CITADEL | Securiti

(Interested in careers?)

## Gold level sponsor

Bloomberg

Engineering

**THANK YOU**

for playing an instrumental role in making the NYC++ Meetup possible

# THANK YOU

to our sponsors…

**Bronze**

- 🙏 **Monochrome Search**
- 🙏 **Undo**

for supporting the NYC++ Meetup

CITADEL | Securiti

(Interested in careers?)

# Functional C++

Gašper Ažman

2024-04-25

# Legal Disclaimers

This is not investment advice.

Citadel Securities is my employer and enabled a lot of this research.

None of this code or text is endorsed by Citadel Securities

These words are my own.

## Calling Card

- C++ since 2000
- MSc in Graph Theory
- A9.com Search Infrastructure
- Citadel, then CitSec

## Calling Card: C++

- 2016: British Standards Institute and WG21
- Now: co-chair of SG4: Networking

# C++20

- `using enum`
- `operator<=>` fixes

## C++23

Deducing *this

```
constexpr auto fibonacci = [](this auto&& self, int x) {
    return x <= 1 ? 1 : x + self(x-1);
};
```

# C++26

- P2900 Contracts
- P2830 `constexpr` type sorting
- P2825 `declcall`
- P2926 Expression Aliases
- P2300 `std::execution` implementation
- P2989 Universal Template Parameters
- P2841 Concept and variable-template template-parameters

# This is slide code

## Omitted

- `noexcept` propagation

## Omitted

- `noexcept` propagation
- SFINAE-correctness

## Omitted

- `noexcept` propagation
- SFINAE-correctness
- nicer errors

## Omitted

- `noexcept` propagation
- SFINAE-correctness
- nicer errors
- (almost) all of the metaprogramming.

## Ommitted (cont)

- compile-time optimizations

## Ommitted (cont)

- compile-time optimizations
- lots of niceness

## Ommitted (cont)

- compile-time optimizations
- lots of niceness
- automatic inference

## Ommitted (cont)

- compile-time optimizations
- lots of niceness
- automatic inference
- and tons of other stuff.

C++ is not made for this. Yet.

I have proposals in flight that will make this better.

The errors are bad.

The compile times are worse.

(but getting better)

The runtime performance is actually pretty great.

When in doubt, do the simple thing.

Functional approaches pay off in the large.

## The Library

All the examples in this presentation actually run!

https://github.com/libfn/functional

By: **Bronek Kozicki** and **Alex Kremer**.

Theory: **Bartosz Milewski** and yours truly.

Under the ISC open-source license.

## Optimize for correctness

Let the compiler prove our code.

Only plausible business logic should compile.

Tests are CRCs for our proofs.

The typesystem *is* a proof system.

Our brain does not fit a lot, and proofs are hard.

We break down problems;

*but do we break down solutions*?

Context makes proofs hard.

# Context of a line of code

## Explicit context:

- local variables
- function arguments
- your object has state
- … and it changes!

## Block context:

- loop condition (true in loop)
- loop postcondition (true after loop)
  - [inverse of loop condition!]
- current enclosing if-conditions
- negations of previous if-conditions (we're in the else block)
- catch (type)
- are we processing a sequence? (ranged for-loops)

## Implicit context:

- global variables
- threadlocal variables
- syscall-accessible stuff
  - environment variables
  - filesystem
  - gid/uid
  - ...

## Implicit context (cont):

- current exception (this is an implicit argument if you're in a Lippincott function!)
- which exceptions *may* be thrown by subcalls
- cursors of open files
- open sockets / connections
- shared memory regions

## Implicit context (cont 2):

- dispatched async operations waiting to complete
  - how do I cancel the current operation? Is it even cancellable?
- operating system limits
- global subsystems (singletons)
  - logging
  - execution resources
  - memory allocator
  - shared network connections

## Implicit context (cont 3):

- state machines: which *state* you're in?
- What execution context is running the code?
  - high priority thread
  - main thread
  - low priority thread
  - top-half of a signal handler
  - GPU, NUMA node, IO thread, compute thread, which threadpool?????

# Example: procedural code

```
auto dumb_depth_first_search(Node* start_node, auto&& op) {
    std::vector todo{start_node}; std::set seen{start_node};
    while (!todo.empty()) {                 // post:!empty()
        auto const node = todo.back(); // pre: !empty()
        todo.pop_back();                    // pre: !empty()
        if (op(node)) { return node; } // post: !op(node)
        for (auto const& child : node->children()) {
            if (seen.contains(&child)) { continue; }
            todo.push_back(&child); // pre: child not seen
            seen.insert(&child);
        }
    } // post: todo.empty(), !op(node) for reachable nodes
    return nullptr; // not found
}
```

For every Context grows a Monad.

-- Teta Pehta

("For every disease there grows a flower")

Monad is meant loosely - I mean a compositional context.

