APPLICATIVE

THE FORGOTTEN FUNCTIONAL PATTERN

BEN DEANE / C++ NOW / MAY 2023

DISCLAIMERS

No AI was used in the creation of this talk.

All code has compiled on some compiler at some point, but may have been accidentally broken in slide translation. It's platonically correct!

Please ask questions as we go; I'll answer/defer as necessary.

GOAL FOR THIS TALK

"Discovery consists of seeing what everybody has seen and thinking what nobody has thought."

-- Albert Szent-György

I hope that by the end of the talk some of you will look at your code with new eyes.

PART 1

I Begin Life on My Own Account, and Don't Like It

- Refresher (or introduction) for functor and monad
- Where applicative fits

FIRST, A QUICK SURVEY

FIRST, A QUICK SURVEY

- I've heard of these functional patterns, but that's all

FIRST, A QUICK SURVEY

- I've heard of these functional patterns, but that's all
- I (think I) kind of understand how they work

FIRST, A QUICK SURVEY

- I've heard of these functional patterns, but that's all
- I (think I) kind of understand how they work
- I don't see many use cases in my code for them

FIRST, A QUICK SURVEY

- I've heard of these functional patterns, but that's all
- I (think I) kind of understand how they work
- I don't see many use cases in my code for them
- I've written code that uses them, and it was good

FIRST, A QUICK SURVEY

- I've heard of these functional patterns, but that's all
- I (think I) kind of understand how they work
- I don't see many use cases in my code for them
- I've written code that uses them, and it was good
- I completely understand a) functors b) monads c) applicative

FIRST, A QUICK SURVEY

- I've heard of these functional patterns, but that's all
- I (think I) kind of understand how they work
- I don't see many use cases in my code for them
- I've written code that uses them, and it was good
- I completely understand a) functors b) monads c) applicative
- Haskell informs all my API designs

OR MAYBE THIS...?

Well, I thought I knew about monads until you just said that, now I'm not so sure…

OR MAYBE THIS...?

Well, I thought I knew about monads until you just said that, now I'm not so sure…

Don't worry about what a <functional pattern> *is*.
Rather we'll think about *what it does*.

REVIEW: (?) FUNCTORS

A functor is a parameterized type (a class template) of one (free) argument.

Some examples of functors:

- `optional`
- `vector` (sequences in general)
- `expected` (with error type bound)

The characteristic function is `fmap`.

fmap IN C++

```
template <typename F,
          template <typename> typename Functor,
          typename T>
constexpr auto fmap(F f, Functor<T> a)
  -> Functor<std::invoke_result_t<F, T>> {
  return Functor{f(unwrap(a))};
}
```

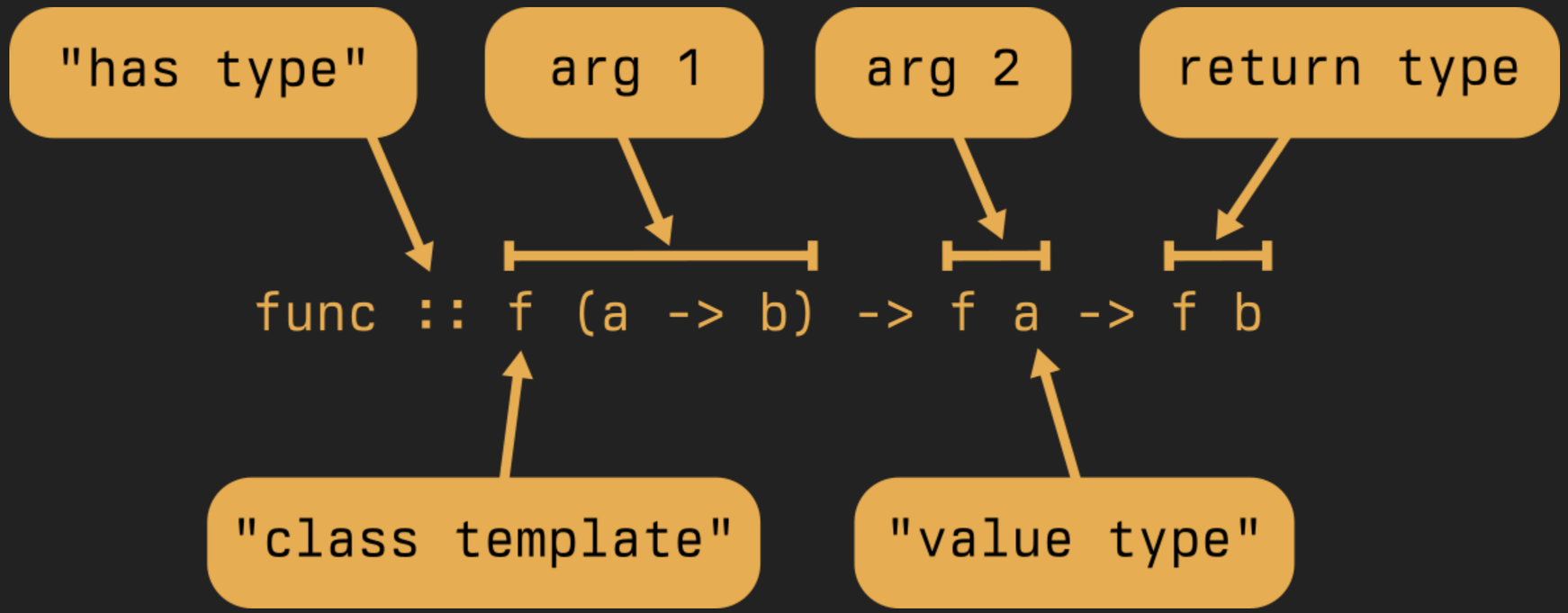
`fmap` IN HASKELL

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

For this reason I will be using haskell type signatures to explain applicative.

Note: `fmap` is also spelt `(<$>)` when used as an infix operator.

HOW TO READ HASKELL SIGNATURES



REVIEW: (?) MONADS

A monad is *also* a parameterized type (a class template) of one (free) argument.

Some examples of monads:

- `optional`
- `vector` (sequences in general)
- `expected` (with error type bound)

(Seem familiar?)

The characteristic function is `bind` (or `join`).

`bind` IN C++

```
template <typename F,
          template <typename> typename Functor,
          typename T>
constexpr auto bind(F f, Functor<T> a) -> Functor<...>;
```

Very similar to `fmap`, but this time the function returns a functorial value.

