename ... Ts>
Box<U> fmap(std::function<U(Ts...)> fn, Box<Ts>... boxes);
```

Alternatively: lift `(T1, T2, ...) -> U` to `(Box<T1>, Box<T2>, ...) -> Box<U>`.

N-ary composition of optionals

We can send mail if we have both an address and an invoice.

N-ary composition of optionals

We can send mail if we have both an address and an invoice.

```
template <typename F>
struct fmap {
    F _fn;

    auto operator()(optional_like auto&& ... box) const {
        if ((... && box.has_value())) {
            return std::make_optional(_fn(FWD(box).value()...));
        } else {
            return std::nullopt;
        }
    }
};
```

N-ary composition of senders

As soon as we have both the address and the invoice, we can prepare the mail.

N-ary composition of senders

As soon as we have both the address and the invoice, we can prepare the mail.

```
template <typename F>
struct fmap {
    F _fn;

    auto operator()(std::execution::sender auto&& ... snd) const {
        return std::execution::when_all(FWD(snd)...)
          | std::execution::then([&](auto&& ... t) {
              return _fn(FWD(t)...);
          });
    }
};
```

N-ary composition of ranges

Each address of the user gets each invoice.

N-ary composition of ranges

Each address of the user gets each invoice.

```
template <typename F>
struct fmap {
    F _fn;

    auto operator()(std::ranges::range auto&& ... rng) const {
        return std::views::cartesian_product(FWD(rng)...)
          | std::views::transform([&](auto&& tpl) {
              return std::apply(_fn, FWD(tpl));
          });
    }
};
```

N-ary composition of ranges in a different way

The `i`th address of the user gets the `i`th invoice.

N-ary composition of ranges in a different way

The ith address of the user gets the ith invoice.

```
template <typename F>
struct fmap {
    F _fn;

    auto operator()(std::ranges::range auto&& ... rng) const {
        return std::views::zip(FWD(rng)...)
          | std::views::transform([&](auto&& tpl) {
              return std::apply(_fn, FWD(tpl));
          });
    }
};
```

N-ary composition of ranges in a different way

The ith address of the user gets the ith invoice.

```
template <typename F>
struct fmap {
    F _fn;

    auto operator()(std::ranges::range auto&& ... rng) const {
        return std::views::zip_transform(_fn, FWD(rng)...);



    }
};
```

Generic when_all

```
template <typename U, typename ... Ts>
Box<U> fmap(std::function<U(Ts...)> fn, Box<Ts>... boxes);

template <typename ... Ts>
Box<std::tuple<Ts...>> when_all(Box<Ts>... boxes) {
    return fmap(FN(std::make_tuple), FWD(boxes)...);
}
```

Nullary composition of boxes

```
template <typename U>
Box<U> fmap(std::function<U()> fn);
```

Nullary composition of boxes

```
template <typename U>
Box<U> fmap(std::function<U()> fn);
```

A way to create a box containing a given value!

```
template <template <typename> typename Box>
auto pure(auto&& value) {
    return fmap_for<Box>([&] { return FWD(value); });
}
```

Aside: Applicative in Haskell

Applicative: The Forgotten Functional Pattern in C++ - Ben Deane - CppNow 2023

Can all functors be made applicative?

Can all functors be made applicative?

`std::map<K, T>` is a functor over `T`.

```
template <typename F>
struct fmap {
    F _fn;

    template <typename K, typename T>
    auto operator()(const std::map<K, T>& map) const {
        std::map<K, std::invoke_result_t<F, T>> result;
        for (const auto& [key, value] : map) {
            result.emplace(key, _fn(value));
        return result;
    }
};
```

Can all functors be made applicative?

`std::map<K, T>` is a functor over `T`.

```
template <typename F>
struct fmap {
    F _fn;

    template <typename K, typename T>
    auto operator()(const std::map<K, T>& map) const {
        std::map<K, std::invoke_result_t<F, T>> result;
        for (const auto& [key, value] : map) {
            result.emplace(key, _fn(value));
        return result;
    }
};
```

Cannot be reasonably extended to multiple maps with different keys.

Can all functors be made applicative?

`std::map<K, T>` is a functor over `T`.

```
template <typename F>
struct fmap {
    F _fn;

    template <typename K, typename T>
    auto operator()(const std::map<K, T>& map) const {
        std::map<K, std::invoke_result_t<F, T>> result;
        for (const auto& [key, value] : map) {
            result.emplace(key, _fn(value));
        return result;
    }
};
```

Cannot be reasonably extended to multiple maps with different keys.

What would the keys of the map returned from `pure(11)` be?!

Combining multiple values into one

```
Address lookup_invoice_address(UserId id);
Address lookup_contact_address(UserId id);
Address lookup_recovery_address(UserId id);
```

Combining multiple values into one

```
Address lookup_invoice_address(UserId id);
Address lookup_contact_address(UserId id);
Address lookup_recovery_address(UserId id);

Address lookup_some_address(UserId id) {
    // Chosen by fair dice roll.
    return lookup_contact_address(id);
}
```

Combining multiple boxes into one

```
Box<Address> lookup_invoice_address(UserId id);
Box<Address> lookup_contact_address(UserId id);
Box<Address> lookup_recovery_address(UserId id);

Box<Address> lookup_some_address(UserId id) {
    ???
}
```

Throwing fmap at the problem

```
Box<Address> lookup_invoice_address(UserId id);
Box<Address> lookup_contact_address(UserId id);
Box<Address> lookup_recovery_address(UserId id);

Box<Address> lookup_some_address(UserId id) {
    return fmap([](auto... addresses) {
        return addresses...[0];
    })(
        lookup_invoice_address(id),
        lookup_contact_address(id),
        lookup_recovery_address(id)
    );
}
```

Throwing fmap at the problem

```
Box<Address> lookup_invoice_address(UserId id);
Box<Address> lookup_contact_address(UserId id);
Box<Address> lookup_recovery_address(UserId id);

Box<Address> lookup_some_address(UserId id) {
    return fmap([](auto... addresses) {
        return addresses...[0];
    })(
        lookup_invoice_address(id),
        lookup_contact_address(id),
        lookup_recovery_address(id)
    );
}
```

Requires all to exist / waits for all of them.

Combining boxes

We need a way to combine multiple boxes into one box.

Combining boxes

We need a way to combine multiple boxes into one box.

Alternative A functor where we can combine multiple boxes into one box

```
template <typename ... T>
Box<U> falt(Box<T>... boxes);
```

Combining optionals

We pick the first address that exists.

Combining optionals

We pick the first address that exists.