Functor, Applicative, Monad, and some other things are all valid here.

*Technically* we also need indexed monads and graded monads.

We will not cover the theory.

## Limiting context

Most context is irrelevant at any one time.

*Which* part is up to convention and discipline.

Procedural languages have problems encoding proofs about context.

In functional languages, all context is explicit.

This helps proofs.

Can we be disciplined in c++?

## Tools

We will need tools.

Banish all context-accessors from business logic into named contexts.

And we will explicitly give access.

And we will be free.

Apprentice: But master, my function has 43 parameters?

Master: you were blind, but now you see.

Zen of python: flat is better than nested.

Have you considered flattening out your dependencies?

Design for laziness: Make the right things painful.

This is *good*. It allows more of the program to be context-free.

Push control flow, mutation, and I/O towards `main()`.

## Quick example

Given:

```
template <typename T, typename... Ts>
concept any_of =
    (... || std::same_as<std::remove_cvref_t<T>, Ts>);
```

C++ has exceptions, and we want to catch any of them:

```cpp
try { g(2); /*throws... what?*/ }
catch (any_of<E1, E2, E3> auto&& exc) {
    /* no you don't says C++ */
} // did I get all of them?
```

You need to know what g throws.

Need polymorphic runtime matching. Need generic matching.

C++ has exceptions, and we want to catch any of them:

```
try { g(2); /*throws... what?*/ }
catch (any_of<E1, E2, E3> auto&& exc) {
    /* no you don't says C++ */
} // did I get all of them?
```

You need to know what g throws.

Need polymorphic runtime matching. Need generic matching.

C++ has exceptions, and we want to catch any of them:

```
try { g(2); /*throws... what?*/ }
catch (any_of<E1, E2, E3> auto&& exc) {
    /* no you don't says C++ */
} // did I get all of them?
```

You need to know what g throws.

Need polymorphic runtime matching. Need generic matching.

## This is better:

```
// auto g(int i) -> expected<int, sum<E1, E2, E3>>;
auto r = g(2).transform_error(
    [](any_of<E1, E2, E3> auto&&) {
        return E2{};
    });
```

## This is better:

```cpp
// auto g(int i) -> expected<int, sum<E1, E2, E3>>;
auto r = g(2).transform_error(
    [](any_of<E1, E2, E3> auto&&) {
        return E2{};
    });
```

What's the type of r?

## This is better:

```
// auto g(int i) -> expected<int, sum<E1, E2, E3>>;
auto r = g(2).transform_error(
    [](any_of<E1, E2, E3> auto&&) {
        return E2{};
    });
```

What's the type of r?

`expected<int, sum<E2>>`

## You don't handle every exception? Fail to compile.

```
g(2).transform_error(
    [](any_of<E1, E3> auto&&) {
        return E2{};
    });
// ERROR: lambda not callable with E2
```

## You want to let some through? Do it explicitly:

```
auto r = g(2).transform_error(
    fn::overload{
        [](E1 const&) { return E2{}; },
        fn::identity /* id-on-E2 and E3 */
    }
)
```

## You want to let some through? Do it explicitly:

```
auto r = g(2).transform_error(
    fn::overload{
        [](E1 const&) { return E2{}; },
        fn::identity /* id-on-E2 and E3 */
    }
)
```

What's the type of r?

## You want to let some through? Do it explicitly:

```
auto r = g(2).transform_error(
    fn::overload{
        [](E1 const&) { return E2{}; },
        fn::identity /* id-on-E2 and E3 */
    }
)
```

What's the type of r?

`expected<int, sum<E2, E3>>`

## "basic" needed language features

## "basic" needed language features

- functors and closures

## "basic" needed language features

- functors and closures
- templates

## "basic" needed language features

- functors and closures
- templates
- function overloading

## "basic" needed language features

- functors and closures
- templates
- function overloading
- `operator|`

## "basic" needed language features

- functors and closures
- templates
- function overloading
- `operator|`
- `operator&`

## "basic" needed language features

- functors and closures
- templates
- function overloading
- `operator|`
- `operator&`
- CTAD - Constructor Template Argument Deduction

## "basic" needed language features

- functors and closures
- templates
- function overloading
- `operator|`
- `operator&`
- CTAD - Constructor Template Argument Deduction
- concepts

# Classic monads

- Maybe: `fn::optional`
- Error: `fn::expected`
- List: `std::ranges`
- Async: `std::execution`
- I/O: `std::execution` IO-resources
- State: what do you think OOP is?
- Environment: `std::execution` has `read()` and `write()`

## The final guideline

Name common patterns and control flow.

# `optional` - **the Maybe monad**

Pattern 1: (`optional` + `or_else`):

*Try multiple things and return the first one that succeeded*.