`bind` IN HASKELL

```
(>>=) :: m a -> (a -> m b) -> m b
```

`bind` is spelt `(>>=)`

`fmap` + `join` = `bind`

When we try to `fmap` a function that returns a functorial value,
the value is double-wrapped.

```
fmap :: (a -> b) -> m a -> m b
join :: m (m a) -> m a

f :: a -> m b

fmap f :: m a -> m (m b)
join . fmap f :: m a -> m b

(=<<) f :: m a -> m b
```

And `join` collapses the double layer to one layer again.

COMPARE: `fmap` AND `bind`

```
fmap  :: (a ->   b) -> f a -> f b
(<$>) :: (a ->   b) -> f a -> f b    -- also fmap

(=<<) :: (a -> m b) -> m a -> m b    -- bind with flipped args

(>>=) :: m a -> (a -> m b) -> m b
```

`(=<<)` is the same as `(>>=)` but with flipped arguments

MONADIC FUNCTIONS FOR `std::optional`

```
template <typename F>
constexpr auto transform(F&& f);
```

"If `*this` contains a value, invokes `f` with the contained value as an argument, and returns a `std::optional` that contains the result of that invocation; otherwise, returns an empty `std::optional`."

(this is `fmap`)

MONADIC FUNCTIONS FOR `std::optional`

```
template <typename F>
constexpr auto and_then(F&& f);
```

"If `*this` contains a value, invokes `f` with the contained value as an argument, and returns the result of that invocation; otherwise, returns an empty `std::optional`.

The return type must be a specialization of `std::optional` (unlike `transform()`)."

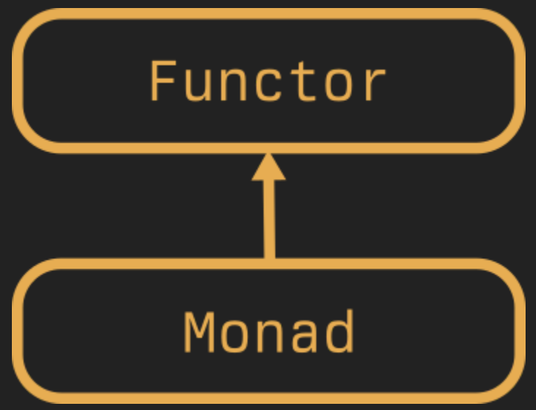
(this is `bind`)

HISTORY OF APPLICATIVE

At first, Haskell has only Functor and Monad.

Functor and Monad classes first appeared in Haskell in the Haskell Report 1.3, May 1996.

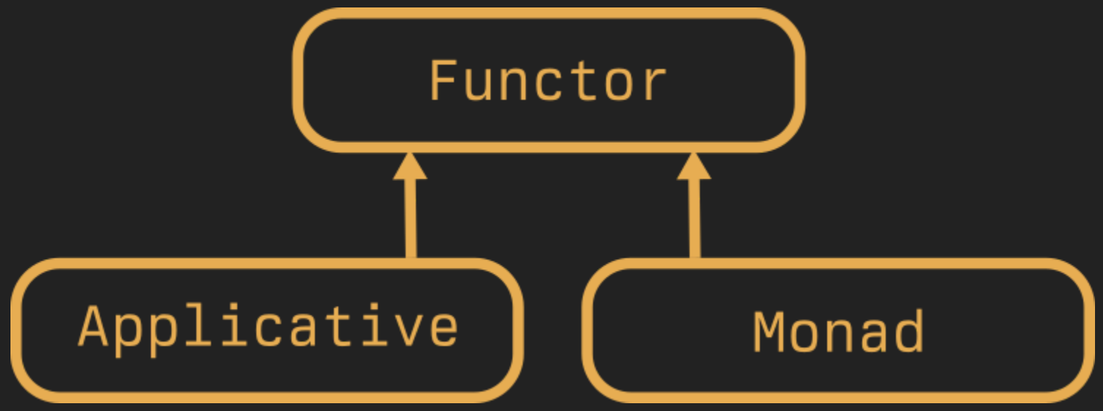
`IO`, `[]` and `Maybe` were both Functors and Monads at this time.



HISTORY OF APPLICATIVE

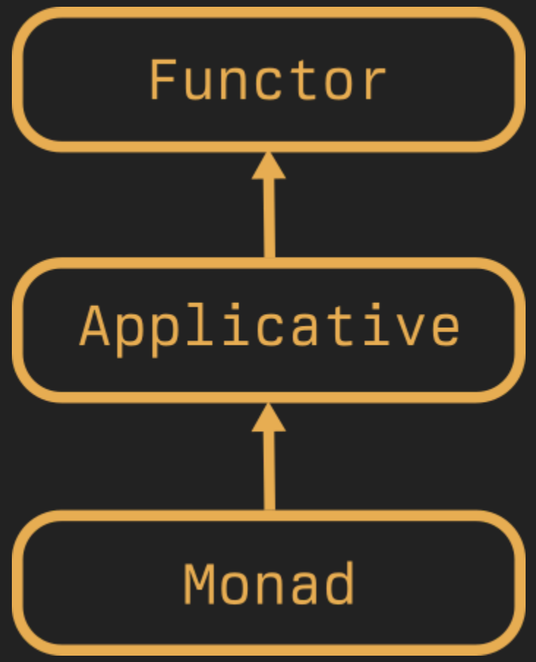
Applicative appeared in *Applicative Programming with Effects* by Conor McBride and Ross Paterson, published in the Journal of Functional Programming in January 2008.

Applicative existed entirely separately from Monad for a while
(both being "derived from" Functor).



HISTORY OF APPLICATIVE

In 2014 a proposal was implemented in GHC 7.10 which interposed Applicative between Functor and Monad. Every Monad is also an Applicative.



And Functorial/Monadic patterns show up in C++ sometime post C++11.

PART 2

In Which We Begin Not To Understand

- The mechanics of Applicative
- Applicative in C++

A VERY BRIEF SURVEY OF EXISTING WORK

- `Functional Programming: Functors and Monads`
  (Michał Dominiak, CppCon 2015)
- `Monoids, Monads and Applicative Functors: Repeated Software Patterns`
  (David Sankel, C++Now 2015, `repeated` at CppCon 2020)

THE MECHANICS OF APPLICATIVE

Basically all existing C++ work on applicative says the same thing:

An applicative functor is characterised by the apply function, which applies a function inside the functor to a value inside the functor.

```
(<$>) ::   (a ->   b) -> f a -> f b    -- fmap
(=<<) ::   (a -> m b) -> m a -> m b    -- (flipped) bind

(<*>) :: f (a ->   b) -> f a -> f b    -- ap(ply)
```

And this can be used to apply functions of more than one argument.