```
auto falt(optional_like auto&&... box) {
    for (auto&& b : {box...}) {
        if (b.has_value()) {
            return FWD(b);
        }
    }
    return std::nullopt;
}
```

Combining senders

We pick the address whose lookup finishes first.

Combining senders

We pick the address whose lookup finishes first.

```
auto falt(std::execution::sender auto&&... snd) {
    return std::execution::when_any(FWD(snd)...);
}
```

(Not yet in C++26)

Combining ranges

We want all addresses of any source.

Combining ranges

We want all addresses of any source.

```
auto falt(std::ranges::range auto&&... rng) {
    return std::views::concat(FWD(rng)...);
}
```

Combining ranges

We want all addresses of any source.

```
auto falt(std::ranges::range auto&&... rng) {
    return std::views::concat(FWD(rng)...);
}
```

Alternatively: We want all addresses of the first source that has any.

Combining zero boxes

```
template <template <typename> Box>
auto falt();
```

Combining zero boxes

```
template <template <typename> Box>
auto falt();
```

The “empty” box that is the identity for `falt`.

Optional `std::nullopt`

Range `std::views::empty` (of some type?)

Sender A sender that never finishes.

Aside: Alternative in Haskell

Haskell doesn't have variadics:

```
empty :: Box a
(<|>) :: Box a -> Box a -> Box a
```

where

```
empty <|> x = x
x <|> empty = x
(x <|> y) <|> z = x <|> (y <|> z)
```

Can all functors be made alternative?

Can all functors be made alternative?

```
template <typename T>
class NonEmptyVec;
```

`NonEmptyVec<T>` is a functor over `T`.

Can all functors be made alternative?

```
template <typename T>
class NonEmptyVec;
```

`NonEmptyVec<T>` is a functor over `T`.

You cannot provide `empty()`.

Applicative vs. alternative

Applicative

- Get all of the values.
- &&
- `std::tuple`
- `pure(t)` to get a box containing a value.

Alternative

- Get one of the values.
- `||`
- `std::variant`
- `empty()` to get the empty box.

Composing boxes with box functions

```
Box<User>    lookup_user(UserId id);
Box<Address> get_address(User user);

Box<Address> lookup_address(UserId id) {
    return ???;
}
```

Throwing fmap at the problem

```
Box<User>    lookup_user(UserId id);
Box<Address> get_address(User user);

Box<Address> lookup_address(UserId id) {
    return fmap(get_address)(lookup_user(id));
}
```

Throwing fmap at the problem

```
Box<User>    lookup_user(UserId id);
Box<Address> get_address(User user);

Box<Address> lookup_address(UserId id) {
    return fmap(get_address)(lookup_user(id));
}
```

Result is a `Box<Box<Address>>`!

Composing boxes with box functions

We need a way to run box-returning functions on boxes.

Composing boxes with box functions

We need a way to run box-returning functions on boxes.

Monad A functor that can apply a mapping returning a boxed value

```
template <typename T, typename U>
Box<U> bind(std::function<Box<U>(T)> fn, Box<T> box);
```

Composing boxes with box functions

We need a way to run box-returning functions on boxes.

Monad A functor that can apply a mapping returning a boxed value

```
template <typename T, typename U>
Box<U> bind(std::function<Box<U>(T)> fn, Box<T> box);
```

Alternatively: lift `T -> Box<U>` to `Box<T> -> Box<U>`.

Composing optionals with optional functions

```
std::optional<Address> lookup_address(UserId id) {
    auto user = lookup_user(id);
    return user ? get_address(*user) : std::nullopt;
}
```

Composing optionals with optional functions

```
std::optional<Address> lookup_address(UserId id) {
    auto user = lookup_user(id);
    return user.and_then(&get_address);
}
```

Composing ranges with range functions

```
std::vector<Address> lookup_address(UserId id) {
    std::vector<Address> addresses;
    for (auto const& user : lookup_user(id))
      for (auto const& address : get_address(user))
        addresses.push_back(address);
    return addresses;
}
```

Composing ranges with range functions

```
auto lookup_address(UserId id) {
    return lookup_user(id)
      | std::views::transform(&get_address) | std::views::join;
}
```

Composing senders with sender functions

```
std::execution::task<Address> lookup_address(UserId id) {
    auto user = co_await lookup_user(id);
    co_return co_await get_address(user);
}
```

Composing senders with sender functions

```
auto lookup_address(UserId id) {
    return lookup_user(id) | std::execution::let_value(&get_address);
}
```

Composing boxes with box functions

```
template <typename F>
struct bind {
    F _fn;

    auto operator()(optional_like auto&& box) const {
        return FWD(box).and_then(_fn);
    }
    auto operator()(std::ranges::range auto&& rng) const {
        return FWD(rng) | std::views::transform(_fn) | std::views::join;
    }
    auto operator()(std::execution::sender auto&& snd) const {
        return FWD(snd) | std::execution::let_value(_fn);
    }
};
```

Composing boxes with box functions

```
template <typename F>
struct bind {
    F _fn;

    auto operator()(optional_like auto&& box) const {
        return FWD(box).and_then(_fn);
    }
    auto operator()(std::ranges::range auto&& rng) const {
        return FWD(rng) | std::views::transform(_fn) | std::views::join;
    }
    auto operator()(std::execution::sender auto&& snd) const {
        return FWD(snd) | std::execution::let_value(_fn);
    }
};

auto lookup_address = lookup_user | bind(FN(get_address));
```

Monads enable choices based on the previous results

```
fn | fmap([&](int x) {
    return x + 1;
})
```

vs.

```
fn | bind([&](auto x) {
    if (x == 0)
        return std::nullopt;
    else
        return std::make_optional(x + 1);
})
```

Monads enable choices based on the previous results

```
fn | fmap([&](int x) {
    return x + 1;
})
```

vs.

```
fn | bind([&](auto x) {
    if (x == 0)
        return std::nullopt;
    else
        return std::make_optional(x + 1);
})
```

Optional Return a value or nothing.

Range Return zero or more values.

Sender Send a value, an error, or stop; wait for a nested computation.

Monads allow flattening

```
auto operator()(std::ranges::range auto&& rng) const {
    return FWD(rng) | std::views::transform(_fn) | std::views::join;
}
```

Monads allow flattening

```
auto operator()(std::ranges::range auto&& rng) const {
    return FWD(rng) | std::views::transform(_fn) | std::views::join;
}

template <template <typename> typename Box, typename T>
Box<T> join(Box<Box<T>> box) {
    return bind([](Box<T> inner) { return inner; })(box);
}
```

Monads allow n-ary composition

```
Box<Mail> prepare_invoice(UserId id) {
    auto address = lookup_address(id);
    auto invoice = lookup_invoice(id);

    return fmap([&](Address address) {
        return fmap([&](Invoice invoice) {
            return Mail(address, invoice);
        })(invoice);
    })(address);
}
```

Result is a `Box<Box<Mail>>`!

Monads allow n-ary composition

```
Box<Mail> prepare_invoice(UserId id) {
    auto address = lookup_address(id);
    auto invoice = lookup_invoice(id);

    return bind([&](Address address) {
        return fmap([&](Invoice invoice) {
            return Mail(address, invoice);
        })(invoice);
    })(address);
}
```

Result is a `Box<Mail>`.

Every monad is an applicative

```
template <typename U, typename T1, typename ... TN>
Box<U> fmap(std::function<U(Ts...)> fn, Box<T1> box1, Box<TN>... boxn) {
    return bind([&](T1 t1) {
      return fmap(std::bind_front(fn, t1), boxn...);
    })(box1);
}
```

when_all in terms of let_value?

```
auto when_all(std::execution::sender auto h, std::execution::sender auto... t)
{
    if constexpr (sizeof...(t) == 0) {
        return h;
    } else {
        return h | std::execution::let_value([&](auto&& result) {
            return when_all(FWD(t)...)
              | std::execution::then([&](auto&&... results) {
                return std::tuple_cat(
                    std::make_tuple(result),
                    std::forward_as_tuple(results...));
              });
        });
    }
}
```

when_all in terms of let_value?

```
auto when_all(std::execution::sender auto h, std::execution::sender auto... t)
{
    if constexpr (sizeof...(t) == 0) {
        return h;
    } else {
        return h | std::execution::let_value([&](auto&& result) {
            return when_all(FWD(t)...)
              | std::execution::then([&](auto&&... results) {
                return std::tuple_cat(
                    std::make_tuple(result),
                    std::forward_as_tuple(results...));
              });
        });
    }
}
```

Executes all senders in sequence!

Is every monad an applicative?

Zip implementation of applicative `fmap` for ranges:

```
template <typename F>
struct fmap {
    F _fn;

    auto operator()(std::ranges::range auto&& ... rng) const {
        return std::views::zip_transform(_fn, FWD(rng)...);
    }
};
```

Is every monad an applicative?

Zip implementation of applicative `fmap` for ranges:

```
template <typename F>
struct fmap {
    F _fn;

    auto operator()(std::ranges::range auto&& ... rng) const {
        return std::views::zip_transform(_fn, FWD(rng)...);
    }
};
```

In `bind`, each `_fn(x)` can return a range with a different number of elements!

Is every monad an applicative?

Zip implementation of applicative `fmap` for ranges:

```
template <typename F>
struct fmap {
    F _fn;

    auto operator()(std::ranges::range auto&& ... rng) const {
        return std::views::zip_transform(_fn, FWD(rng)...);
    }
};
```

In `bind`, each `_fn(x)` can return a range with a different number of elements!

You cannot implement a `bind` that would generate the variadic `fmap`.

Recap

Functor A box whose contents can be mapped over

Applicative A functor where we can apply a mapping to multiple boxes

Alternative A functor where we can combine multiple boxes into one box

Monad A functor that can apply a mapping returning a boxed value

Contravariant functors

Variance

```
class User {};
class PremiumUser : public User {};

template <typename T>
using C = ...;
```

`PremiumUser` is a subtype of `User`, but what about `C<PremiumUser>` and `C<User>`?

Variance

Invariant `C<PremiumUser>` and `C<User>` are unrelated.

```
template <typename T>
using C = std::vector<T>; // invariant
```

Variance

Covariant `C<PremiumUser>` is a subtype of `C<User>`.

```
template <typename T>
using C = std::function<T*()>; // covariant
```

Variance

Covariant `C<PremiumUser>` is a subtype of `C<User>`.

```
template <typename T>
using C = std::function<T*()>; // covariant

void process_user(C<User> user_factory);
void process_premium_user(C<PremiumUser> premium_user_factory);
```

Variance

Contravariant `C<User>` is a subtype of `C<PremiumUser>`.

```
template <typename T>
using C = std::function<void(T*)>; // contravariant
```

Variance

Contravariant `C<User>` is a subtype of `C<PremiumUser>`.

```
template <typename T>
using C = std::function<void(T*)>; // contravariant

void for_each_premium_user(C<PremiumUser> premium_user_consumer);
void for_each_user(C<User> user_consumer);
```

Variance in functors

(Covariant) functors

- A type constructor `Box<T>`
- A way to apply a function `T -> U` to a box `Box<T>` to get a box `Box<U>`

```
template <typename T, typename U>
Box<U> fmap(std::function<U(T)> fn, Box<T> box);
```

Variance in functors

(Covariant) functors

- A type constructor `Box<T>`
- A way to apply a function `T -> U` to a box `Box<T>` to get a box `Box<U>`

```
template <typename T, typename U>
Box<U> fmap(std::function<U(T)> fn, Box<T> box);
```

Contravariant functors

- A type constructor `ContraBox<T>`
- A way to apply a function `U -> T` to a box `ContraBox<T>` to get a box `ContraBox<U>`

```
template <typename T, typename U>
ContraBox<U> contramap(std::function<T(U)> fn, ContraBox<T> box);
```

Producers vs. consumers

(Covariant) functors Producers of `T`

- `fmap`: given a function `T -> U` lift it to a function `Box<T> -> Box<U>`
- e.g. ranges

Producers vs. consumers

(Covariant) functors Producers of `T`

- `fmap`: given a function `T -> U` lift it to a function `Box<T> -> Box<U>`
- e.g. ranges

Contravariant functors Consumers of `T`

- `contramap`: given a function `U -> T` lift it to a function `ContraBox<T> -> ContraBox<U>`
- e.g. predicates

Composing predicates

```
bool is_overdue(Invoice invoice);
Invoice lookup_invoice(UserId id);

bool has_overdue_invoice(UserId id) {
    return is_overdue(lookup_invoice(id)); // "contramap"
}
```

Composing predicate functions