THE POINT OF APPLICATIVE!

And this can be used to apply functions of more than one argument!

APPLICATIVE GENERALIZES FMAP

How to write fmap for more than one-arg functions?

```
fmap0 ::  a                    -> f a
fmap1 :: (a -> b)              -> f a -> f b
fmap2 :: (a -> b -> c)         -> f a -> f b -> f c
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

`fmap0` is also called `pure`.

APPLICATIVE IN HASKELL

```haskell
pure :: a -> f a

(<*>) :: f (a -> b) -> f a -> f b
```

Apply is spelt `(<*>)`.

And in Haskell it has this strange signature because of the way Haskell works.

Hence, people say "it applies a function inside the functor".

HASKELL DOESN'T HAVE N-ARGUMENT FUNCTIONS

Haskell has built-in partial application/curried functions.

All functions are unary functions.

When we call an "n-argument" function with one argument, we get an "n-1 argument" function returned. (This is why Haskell type signatures have multiple arrows, not commas.)

This is why applicative has this form… *it's because of Haskell function mechanics.*

MECHANICS OF APPLICATIVE IN HASKELL

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

fmap (+) (Just 1) :: Maybe (Int -> Int)
(<*>) (fmap (+) (Just 1)) (Just 2) :: Maybe Int
```

MECHANICS OF APPLICATIVE IN HASKELL

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

(+) 1 2
-- => 3

     (+) <$> (Just 1) <*> (Just 2)
-- => Just 3
pure (+) <*> (Just 1) <*> (Just 2)
-- => Just 3

pure (+) <*> (Just 1)
-- => Just (+1)
```

A VERY BRIEF SURVEY OF EXISTING C++ WORK

- P0650 (C++ Monadic Interface, Vicente J. Botet Escribá) "TODO: Add some motivation and rationale for `ap`."
- P2502 (Monadic Functions for `std::expected`, Jeff Garland) No mention of applicative.
- P0798 (Monadic Functions for `std::optional`, Sy Brand) "I couldn't think of many use-cases of this in C++."
- P2374 (`views::cartesian_product`, Sy Brand & Michał Dominiak) No mention of applicative.
- P2321 (`zip`, Tim Song) No mention of applicative.
- P2561 (An Error Propagation Operator, Barry Revzin) No mention of applicative.
- P2300 (`std::execution`, many authors) No mention of applicative (or monad - on purpose)?

PART 3

A Long-Expected Party

- Applying (ha) applicative to `std::optional` / `std::expected`
- (Nearly) no more Haskell code
- Lots more C++ code

P2561

P2561 (An Error Propagation Operator) posits the following code:

```
auto foo(int) -> std::expected<int, E>;
auto bar(int) -> std::expected<int, E>;

auto strcat(int i) -> std::expected<std::string, E>
{
  auto f = foo(i);
  if (not f) {
    return std::unexpected(f.error());
  }

  auto b = bar(i);
  if (not b) {
    return std::unexpected(b.error());
  }

  return std::format("{}{}", *f, *b);
}
```

PARAPHRASING P2561

"How is this problem solved today? Either:

- manual error propagation
- macros (most libraries e.g. Boost.Outcome)

The only things that have nice syntax are exception and coroutines [both of which may entail unacceptable overhead].

The best case today involves macros."

FROM P0798

"12.6 Applicative functors

…

I couldn’t think of many use-cases of this in C++."

Sy, Barry has your use case!

LET'S ANSWER THE QUESTION FROM P2561

Let's try to write a nicer-looking way to handle applying a function to multiple (possibly failing) values.

Goals:

- no macros
- no manual control flow
- declarative expression-oriented style
- good-looking code
- efficient

Let's do it with `optional` first before we tackle `expected`.

MONADIC SOLUTION

To begin with, we can solve this with the monadic API of course.

```
auto strcat(int i) -> std::optional<std::string> {
  return foo(i).and_then([&](auto f) {
    return bar(i).and_then([&](auto b) -> std::optional<std::string> {
      return std::format("{}{}", f, b);
    });
  });
}
```

HOW DID WE DO?

No macros?	❤️
No manual control flow?	💔
Declarative style?	☠️
Good-looking code?	🤮
Efficient?	❤️

APPLICATIVE SOLUTION

We'll start by implementing partial application in C++.

PARTIAL APPLICATION, C++ STYLE

```
template <typename F, typename... Args>
constexpr auto curry(F &&f, Args &&...args) -> decltype(auto) {
  if constexpr (std::invocable<F, Args...>) {
    return std::invoke(std::forward<F>(f), std::forward<Args>(args)...);
  } else {
    return [f = std::forward<F>(f),
            ...args = std::forward<Args>(args)]<typename Self, typename... As>(
               this Self&& self, As &&...as) -> decltype(auto) {
      return curry(std::forward_like<Self>(f),
                   std::forward_like<Self>(args)...,
                   std::forward<As>(as)...);
    };
  }
}
```

PARTIAL APPLICATION, C++ STYLE

(Now you see one of my personal incentives for P0847 *Deducing* `this`)

See also *C++Weekly Episode 318: My Meetup Got Nerd Sniped!*

Tip: attend your local meetup! If you don't have one,
consider attending the *Denver C++ Meetup* online! Find us on `meetup.com`.

OK, LET'S TRY IT

Here's `fmap`, taking into account partial application.

```
template <typename F, typename TOpt>
constexpr auto fmap(F &&f, TOpt &&opt)
    -> std::optional<decltype(curry(std::forward<F>(f),
                                    *std::forward<TOpt>(opt)))> {
  if (opt) {
    return curry(std::forward<F>(f), *std::forward<TOpt>(opt));
  }
  return {};
}
```

OK, LET'S TRY IT

Here's ap, taking into account partial application.

```
template <typename FOpt, typename TOpt>
constexpr auto ap(FOpt &&f, TOpt &&opt)
    -> std::optional<decltype(curry(*std::forward<FOpt>(f),
                                    *std::forward<TOpt>(opt)))> {
  if (f and opt) {
    return curry(*std::forward<FOpt>(f), *std::forward<TOpt>(opt));
  }
  return {};
}
```

OK, LET'S TRY IT

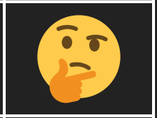
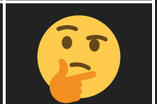
```
constexpr auto foo(int) -> std::optional<int>;
constexpr auto bar(int) -> std::optional<int>;

auto strcat(int i) -> std::optional<std::string> {
  auto f = foo(i);
  auto b = bar(i);

  auto l = [](int f, int b) -> std::string {
    return std::format("{}{}", f, b);
  };

  return ap(fmap(l, f), b);
}
```

HOW DID WE DO?

No macros?	
No manual control flow?	
Declarative style?	
Good-looking code?	
Efficient?	

HOW DID WE DO?

HOW DID WE DO?

- no `if` statements

HOW DID WE DO?

- no `if` statements
- no early `return` statements

HOW DID WE DO?

- no `if` statements
- no early `return` statements
- no manual control flow

HOW DID WE DO?

- no `if` statements
- no early `return` statements
- no manual control flow
- not bad compared to exceptions/coroutines versions

HOW DID WE DO?

- no `if` statements
- no early `return` statements
- no manual control flow
- not bad compared to exceptions/coroutines versions
- we did wrap `std::format` in a lambda

HOW DID WE DO?

- no `if` statements
- no early `return` statements
- no manual control flow
- not bad compared to exceptions/coroutines versions
- we did wrap `std::format` in a lambda
- we could have just curried `std::format` if we were allowed to

HOW DID WE DO?

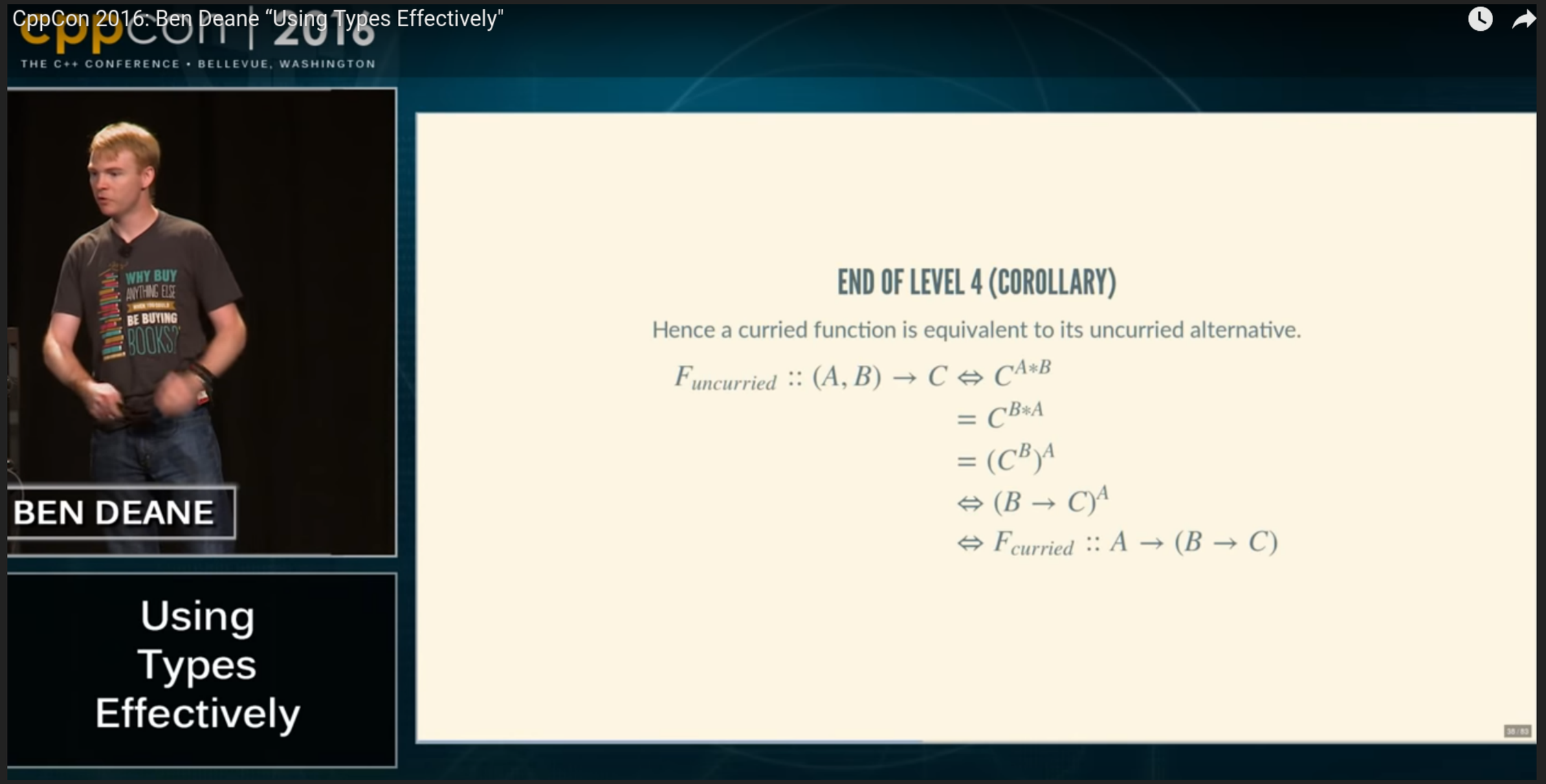
- no `if` statements
- no early `return` statements
- no manual control flow
- not bad compared to exceptions/coroutines versions
- we did wrap `std::format` in a lambda
- we could have just curried `std::format` if we were allowed to
- prefix syntax is a bit awkward

HOW DID WE DO?

- no `if` statements
- no early `return` statements
- no manual control flow
- not bad compared to exceptions/coroutines versions
- we did wrap `std::format` in a lambda
- we could have just curried `std::format` if we were allowed to
- prefix syntax is a bit awkward

But C++ isn't Haskell… we have functions of more than one argument!

BEFORE YOU SAY IT... I KNOW



LET'S TRY IT (TAKE 2)

We have functions of more than one argument in C++, so why not have `fmap` take multiple arguments to start with?

LET'S TRY IT (TAKE 2)

```
template <typename F, typename... TOpts>
constexpr auto fmap(F &&f, TOpts &&...opts)
    -> std::optional<decltype(std::invoke(std::forward<F>(f),
                                          *std::forward<TOpts>(opts)...))> {
  if ((... and opts)) {
    return std::invoke(std::forward<F>(f), *std::forward<TOpts>(opts)...);
  }
  return {};
}
```

Notice:

- we don't need `curry` any more (C++ is not Haskell)
- this is not the same as `optional::transform`

LET'S TRY IT (TAKE 2)

```
auto strcat(int i) -> std::optional<std::string> {
  auto f = foo(i);
  auto b = bar(i);

  auto l = [](int f, int b) -> std::string {
    return std::format("{}{}", f, b);
  };

  return fmap(l, f, b);
}
```

WAS THAT BETTER?

No macros?	❤️
No manual control flow?	❤️
Declarative style?	❤️
Good-looking code?	❤️
Efficient?	🤔

WAS THAT BETTER?