```
template <typename T>
using Predicate = std::function<bool(T)>;

template <typename F>
struct contramap {
    F _fn;

    template <typename T>
    auto operator()(Predicate<T> pred) const -> Predicate<param_of<F>> {
        return [=](auto const& arg) {
            return _fn(arg) | pred;
        };
    }
};
```

Composing predicate functions

```
Predicate<Invoice> invoice_will_be_overdue_by(Date date);
Invoice lookup_invoice(UserId id);
```

Composing predicate functions

```
Predicate<Invoice> invoice_will_be_overdue_by(Date date);
Invoice lookup_invoice(UserId id);

Predicate<UserId> user_has_overdue_invoice_by(Date date) {
    return contramap(lookup_invoice)(invoice_will_be_overdue_by(date));
}
```

Composing predicate functions

```
Predicate<Invoice> invoice_will_be_overdue_by(Date date);
Invoice lookup_invoice(UserId id);

Predicate<UserId> user_has_overdue_invoice_by(Date date) {
    return invoice_will_be_overdue_by(date) | contramap(lookup_invoice);
}
```

Predicate functions in the STL

```
std::ranges::find_if(invoices, invoice_will_be_overdue_by(date));
```

Predicate functions in the STL

```
std::ranges::find_if(invoices, invoice_will_be_overdue_by(date));

std::ranges::find_if(users,
    invoice_will_be_overdue_by(date) | contramap(lookup_invoice)
);
```

Predicate functions in the STL

```
std::ranges::find_if(invoices, invoice_will_be_overdue_by(date));

std::ranges::find_if(users,
    invoice_will_be_overdue_by(date) | contramap(lookup_invoice)
);

std::ranges::find_if(users, invoice_will_be_overdue_by(date), lookup_invoice);
```

Projection functions are lifted to predicates via contramap.

Composing serializers

```
template <typename T>
using Serializer = std::function<void(T)>;
```

Composing serializers

```
template <typename T>
using Serializer = std::function<void(T)>;

Serializer<Invoice> serialize_invoice(File file);
Invoice lookup_invoice(UserId id);

Serialize<UserId> serialize_users_invoice(File file) {
    return contramap(lookup_invoice)(serialize_invoice(file));
}
```

Contravariant applicative

(Covariant) Applicative

```
template <typename U, typename ... Ts>
Box<U> fmap(std::function<U(Ts...)> fn, Box<Ts>... boxes);
```

Contravariant applicative

(Covariant) Applicative

```
template <typename U, typename ... Ts>
Box<U> fmap(std::function<U(Ts...)> fn, Box<Ts>... boxes);
```

Contravariant applicative ("divisible")

```
template <typename U, typename ... Ts>
ContraBox<U> contramap(std::function<std::tuple<Ts..>(U)> fn,
ContraBox<Ts>... boxes);
```

Dividing responsibilities between predicate functions

```
Predicate<Invoice> invoice_due_date_is_before(Date date);
Predicate<Payment> payment_is(PaymentStatus status);

Predicate<Invoice> invoice_will_be_overdue_by(Date date) {
    return contramap([](Invoice invoice) {
        return std::tie(invoice.due_date(), invoice.payment_status());
    })(
        invoice_due_date_is_before(date),
        payment_is(PaymentStatus::Unpaid)
    );
}
```

Dividing responsibilities between predicate functions

```
template <typename F>
struct contramap {
    F _fn;

    template <typename ... T>
    auto operator()(Predicate<T> ... pred) const -> Predicate<param_of<F>> {
        return [=](auto const& arg) {
            auto const& [fields...] = _fn(arg);
            return (pred(fields) && ...);
        };
    }
};
```

Dividing responsibilities between serializers

```
template <typename F>
struct contramap {
    F _fn;

    template <typename ... T>
    auto operator()(Serializer<T> ... s) const -> Serializer<param_of<F>> {
        return [=](auto const& arg) {
            auto const& [fields...] = _fn(arg);
            return (s(fields), ...);
        };
    }
};
```

Now we can serialize a tuple!

Dividing responsibilities between zero divisibles

```
template <typename U>
ContraBox<U> contramap(std::function<void(U)> fn);
```

Dividing responsibilities between zero divisibles

```
template <typename U>
ContraBox<U> contramap(std::function<void(U)> fn);
```

A way to create a contra box that consumes a `U` without “any effect”.

```
auto conquer() { // "divide" and "conquer"
    return [](auto const&) { return true; };
}
```

Aside: Haskell documentation

```
class Contravariant f => Divisible f
```

A Divisible contravariant functor is the contravariant analogue of Applicative.

Aside: Haskell documentation

```
class Contravariant f => Divisible f
```

A Divisible contravariant functor is the contravariant analogue of Applicative.

In denser jargon, a Divisible contravariant functor is a monoid object in the category of presheaves from Hask to Hask, equipped with Day convolution mapping the Cartesian product of the source to the Cartesian product of the target.

Aside: Haskell documentation

```
class Contravariant f => Divisible f
```

A Divisible contravariant functor is the contravariant analogue of Applicative.

In denser jargon, a Divisible contravariant functor is a monoid object in the category of presheaves from Hask to Hask, equipped with Day convolution mapping the Cartesian product of the source to the Cartesian product of the target.

By way of contrast, an Applicative functor can be viewed as a monoid object in the category of copresheaves from Hask to Hask, equipped with Day convolution mapping the Cartesian product of the source to the Cartesian product of the target.

Contravariant alternative

(Covariant) Alternative

```
template <typename ... T>
Box<U> falt(Box<T>... boxes);
```

Contravariant alternative

(Covariant) Alternative

```
template <typename ... T>
Box<U> falt(Box<T>... boxes);
```

Contravariant alternative (“decidable”)

```
template <typename ... T>
ContraBox<U> choose(std::function<std::variant<T...>(U)> d,
ContraBox<T>... boxes);
```

Deciding between predicate functions

```
Predicate<RegularUser> free_trial_on(Date date);

Predicate<User> has_premium_access_on(Date date) {
    return choose([](User const& user) {
        if (auto* regular = dynamic_cast<RegularUser const*>(&user)) {
            return std::variant<RegularUser, PremiumUser>{*regular};
        } else if (auto* premium = dynamic_cast<PremiumUser const*>(&user)) {
            return std::variant<RegularUser, PremiumUser>{*premium};
        } else {
            throw std::runtime_error("Unknown user type");
        }
    })(
        free_trial_on(date),
        [](PremiumUser const&) { return true; }
    );
}
```

Deciding between decidables

For contraboxes that are functions taking a `T`:

```
auto choose(auto d, auto ... f) {
    return [=](auto const& arg) {
        std::visit(Overload(f...), d(arg));
    });
}
```

Deciding between decidables

For contraboxes that are functions taking a `T`:

```
auto choose(auto d, auto ... f) {
    return [=](auto const& arg) {
        std::visit(Overload(f...), d(arg));
    });
}

auto choose(auto d) { // lose in Haskell
    return [=](auto const& arg) {
        d(arg);
        std::unreachable();
    });
}
```

Divisible vs. decidable

Divisible

- Consume all of the sub-values.
- `,`
- `std::tuple`
- `conquer()` to consume without effect.

Decidable

- Consume one of the sub-values.
- `std::visit`
- `std::variant`
- `lose()` to indicate that no value exists.

Recap

Contravariant functor Consumers of `T` that can be pre-composed with a mapping

Divisible - contravariant applicative Consumers of `T` created by dividing it into parts

Decidable - contravariant alternative Consumers of `T` created by choosing between multiple consumers

Contravariant monads?

(Covariant) Monad

```
template <typename T, typename U>
Box<U> bind(std::function<Box<U>(T)> fn, Box<T> box);

template <typename T>
Box<T> join(Box<Box<T>> box);
```

Contravariant monads?

(Covariant) Monad

```
template <typename T, typename U>
Box<U> bind(std::function<Box<U>(T)> fn, Box<T> box);

template <typename T>
Box<T> join(Box<Box<T>> box);
```

We have no `std::function<U(T)>` we can flip to `std::function<T(U)>`!

Contravariant monads?

(Covariant) Monad

```
template <typename T, typename U>
Box<U> bind(std::function<Box<U>(T)> fn, Box<T> box);

template <typename T>
Box<T> join(Box<Box<T>> box);
```

We have no `std::function<U(T)>` we can flip to `std::function<T(U)>`!

Comonad

```
template <typename T, typename U>
Box<U> extend(std::function<U(Box<T>)> fn, Box<T> box);

template <typename T>
Box<Box<T>> duplicate(Box<T> box);
```

Contravariant monads?

"A Monad is just a monoid in the category of endofunctors."

Contravariant monads?

"A Monad is just a monoid in the category of endofunctors."

Functor transformation from one category to another.
Endofunctor transformation from a category to itself.

Contravariant monads?

"A Monad is just a monoid in the category of endofunctors."

Functor transformation from one category to another.
Endofunctor transformation from a category to itself.

Flipping the arrows of an endofunctor doesn't change anything.

Foldables and traversables

Unwrapping boxes, revisited

So far:

- `Box<T> -> Box<U>` from `T -> U` via `fmap`
- `Box<T> -> Box<U>` from `T -> Box<U>` via `bind`
- `T -> Box<T>` via `pure`

Unwrapping boxes, revisited

So far:

- `Box<T> -> Box<U>` from `T -> U` via `fmap`
- `Box<T> -> Box<U>` from `T -> Box<U>` via `bind`
- `T -> Box<T>` via `pure`

No `Box<T> -> T` in general.

Unwrapping boxes, revisited

So far:

- `Box<T> -> Box<U>` from `T -> U` via `fmap`
- `Box<T> -> Box<U>` from `T -> Box<U>` via `bind`
- `T -> Box<T>` via `pure`

No `Box<T> -> T` in general.

But what if we have a `Box` that actually is a “normal” data structure?

Summarizing a box

Consider `std::vector<T>`:

```
T foo(std::vector<T> const& ts);
```

Summarizing a box

Consider `std::vector<T>`:

```
T foo(std::vector<T> const& ts);
```

- `std::ranges::front`?
- `std::ranges::back`?

Summarizing a box

```
T sum(std::vector<T> const& ts) {
    T result = 0;
    for (auto const& t : ts) {
        result += t;
    }
    return result;
}
```

Summarizing a box

```
T sum(std::vector<T> const& ts) {
    T result = 0;
    for (auto const& t : ts) {
        result += t;
    }
    return result;
}

T max(std::vector<T> const& ts) {
    T result = T::min();
    for (auto const& t : ts) {
        result = std::max(result, t);
    }
    return result;
}
```

Requirements for summarizing

- An initial value (e.g. `0` for sum, `T::min()` for max)
- A way to combine the initial value with each element (e.g. `operator+=` for sum, `std::max` for max)

Requirements for summarizing

- An initial value (e.g. `0` for sum, `T::min()` for max)
- A way to combine the initial value with each element (e.g. `operator+=` for sum, `std::max` for max)

Monoid A type `M` with

- a binary operation to combine two values:

```
M op(M const& lhs, M const& rhs);
```

- an identity element `id` with:

```
op(id, m) == m
op(m, id) == m
```

Summarizing a box

Foldable A box that can be summarized with a monoid:

```
template <typename T, typename M>
M fold_map(Box<T> box, std::function<M(T)> fn);
```

Summarizing a box

Foldable A box that can be summarized with a monoid:

```
template <typename T, typename M>
M fold_map(Box<T> box, std::function<M(T)> fn);
```

Alternatively:

```
template <typename T, typename M>
M fold(Box<T> box, std::function<void(M&, T)> op, M id);
```

Summarizing a box

```
template <typename T>
T sum(Box<T> const& ts) {
    return fold(box, [](T& acc, T t) { acc += t; }, 0);
}
```

Summarizing a box

```
template <typename T>
T sum(Box<T> const& ts) {
    return fold(box, [](T& acc, T t) { acc += t; }, 0);
}

template <typename T> requires Monoid<T>
T max(Box<T> const& ts) {
    return fold(box, [](T& acc, T t) { acc = std::max(acc, t); }, T::min());
}
```

Summarizing a box

```
template <typename T>
std::vector<T> to_vector(Box<T> const& ts) {
    return fold(ts, [](std::vector<T>& acc, T t) { acc.push_back(t); }, {});
}
```

Summarizing a box

```
template <typename T>
std::vector<T> to_vector(Box<T> const& ts) {
    return fold(ts, [](std::vector<T>& acc, T t) { acc.push_back(t); }, {});
}

template <typename T>
std::size_t length(Box<T> const& ts) {
    return fold(ts, [](std::size_t& acc, T) { ++acc; }, 0);
}
```

Summarizing a box