```
auto strcat(int i) -> std::optional<std::string> {
  auto f = foo(i);

  // if foo failed, we don't return here:
  // we still evaluate bar(i)

  auto b = bar(i);

  auto l = [](int f, int b) -> std::string {
    return std::format("{}{}", f, b);
  };

  return fmap(l, f, b);
}
```

A FUNDAMENTAL ISSUE

Haskell is lazy by default.

C++ is not.

To get around this and still be declarative,
we'll need to evaluate function arguments lazily.

TAKE 2.5

```
return fmap(l, lazy_call(foo, i), lazy_call(bar, i));
```

```
template <typename F, typename... Args> auto lazy_call(F &&f, Args &&...args) {
  return lazy{[&] {
    return std::invoke(std::forward<F>(f), std::forward<Args>(args)...);
  }};
}
```

TAKE 2.5

```cpp
template <typename F> struct lazy {
  F f{};
  std::invoke_result_t<F> opt{};

  constexpr explicit(true) operator bool() {
    opt = std::invoke(f);
    return opt.has_value();
  }

  template <typename Self>
  constexpr auto operator*(this Self&& self) {
    return *(std::forward<Self>(self).opt);
  }
};
```

TAKE 2.5

```
auto strcat(int i) -> std::optional<std::string> {
  auto l = [](int f, int b) -> std::string {
    return std::format("{}{}", f, b);
  };

  return fmap(l, lazy_call(foo, i), lazy_call(bar, i));
}
```

Better? Maybe? Yes?

Take your pick. Use lazy evaluation if you want.
Or just take the eager function evaluation.
It really depends on how expensive your functions are?

Q: DOES THE OPTIMIZER SEE THROUGH ALL THIS?

A: Yes (at least in my experiments)

HOW DID WE DO?

No macros?	
No manual control flow?	
Declarative style?	
Good-looking code?	
Efficient?	

Let's see what it looks like with `std::expected`.

EXTENDING THIS TO `std::expected`

```
auto strcat(int i) -> std::expected<std::string, E> {
  auto l = [](int f, int b) -> std::string {
    return std::format("{}{}", f, b);
  };

  return fmap(l, lazy_call(foo, i), lazy_call(bar, i));
}
```

Can we make this code work the same way for `std::expected`?

Of course you know the answer is yes.

(See: Betteridge's Law of Presentations.)

EXTENDING THIS TO `std::expected`

```
template <typename F, typename... TExps> auto fmap(F &&f, TExps &&...exps) {
  using T = decltype(std::invoke(std::forward<F>(f), *std::forward<TExps>(exps)...));
  using E = std::common_type_t<typename std::remove_cvref_t<TExps>::error_t...>;
  using R = std::expected<T, E>;
  if ((... and exps)) {
    return R{std::invoke(std::forward<F>(f), *std::forward<TExps>(exps)...)};
  }

  // else return an error
  ...
}
```

The success case is basically the same as with `std::optional`…

(In real life I would probably constrain the function arguments to take a fixed error type…
here I used `std::common_type_t` for expediency.)

EXTENDING THIS TO `std::expected`

```
template <typename F> struct lazy {
  using error_t = std::invoke_result_t<F>::error_type;
  F f{};
  std::invoke_result_t<F> exp{};

  constexpr explicit(true) operator bool() {
    exp = std::invoke(f);
    return exp.has_value();
  }

  template <typename Self>
  constexpr auto operator*(this Self&& self) {
    return *std::forward<Self>(self).exp;
  }
};
```

For the success case, basically the same as the lazy `std::optional`…

HOW TO RECOVER THE ERROR?

We want short circuiting (actually we *need* it now since we already short-circuited evaluating the functions). We used an `and` fold to get the values. How about an `or` fold to get the error?

```
template <typename T> struct invert {
  T t;

  constexpr explicit(false) operator bool() {
    return not t.exp.has_value();
  }

  template <typename Self>
  constexpr auto error(this Self&& self) {
    return std::forward<Self>(self).exp.error();
  }
};
```

Plan: invert the sense and then write a fold expression to extract the error.

HOW TO RECOVER THE ERROR?

```
template <typename F, typename... TExps> auto fmap(F &&f, TExps &&...exps) {
  ...

  // else return an error
  std::optional<E> error{};
  const auto extract_error = [&] <typename TInv> (TInv&& t) -> bool {
    if (t) {
      error = std::forward<TInv>(t).error();
    }
    return t;
  };
  (... or extract_error(invert{std::forward<Ts>(ts)}));
  return R{std::unexpect, *error};
}
```

C++ is statically typed so there's no such thing as "truthy" values… the fold must return a `bool`. Hence the awkward extraction.

We know it's safe to unwrap the optional because we *know* one of the functions failed.

WITH `std::expected`

```
return fmap(l, lazy_call(foo, i), lazy_call(bar, i));
```

The same call-site structure as with `std::optional`.

This time, `foo` and `bar` return `std::expected<int, E>`.

Q: DOES THE OPTIMIZER *STILL* SEE THROUGH ALL THIS?

A: Again yes (at least in my experiments)

A CONCLUSION FROM THIS EXPERIMENT?

Does C++ need an error propagation operator? Maybe. C++ is full of things that are done in libraries that could be done better in the language.

But: we need to be aware of applicative solutions when we have this conversation.

Because the problem posed here is *what applicative is for*.

IMO the best case today does not involve macros.

PART 4

What I Saw of the Destruction of Weybridge and Shepperton

- Applicative vs Monad?

A QUESTION OF POWER?

A QUESTION OF POWER?

Every monad is an applicative.

A QUESTION OF POWER?

Every monad is an applicative.

Monad is "more powerful" than applicative.

A QUESTION OF POWER?

Every monad is an applicative.

Monad is "more powerful" than applicative.

You can do everything with monad that you can do with applicative.

A QUESTION OF POWER?

Every monad is an applicative.

Monad is "more powerful" than applicative.

You can do everything with monad that you can do with applicative.

But: more power means more constraints.

ANYTHING APPLICATIVE CAN DO...

...monad can do. But not better.

```
auto strcat(int i) -> std::optional<std::string> {
  return foo(i).and_then([&](auto f) {
    return bar(i).and_then([&](auto b) -> std::optional<std::string> {
      return std::format("{}{}", f, b);
    });
  });
}
```

WHAT'S THE DIFFERENCE?

WHAT'S THE DIFFERENCE?

Monad is for when you need to make choices based on what came before.

WHAT'S THE DIFFERENCE?

Monad is for when you need to make choices based on what came before.

In other words, when you have data dependencies.

WHAT'S THE DIFFERENCE?

Monad is for when you need to make choices based on what came before.

In other words, when you have data dependencies.

Monadic `bind` is *inherently serial*.

WHAT'S THE DIFFERENCE?

Monad is for when you need to make choices based on what came before.

In other words, when you have data dependencies.

Monadic `bind` is *inherently serial*.

Applicative is *inherently parallel* in evaluating arguments.

APPLICATIVE IS PARALLEL IN NATURE

```
template <typename F, typename... Ts> auto fmap(F &&f, Ts &&...ts) {
  // ...

  // here we have an engineering choice:
  // we can evaluate all these in parallel!
  if ((... and ts)) {
    // return success
  }
  // return failure
}
```

MONAD VS APPLICATIVE

Use a monad when you need that dependent choice.

Otherwise, why constrain yourself to serial operation?

In part, this is how functional programmers think about architecture.

PART 5

In which Eeyore has a birthday and gets two presents

- more applicatives
- sequences

SEQUENCES

A range is a functor.

`fmap` is implemented by `transform`.

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

```cpp
auto ints = std::views::iota(1, 10);
auto squares = ints
  | std::views::transform([] (auto i) { return i * i; });
```

SEQUENCES

A range is also a monad.

`bind` is implemented by `transform | join`.

```haskell
(=<<) :: (a -> m b) -> m a -> m b -- transform | join
```

```cpp
auto next_positions(Board b) -> std::ranges::range auto;

auto possibilities =
  std::views::single(current_board)
  | std::views::transform(next_positions)
  | std::views::join;
```

EVERY MONAD IS AN APPLICATIVE - HOW?

So how is range an applicative?

```
fmap  ::    (a ->   b) -> f a -> f b -- transform
(=<<) ::    (a -> m b) -> m a -> m b -- transform | join
(<*>) :: f (a ->   b) -> f a -> f b -- ???
```

We need a function that takes a range of functions and applies them to a range of values…

EVERY MONAD IS AN APPLICATIVE - HOW?

…or a function that generalizes `fmap`.

```
fmapN :: (a -> b -> ...) -> f a -> f b -> ...
```

Recall: this is the *reason* to have `(<*>)`.

ranges::views::zip_transform

```
template <typename F, std::ranges::viewable_range... Rs>
constexpr auto zip_transform(F&& f, Rs&&... rs);
```

"…produces a view whose i^{th} element is the result of applying the invocable object to the i^{th} elements of all views."

ranges::zip_transform_view

So `ranges::zip_transform_view` is an applicative functor.

Is it also a monad? Let's go through the types involved.

```
-- f1 and f2 are functions that return lists
f1 :: a -> [a]
f2 :: a -> [a]

-- fmap "zips" the lists together
[f1, f2] <$> [a, b]
-- => [[f1a, f1b, f1c], [f2a, f2b]]
```

How can we `join` this result?

`ranges::zip_transform_view`

The natural way to join would be to take the i^{th} element from the i^{th} list.

But the i^{th} list might not have an i^{th} element.

Other ways to join the list run afoul of monad laws (usually associativity).

So `ranges::zip_transform_view` isn't a monad. But it is an applicative!

LET'S TRY AGAIN

There's another function with (almost) this signature.

```
fmapN :: (a -> b -> ...) -> f a -> f b -> ...
```

```
template <ranges::viewable_range... Rs>
constexpr auto cartesian_product(Rs&&... rs);
```

The function that's (implicitly) applied here is `std::make_tuple`.

ranges::cartesian_product_view

```
cartesian_product :: R a -> R b -> ... -> R (a, b, c)
cartesian_product [1,2,3] [a,b,c]
-- => [{1, a}, {1, b}, {1, c},
--     {2, a}, {2, b}, {2, c},
--     {3, a}, {3, b}, {3, c}]
```

And then we can `fmap` (`transform`) the resulting range with a formulation of `std::apply`.

ranges::cartesian_product_view

So `ranges::cartesian_product_view` is an applicative functor.

Is it also a monad?

```
-- f is a function that returns a list
f :: a -> b -> [c]

cartesian_product :: R a -> R b -> ... -> R (a, b, c)
cartesian_product [a1, a2] [b1, b2]
-- => [{a1, b1}, {a1, b2}, {a2, b1}, {a2, b2}]

-- now replacing make_tuple with f
-- => [[c11], [c12], [c21], [c22]]
```

How can we `join` this result? Just by joining the lists.

`ranges::cartesian_product_view` is also a monad.

IN CASE IT ESCAPED YOUR NOTICE...

C++23 sort of (?) has list comprehensions.

```python
[x * y for x in [2,3] for y in [4,5]]
# => [8, 10, 12, 15]
```

```haskell
[x * y | x <- [2,3], y <- [4,5]]
-- => [8, 10, 12, 15]
```

```cpp
auto const xs = std::array{2,3};
auto const ys = std::array{4,5};
auto const r = std::views::cartesian_product(xs, ys)
  | std::views::transform([] (auto const& t) {
    return std::apply(std::multiplies{}, t);
  });
// => {8, 10, 12, 15}
```

Admittedly ~~they suck~~ they're a bit verbose compared Python/Haskell.

PART 6

Cetology

- a bit of fun

WHAT ABOUT HETEROGENEOUS SEQUENCES?

Can we make `cartesian_product` for `std::tuple`?

Of course you know the answer is yes.

(See: Betteridge's Law of Presentations.)

CARTESIAN PRODUCT FOR TUPLES

```
auto const xs = std::tuple{"A", 1, true};
auto const ys = std::tuple{'x', 3.14f};
auto const t = cartesian_product(xs, ys);
// => {{"A",  'x'}, {"A",  3.14f},
//     {1,    'x'}, {1,    3.14f},
//     {true, 'x'}, {true, 3.14f}}
```

CARTESIAN PRODUCT FOR TUPLES