```
Money total_revenue(Box<Invoice> const& invoices) {
    return fold(invoices,
      [](Money& acc, Invoice invoice) { acc += invoice.amount(); },
      Money{0}
    );
}
```

Summarizing a box

If the box is also a functor:

```
Money total_revenue(Box<Invoice> const& invoices) {
    return sum(invoices | fmap(get_amount));
}
```

Not all foldables are functors

```
template <typename T, typename M>
M fold(std::set<T> box, std::function<void(M&, T)> op, M id) {
    M result = id;
    for (auto const& t : box) {
        op(result, t);
    }
    return result;
}
```

Not all foldables are functors

```
template <typename T, typename M>
M fold(std::set<T> box, std::function<void(M&, T)> op, M id) {
    M result = id;
    for (auto const& t : box) {
        op(result, t);
    }
    return result;
}
```

But how would you implement `fmap([](int i) { return std::complex<double>(i, 0); }, std::set<int>{1, 2, 3})`?

Not all functors are foldable

Foldable functor:

- Optional
- Range

Non-foldable functor:

- Sender (would require blocking to give you the plain `T`)

Aside: Applicative and alternative are monoids

Applicative: Monoidal combination of effects

```
auto id = pure(std::tuple<>{});

auto op(auto const& lhs, auto const& rhs) {
    return fmap(FN(std::tuple_cat))(lhs, rhs);
}
```

Aside: Applicative and alternative are monoids

Applicative: Monoidal combination of effects

```
auto id = pure(std::tuple<>{});

auto op(auto const& lhs, auto const& rhs) {
    return fmap(FN(std::tuple_cat))(lhs, rhs);
}
```

Alternative: Monoidal combination of structure

```
auto id = empty;

auto op(auto const& lhs, auto const& rhs) {
    return falt(lhs, rhs);
}
```

Validating boxes

```
bool is_valid_user(User user);
```

Validating boxes

```
bool is_valid_user(User user);

bool valid_users(Box<User> const& users) {
    return fold(users,
      [](bool& acc, User user) { acc = acc && is_valid_user(user); },
      true
    );
}
```

(We probably want a short-circuit optimization)

Validating boxes

But what if we want to keep the users?

Validating boxes

But what if we want to keep the users?

```
std::optional<User> validate_user(User user) {
    if (is_valid_user(user)) {
        return user;
    } else {
        return std::nullopt;
    }
}
```

Validating boxes

But what if we want to keep the users?

```
std::optional<User> validate_user(User user) {
    if (is_valid_user(user)) {
        return user;
    } else {
        return std::nullopt;
    }
}

std::optional<Box<User>> validate(Box<User> const& users) {
    ???
}
```

Validating boxes

We cannot use `fold`, because we lose the original box!

```
std::optional<std::vector<User>> validate(Box<User> const& users) {
    return fold(users,
      [](std::optional<std::vector<User>>& acc, User user) {
          if (acc && is_valid_user(user)) {
              acc->push_back(user);
          } else {
              acc = std::nullopt;
          }
      },
      std::vector<User>{}
    );
}
```

Validating boxes

We cannot use `fmap`, because we get a `Box<std::optional<User>>` instead of a `std::optional<Box<User>>`!

```
Box<std::optional<User>> validate(Box<User> const& users) {
    return users | fmap(validate_user);
}
```

Validating boxes

We need a way to "flip" the structure of `Box<std::optional<T>>` to `std::optional<Box<T>>`!

```
template <typename T>
std::optional<std::vector<T>> sequence(std::vector<std::optional<T>> boxes) {
    std::optional<std::vector<T>> result = std::vector<T>{};
    for (auto const& box : boxes) {
        if (result && box) {
            result->push_back(*box);
        } else {
            result = std::nullopt;
        }
    }
    return result;
}
```

Validating boxes

We need a way to "flip" the structure of `Box<std::optional<T>>` to `std::optional<Box<T>>`!

```
template <typename T>
std::optional<std::vector<T>> sequence(std::vector<std::optional<T>> boxes) {
    std::optional<std::vector<T>> result
      = pure<std::optional>(std::vector<T>{});
    for (auto const& box : boxes) {
        result = fmap(push_back)(result, box);
    }
    return result;
}
```

Validating boxes

We need a way to "flip" the structure of `Box<std::optional<T>>` to `std::optional<Box<T>>`!

```
template <template <typename> typename Applicative, typename T>
Applicative<std::vector<T>> sequence(std::vector<Applicative<T>> boxes) {
    std::optional<std::vector<T>> result
      = pure<Applicative>(std::vector<T>{});
    for (auto const& box : boxes) {
        result = fmap(push_back)(result, box);
    }
    return result;
}
```

Generalizing n-ary composition of boxes

We essentially have `fmap` for `std::vector` of boxes!

Generalizing n-ary composition of boxes

We essentially have `fmap` for `std::vector` of boxes!

```
template <typename ... Ts>
auto when_all(std::execution::task<Ts> ... tasks)
  -> std::execution::task<std::tuple<Ts...>>;
```

Generalizing n-ary composition of boxes

We essentially have `fmap` for `std::vector` of boxes!

```
template <typename ... Ts>
auto when_all(std::execution::task<Ts> ... tasks)
  -> std::execution::task<std::tuple<Ts...>>;

template <typename T>
auto when_all(std::vector<std::execution::task<T>> tasks)
  -> std::execution::task<std::vector<T>> {
    return sequence(tasks);
}
```

Validating boxes

```
std::optional<Box<User>> validate(Box<User> const& users) {
    return users | fmap(validate_user) | sequence;
}
```

Validating boxes

```
std::optional<Box<User>> validate(Box<User> const& users) {
    return traverse(users, validate_user);
}
```

Traversable boxes

Traversable A box that can be traversed with an applicative:

```
template <typename T, typename U,
    template <typename> typename Applicative>
Applicative<Box<U>> traverse(Box<T> box,
  std::function<Applicative<U>(T)> fn);
```

Traversable boxes

Traversable A box that can be traversed with an applicative:

```
template <typename T, typename U,
    template <typename> typename Applicative>
Applicative<Box<U>> traverse(Box<T> box,
  std::function<Applicative<U>(T)> fn);
```

```
template <template <typename> typename Applicative, typename T>
Applicative<std::vector<T>> sequence(std::vector<ApplicativeT>> const& boxes) {
    return traverse(boxes, identity);
}
```

The Const functor

```
template <typename T>
struct Const {
    T value;
};

template <typename F>
struct fmap {
    F _fn;

    template <typename T>
    auto operator()(Const<T> t) const {
        return t; // ignore _fn
    }
};
```

The Const functor

```
template <typename T>
struct Const {
    T value;
};

template <typename F>
struct fmap {
    F _fn;

    template <typename ... M>
      requires (Monoid<M> && ...) && same_type<M...>
    auto operator()(Const<M>... ts) const {
        return /* fold over ts using the monoid */;
    }
};
```

Traversable implies foldable

```
template <typename T, typename M>
M fold_map(Box<T> box, std::function<M(T)> fn) {
    return traverse(box, [fn](T t) { return Const{fn(t)}; }).value;
}
```

Traversable implies foldable

```
template <typename T, typename M>
M fold_map(Box<T> box, std::function<M(T)> fn) {
    return traverse(box, [fn](T t) { return Const{fn(t)}; }).value;
}
```

But the reverse may require additional constraints.

The Identity functor

```
template <typename T>
struct Identity {
    T value;
};

template <typename F>
struct fmap {
    F _fn;

    template <typename ... Ts>
    auto operator()(Identity<Ts> ... ts) const {
        return Identity{_fn(ts.value...)};
    }
};
```

Traversable implies functor

```
template <typename T, typename U>
Box<U> fmap(Box<T> box, std::function<U(T)> fn) {
    return traverse(box, [fn](T t) { return Identity{fn(t)}); }).value;
}
```

Traversable implies functor

```
template <typename T, typename U>
Box<U> fmap(Box<T> box, std::function<U(T)> fn) {
    return traverse(box, [fn](T t) { return Identity{fn(t)}); }).value;
}
```

Therefore, foldable does not imply traversable.

Recap:

Foldable A box that can be summarized with a monoid

Traversable A box that can be traversed with an applicative

A taste of lenses

Composition with sub-parts

So far: Compose `T -> U` and `U -> V` to `T -> V` (potentially boxed).

Composition with sub-parts

So far: Compose `T -> U` and `U -> V` to `T -> V` (potentially boxed).

What about composing `T -> (U1, U2, U3, U4)` and `U3 -> V` to `T -> (U1, U2, V, U4)`?

Validating structs

```
struct ContactInformation {
    Email primary;
    Email recovery;
    Address physical;
};
```

Validating structs

```
struct ContactInformation {
    Email primary;
    Email recovery;
    Address physical;
};

std::optional<Email> validate_email(Email email);

std::optional<ContactInformation> validate(ContactInformation const& contact) {
    ???
}
```

Traversals

Traversal A customizable way to traverse an object.

```
template <template <typename> typename Applicative>
auto traversal(Obj obj, std::function<Applicative<T>(T)> fn)
  -> Applicative<Obj>;
```

Definining a traversal for our struct

```
struct EmailTraversal {
    template <template <typename> typename Applicative>
    Applicative<ContactInformation> operator()(
        ContactInformation const& self,
        std::function<Applicative<Email>(Email)> fn
    ) {
        return fmap([&](Email primary, Email recovery) {
            auto result = self;
            result.primary = primary;
            result.recovery = recovery;
            return result;
        })(fn(self.primary), fn(self.recovery));
    }
};
```

Validating structs

```
std::optional<ContactInformation> validate(ContactInformation const& contact) {
    return EmailTraversal{}(contact, validate_email);
}
```

Transforming structs

```
Email normalize_email(Email email);
```

Transforming structs

```
Email normalize_email(Email email);

ContactInformation normalize(ContactInformation const& contact) {
    return EmailTraversal{}(contact, normalize_email);
}
```

Transforming structs

```
Email normalize_email(Email email);

ContactInformation normalize(ContactInformation const& contact) {
    return EmailTraversal{}(contact, normalize_email);
}
```

But `normalize_email` does not return an `Applicative`!

Transforming structs

```
Email normalize_email(Email email);

ContactInformation normalize(ContactInformation const& contact) {
    return EmailTraversal{}(contact, [&](Email email) {
      return Identity{normalize_email(email)};
    }).value;
}
```

Transforming structs using over

```
template <typename Traversal, typename Obj, typename Fn>
Obj over(Obj obj, Fn fn) {
    return Traversal{}(obj, [&](auto x) {
        return Identity{fn(x)};
    }).value;
}
```

Transforming structs using over

```
template <typename Traversal, typename Obj, typename Fn>
Obj over(Obj obj, Fn fn) {
    return Traversal{}(obj, [&](auto x) {
        return Identity{fn(x)};
    }).value;
}

ContactInformation normalize(ContactInformation const& contact) {
    return over<EmailTraversal>(contact, normalize_email);
}
```

Setting struct members using `set`

```
template <typename Traversal, typename Obj, typename T>
Obj set(Obj obj, T value) {
    return over<Traversal>(obj, [&](auto) {
        return value;
    });
}
```

Setting struct members using set

```
template <typename Traversal, typename Obj, typename T>
Obj set(Obj obj, T value) {
    return over<Traversal>(obj, [&](auto) {
        return value;
    });
}

ContactInformation clear_emails(ContactInformation const& contact) {
    return set<EmailTraversal>(contact, Email{});
}
```

What about the address?

```
struct AddressTraversal {
    template <template <typename> typename Applicative>
    Applicative<ContactInformation> operator()(
      ContactInformation const& self,
      std::function<Applicative<Address>(Address)> fn
    ) {
        return fmap([&](Address address) {
            auto result = self;
            result.physical = address;
            return result;
        })(fn(self.physical));
    }
};
```

But there is only one address!

What about the address?

```
struct AddressLens {
    template <template <typename> typename Functor>
    Functor<ContactInformation> operator()(
      ContactInformation const& self,
      std::function<Functor<Address>(Address)> fn
    ) {
        return fmap([&](Address address) {
            auto result = self;
            result.physical = address;
            return result;
        })(fn(self.physical));
    }
};
```

Lenses

Lens A traversal that targets exactly one member.