```
cartesian_product({1, 2}, {3, 4});
// => {{1, 3}, {1, 4}, {2, 3}, {2, 4}}

cartesian_product({1, 2});
// => {{1}, {2}}

cartesian_product({});
// => {{}}
```

CARTESIAN PRODUCT FOR TUPLES

Sketch of a recursive algorithm:

CARTESIAN PRODUCT FOR TUPLES

Sketch of a recursive algorithm:

1. Split the given tuples into first and rest

CARTESIAN PRODUCT FOR TUPLES

Sketch of a recursive algorithm:

1. Split the given tuples into first and rest
2. Let C be the cartesian product of the rest

CARTESIAN PRODUCT FOR TUPLES

Sketch of a recursive algorithm:

1. Split the given tuples into first and rest
2. Let C be the cartesian product of the rest
3. For each element in the first tuple, prepend it to each tuple in C

CARTESIAN PRODUCT FOR TUPLES

Sketch of a recursive algorithm:

1. Split the given tuples into first and rest
2. Let C be the cartesian product of the rest
3. For each element in the first tuple, prepend it to each tuple in C
4. Base case: if there are no tuples left, return a tuple of one empty tuple

CARTESIAN PRODUCT FOR TUPLES

```
template <typename... Ts> constexpr auto cartesian_product(Ts &&...ts) {
  if constexpr (sizeof...(Ts) == 0) {
    return std::make_tuple(std::tuple{});
  } else {
    return []<typename First, typename... Rest>(First &&first, Rest &&...rest) {
      auto const c = cartesian_product(std::forward<Rest>(rest)...);
      return std::apply(
          [&]<typename... Elems>(Elems &&...elems) {
            const auto prepend = [&]<typename E>(E &&e) {
              return std::apply(
                  [&](auto... subs) {
                    return std::make_tuple(std::tuple_cat(
                        std::forward_as_tuple(std::forward<E>(e)), subs)...);
                  },
                  c);
            };
            return std::tuple_cat(prepend(std::forward<Elems>(elems))...);
          },
          std::forward<First>(first));
    }(std::forward<Ts>(ts)...);
  }
}
```

BY THE WAY

Have I mentioned how annoying this can be when it takes you by surprise in generic code?

```
auto t = std::tuple{std::tuple{}};
```

OF COURSE THAT USED `std::make_tuple`

Let's generalize it.

It might actually be easier if we *don't* fix the function to `std::make_tuple`.

Notice that in the recursive solution, we build up the tuples as if by partial application, and applying arguments "in reverse".

GENERALIZING

Let's reuse `curry` but flip the argument application.

```
template <typename F, typename... Args>
constexpr auto flip_curry(F &&f, Args &&...args) -> decltype(auto) {
  if constexpr (std::invocable<F, Args...>) {
    return std::invoke(std::forward<F>(f), std::forward<Args>(args)...);
  } else {
    return [f = std::forward<F>(f),
            ... args = std::forward<Args>(args)]<typename... As>(
               As &&...as) -> decltype(auto) {
      return flip_curry(f, std::forward<As>(as)..., args...);
    };
  }
}
```

WITH GENERALIZED FUNCTION

```
template <typename... Ts> constexpr auto cartesian_product(auto &&f, Ts &&...ts) {
  if constexpr (sizeof...(Ts) == 0) {
    return std::make_tuple(flip_curry(f));
  } else {
    return [&]<typename First, typename... Rest>(First &&first, Rest &&...rest) {
      auto const c = cartesian_product(f, std::forward<Rest>(rest)...);
      return std::apply(
          [&]<typename... Elems>(Elems &&...elems) {
            const auto prepend = [&]<typename E>(E &&e) {
              return std::apply(
                  [&](auto... subs) {
                    return std::make_tuple(subs(std::forward<E>(e))...);
                  },
                  c);
            };
            return std::tuple_cat(prepend(std::forward<Elems>(elems))...);
          },
          std::forward<First>(first));
    }(std::forward<Ts>(ts)...);
  }
}
```

WITH std::make_tuple

```
template <typename... Ts> constexpr auto cartesian_product(Ts &&...ts) {
  if constexpr (sizeof...(Ts) == 0) {
    return std::make_tuple(std::tuple{});
  } else {
    return []<typename First, typename... Rest>(First &&first, Rest &&...rest) {
      auto const c = cartesian_product(std::forward<Rest>(rest)...);
      return std::apply(
          [&]<typename... Elems>(Elems &&...elems) {
            const auto prepend = [&]<typename E>(E &&e) {
              return std::apply(
                  [&](auto... subs) {
                    return std::make_tuple(std::tuple_cat(
                        std::forward_as_tuple(std::forward<E>(e)), subs)...);
                  },
                  c);
            };
            return std::tuple_cat(prepend(std::forward<Elems>(elems))...);
          },
          std::forward<First>(first));
    }(std::forward<Ts>(ts)...);
  }
}
```

PART 7

Down the Rabbit Hole

- even more applicatives
- parsing
- asynchronous execution

PARSING

One of the original uses of applicative.

Recall: "A parser for things is a function from strings to lists of pairs of things and strings."

```
Parser a :: String -> [(a, String)]
```

```
template <typename T>
using parser = auto (*)(string_view) -> range<pair<T, string_view>>;
```

PARSING WITH APPLICATIVE

From *`constexpr` ALL the things!*

```
template <typename P1, typename P2, typename F,
          typename R = std::invoke_result_t<F, parse_t<P1>, parse_t<P2>>>
constexpr auto combine(P1, P2, F);
```

```
combine :: Parser a -> Parser b -> (a -> b -> c) -> Parser c
```

Look familiar? This is `fmap2` (with flipped arguments).

PARSING WITH APPLICATIVE

Back in 2017, I used `combine` mostly in two flavours:

```
template <parser P1, parser P2>
constexpr auto operator<(P1 p1, P2 p2) {
  return combine(p1, p2, [] (auto, auto rhs) { return rhs; });
}

template <parser P1, parser P2>
constexpr auto operator>(P1 p1, P2 p2) {
  return combine(p1, p2, [] (auto lhs, auto) { return lhs; });
}
```

PARSING ALL THE THINGS

For instance:

```
// a string parser
return quote_parser < str_parser > quote_parser;
```

Where `quote_parser` parses a single quote and `str_parser` parses a string of characters that aren't quotes.

I didn't make this technique up for the talk - guess where I got it?

PARSING ALL THE THINGS

```
(<*) :: Applicative f => f a -> f b -> f a
(*>) :: Applicative f => f a -> f b -> f b
```

Handy combinators from Haskell's Control.Applicative library - they are variations on the fully-generalized `(<*>)` that just "keep the thing that the arrow points to".

In my case, I didn't have a general choice of operators, so I went with `operator<` and `operator>` and I chose the semantic of "keep the larger thing".

COMPARISON WITH MONADIC PARSING

Here's how a typical parse looks when we use monads:

```haskell
-- parse a 2-digit hex number 0xab
parseHexNum :: Parser Int
parseHexNum = do
  parseStr "0x"
  high_nybble <- parseHexDigit
  low_nybble <- parseHexDigit
  return $ hexify high_nybble low_nybble

parseHexDigit :: Parser Char
hexify :: Char -> Char -> Int
```

…procedural. (It looks horrible in C++.)

COMPARISON WITH MONADIC PARSING

And using applicative:

```
parseHexNum :: Parser Int
parseHexNum =
  hexify
  <$> (parseStr "0x" *> parseHexDigit)
  <*> parseHexDigit
```

…declarative.

(This example paraphrased from *Real World Haskell* Ch.16 pp.396-7, published in November 2008.)

PARSING

Almost always*, applicative is all you need.

You only need a monad if you need to make a decision on which parser to run based on what was parsed before.

In other words, if you have context-sensitive parsing.

Remember: monads are *inherently serial*.

PARSING

Almost always*, applicative is all you need.

You only need a monad if you need to make a decision on which parser to run based on what was parsed before.

In other words, if you have context-sensitive parsing.

Remember: monads are *inherently serial*.

* Yes I know I'm talking to the wrong crowd. Hello C++ grammar.

ASYNCHRONOUS EXECUTION

Let's talk about senders and receivers for a minute.

I'm pretty sure the authors of P2300 made a conscious decision to excise any functional terms from their vocabulary while explaining the continuation monad…

SPOT THE FUNCTIONAL PATTERN

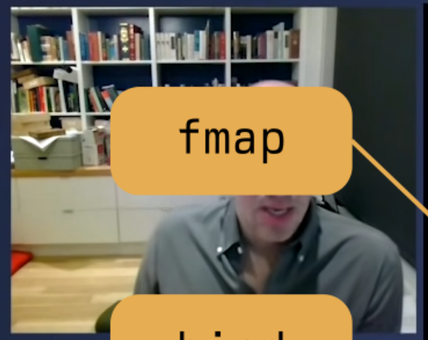
Eric Niebler

SENDER ADAPTORS OF STD::EXECUTION

Sender adaptors	Returns a sender that…
`then(sender, fn) → sender`	… transforms the result value of the operation with a function.
`upon_[error|done](sender, fn) → sender`	… transforms the error and done signals with a function.
`let_[value|…](sender, fn) → sender`	… passes the result of input sender to function, which returns new sender.
`when_all(senders...) → sender`	… completes when all the input senders complete.
`on(scheduler, sender) → sender`	… starts the input sender in the context of the input scheduler.
`into_variant(sender) → sender`	… packages all possible results of input sender into a variant of tuples.
`bulk(sender, size, fn) → sender`	… launches a bulk operation.
`split(sender) → sender`	… permits multiple receivers to be connected (forks the execution graph).
`done_as_optional(sender) → sender`	… commutes the done signal into a nullopt.
`done_as_error(sender, error) → sender`	… commutes the done signal into an error.

18

SPOT THE FUNCTIONAL PATTERN



fmap

bind

(<*>)

SENDER ADAPTORS OF STD::EXECUTION

Sender adaptors	Returns a sender that…
`then(sender, fn) → sender`	… transforms the result value of the operation with a function.
`upon_[error|done](sender, fn) → sender`	… transforms the error and done signals with a function.
`let_[value|…](sender, fn) → sender`	… passes the result of input sender to function, which returns new sender.
`when_all(senders...) → sender`	… completes when all the input senders complete.
`on(scheduler, sender) → sender`	… starts the input sender in the context of the input scheduler.
`into_variant(sender) → sender`	… packages all possible results of input sender into a variant of tuples.
`bulk(sender, size, fn) → sender`	… launches a bulk operation.
`split(sender) → sender`	… permits multiple receivers to be connected (forks the execution graph).
`done_as_optional(sender) → sender`	… commutes the done signal into a nullopt.
`done_as_error(sender, error) → sender`	… commutes the done signal into an error.

P2300

P2300 name	What it does	Functional pattern
`then`	transforms the value with a function	`fmap`
`let_value`	passes the result to a function that returns a sender	`bind`
`when_all`	completes when all the input senders complete	`(<*>)`
`just`	sends the provided values	`pure`

P2300

This is how we know (in part) that P2300 is a sound design…

It's not a novel thing dreamt up by architecture astronauts. It's based on solid functional patterns that have been known for decades.

So we know it has the expressive power to solve problems.

P2300: APPLICATIVE VS MONAD

More than any other example, P2300 highlights how applicative (`when_all`) is fundamentally parallel, while monad (`let_value`) is fundamentally serial.

Because P2300 is all about potentially concurrent execution.

P2300: APPLICATIVE VS MONAD

The characteristic monadic function `let_value` has a data dependency on its argument.
It's for when you need choice about which piece of work
to kick off based on what went before.

The `fmap`-generalization `when_all` can clearly run all its input senders in parallel, calling the (multi-argument) function when all arguments are ready.

Notice: `when_all` *could* be implemented with `let_value` by running the inputs serially and carrying values through monadically; that would *obviously be wrong* in the context of a concurrent execution framework.

CONCLUSION

"You Can't Imagine It Unless You've Been There"

That's probably about all I have time for.

CONCLUSION

Summary of applicative instances from this talk:

- `std::optional`
- `std::expected`
- ranges (homogeneous and heterogeneous)
  - zip
  - cartesian product
- parsing
- continuations

CONCLUSION

Things I didn't have time to cover:

- validation (similar to parsing)
- function application itself
  - pure is Kestrel
  - apply is Starling*
- much much more

* Conor has been nerd-sniped.

CONCLUSION

"Should we just throw the Monad class away and use Applicative instead? Of course not!

...

The moral is this: if you've got an Applicative functor, that's good;
if you've also got a Monad, that's even better!

And the dual of the moral is this: if you want a Monad, that's good;
if you only want an Applicative functor, that's even better!"

from *Applicative Programming with Effects* by Conor McBride and Ross Paterson

TAKEAWAYS

When thinking about functional architecture, think not about *how functional patterns are implemented* (usually in Haskell) but rather about *what they are for*.

Pattern	Characteristic	What it's for
functor	`fmap`	Work on values without "opening the box"
applicative	`(<*>)` (apply)	Combine multiple functor values
monad	`bind`/`join`	Make a data-dependent choice about what to do

C++ is not Haskell; Haskell formulations are not the only way and can limit our thinking.