```
template <template <typename> typename Functor>
auto lens(Obj self, std::function<Functor<T>(T)> fn) -> Functor<Obj>;
```

Every lens is a traversal

```
template <template <typename> typename Applicative>
auto traversal(Obj self, std::function<Applicative<T>(T)> fn)
  -> Applicative<Obj>;

template <template <typename> typename Functor>
auto lens(Obj self, std::function<Functor<T>(T)> fn)
  -> Functor<Obj>;
```

Contravariance of function arguments!

Using the lens

Validate the address

```
std::optional<Address> validate_address(Address address);

std::optional<ContactInformation> validate(ContactInformation const& contact) {
    return AddressLens{}(contact, validate_address);
}
```

Using the lens

Validate the address

```
std::optional<Address> validate_address(Address address);

std::optional<ContactInformation> validate(ContactInformation const& contact) {
    return AddressLens{}(contact, validate_address);
}
```

Normalize the address

```
Address normalize_address(Address address);

ContactInformation normalize(ContactInformation const& contact) {
    return over<AddressLens>(contact, normalize_address);
}
```

Accessing members using `view`

```
template <typename Lens, typename Obj>
auto view(Obj obj) {
    return Lens{}(obj, [](auto x) {
        return Const{x};
    }).value;
}
```

Accessing members using `view`

```
template <typename Lens, typename Obj>
auto view(Obj obj) {
    return Lens{}(obj, [](auto x) {
        return Const{x};
    }).value;
}

template <typename Obj>
Address get_address(Obj const& obj) {
    return view<AddressLens>(obj);
}
```

Accessing members using view

```
template <typename Lens, typename Obj>
auto view(Obj obj) {
    return Lens{}(obj, [](auto x) {
        return Const{x};
    }).value;
}

template <typename Obj>
Address get_address(Obj const& obj) {
    return AddressLens{}(obj, [](auto x) {
        return Const{x};
    }).value;
}
```

Accessing members using view

```
template <typename Lens, typename Obj>
auto view(Obj obj) {
    return Lens{}(obj, [](auto x) {
        return Const{x};
    }).value;
}

template <typename Obj>
Address get_address(Obj const& obj) {
    return fmap([&](Address address) {
        auto result = obj;
        result.physical = address;
        return result;
    })(Const{obj.physical}).value;
}
```

Accessing members using `view`

```
template <typename Lens, typename Obj>
auto view(Obj obj) {
    return Lens{}(obj, [](auto x) {
        return Const{x};
    }).value;
}

template <typename Obj>
Address get_address(Obj const& obj) {
    return Const{obj.physical}).value;
}
```

Accessing members using `view`

```
template <typename Lens, typename Obj>
auto view(Obj obj) {
    return Lens{}(obj, [](auto x) {
        return Const{x};
    }).value;
}

template <typename Obj>
Address get_address(Obj const& obj) {
    return obj.physical;
}
```

Recap

`Traversal` A custom way to traverse an object

`Lens` A traversal that targets exactly one member

Recap

`Traversal` A custom way to traverse an object

`Lens` A traversal that targets exactly one member

- `over<Traversal>(obj, fn)`: transform the members of `obj` targeted by `Traversal` using `fn`
- `set<Traversal>(obj, value)`: set the members of `obj` targeted by `Traversal` to `value`
- `view<Lens>(obj)`: return the member of `obj` targeted by `Lens`

Conclusion

Why do you care?

Why do you care?

- Composition patterns give vocabulary

Why do you care?

- Composition patterns give vocabulary
- Recognize relationships

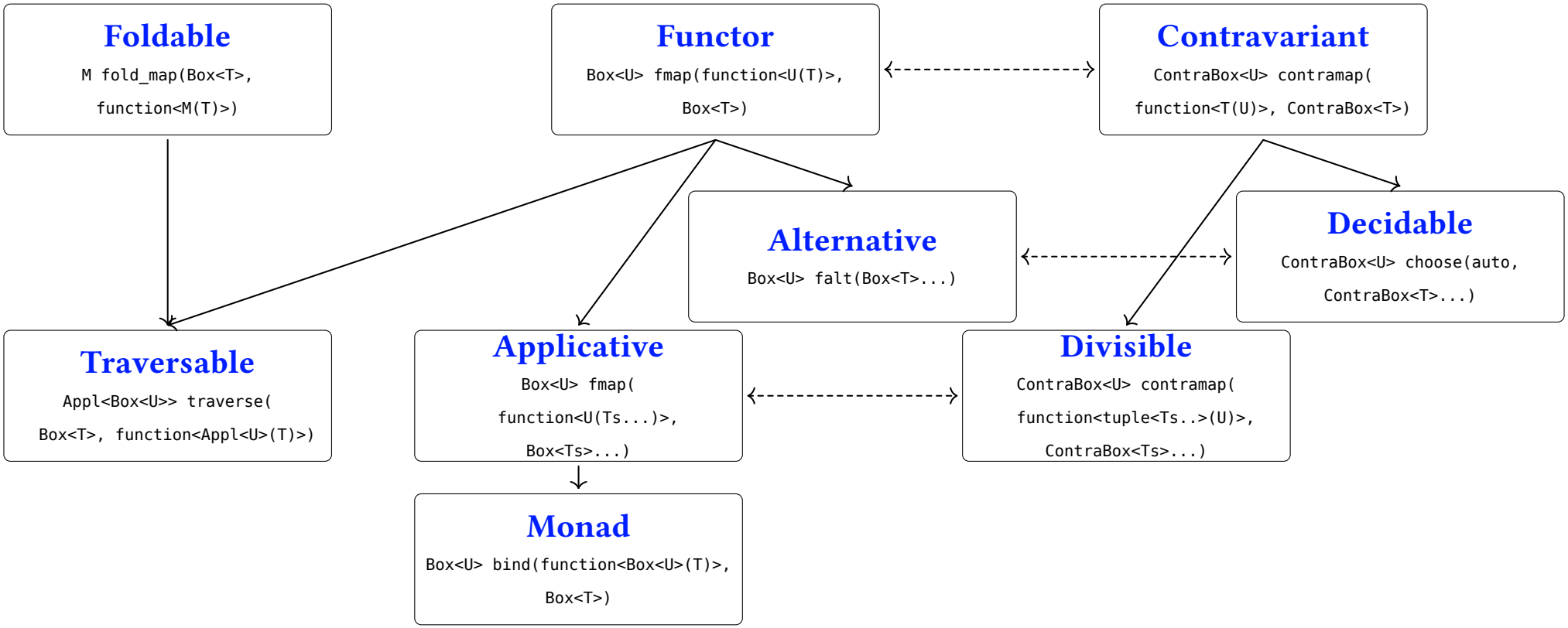
Why do you care?

- Composition patterns give vocabulary
- Recognize relationships
- Higher-order generic code

Functional programming is all about

COMPOSITION.

Functional programming is all about COMPOSITION.

1. Write well-tested, optimized building blocks.
2. Compose them using functional paradigms.

Functional composition beyond monads