## Procedural way of implementing get_hostname:

```
auto is_hostname_in_args(int, char const* const*) -> bool;
auto get_hostname_from_args(int, char const* const*) -> char const*;

auto get_hostname(int argc, char const* const* argv,
                  std::string default_hostname)
        -> std::string {
    // split query / getter
    if (is_hostname_in_args(argc, argv)) {
        // perhaps... might use optional here too?
        return get_hostname_from_args(argc, argv);
    }
    // ad-hoc Maybe
    if (char const* maybe_host = getenv("SERVICE_HOSTNAME");
        (maybe_host != nullptr) || (*maybe_host != '\0')) {
        return maybe_host;
    }
    return default_hostname;
}
```

## Functional rewrite

First, adapt to uniformity.

## Before: query/getter pair

```
auto is_hostname_in_args(int, char const* const*) -> bool;
auto get_hostname_from_args(int, char const* const*) -> char const*;
```

## After: `optional`-returning functor.

```
constexpr auto maybe_hostname_from_args =
    [](int argc, char const* const* argv)
        -> std::optional<std::string> {
    if (is_hostname_in_args(argc, argv)) {
        return get_hostname_from_args(argc, argv);
    }
    return std::nullopt;
};
```

## Before: nullptr-on-nothing

```
auto std::getenv(char const*) -> char const*;
```

## After: nullopt-on-nothing

```
constexpr auto get_env = [](std::string const& varname)
        -> optional<std::string> {
    if (char const* value = std::getenv(varname.c_str());
            value != nullptr) {
        return optional(std::string(value));
    }
    return {std::nullopt};
};
```

## Tool: `filter`

For the "nonempty" bit, we'll want a `filter`:

```
constexpr auto filter = [](auto predicate) {
    return [=]<class T>(T&& value) -> optional<std::remove_cvref_t<T>>
        if (predicate(value)) {
            return value;
        }
        return std::nullopt;
    };
};
```

## Tool: filter

For the "nonempty" bit, we'll want a filter:

```
constexpr auto filter = [](auto predicate) {
    return [=]<class T>(T&& value) -> optional<std::remove_cvref_t<T>>
        if (predicate(value)) {
            return value;
        }
        return std::nullopt;
    };
};
```

## First pass: put it together:

```
constexpr auto nonempty =
    [](auto const& s){return !s.empty();};

auto get_hostname(
            int argc, char const* const* argv,
            std::string const& default_hostname)
        -> std::string {
    return maybe_hostname_from_args(argc, argv)
          .or_else([]{
            return get_env("SERVICE_HOSTNAME")
                   .and_then(filter(nonempty));
           })
          // can add other ways
          .value_or(auto(default_hostname));
}
```

Blech.

## First pass: put it together:

```
constexpr auto nonempty =
    [](auto const& s){return !s.empty();};

auto get_hostname(
            int argc, char const* const* argv,
            std::string const& default_hostname)
        -> std::string {
    return maybe_hostname_from_args(argc, argv)
          .or_else([]{
            return get_env("SERVICE_HOSTNAME")
                   .and_then(filter(nonempty));
           })
          // can add other ways
          .value_or(auto(default_hostname));
}
```

Blech.

## We defined filter wrong. Let's fix it.

```
constexpr struct filter_t {
  template <typename P> struct closure { P pred; };

  template <typename P>
  constexpr auto operator()(P&& predicate) const {
      return closure{std::forward<P>(predicate)};
  }

  template <fn::some_optional Opt, typename P>
  friend constexpr auto operator|(Opt&& opt, closure<P> const& cl) {
      return std::forward<Opt>(opt).and_then(
          [&](auto&& v) -> std::remove_cvref_t<Opt> {
              if (cl.pred(v)) {
                  return v;
              }
              return std::nullopt;
          });
  }
} filter;
```

## We defined filter wrong. Let's fix it.

```cpp
constexpr struct filter_t {
  template <typename P> struct closure { P pred; };

  template <typename P>
  constexpr auto operator()(P&& predicate) const {
      return closure{std::forward<P>(predicate)};
  }

  template <fn::some_optional Opt, typename P>
  friend constexpr auto operator|(Opt&& opt, closure<P> const& cl) {
      return std::forward<Opt>(opt).and_then(
          [&](auto&& v) -> std::remove_cvref_t<Opt> {
              if (cl.pred(v)) {
                  return v;
              }
              return std::nullopt;
          });
  }
} filter;
```

## We defined filter wrong. Let's fix it.

```
constexpr struct filter_t {
  template <typename P> struct closure { P pred; };

  template <typename P>
  constexpr auto operator()(P&& predicate) const {
      return closure{std::forward<P>(predicate)};
  }

  template <fn::some_optional Opt, typename P>
  friend constexpr auto operator|(Opt&& opt, closure<P> const& cl) {
      return std::forward<Opt>(opt).and_then(
          [&](auto&& v) -> std::remove_cvref_t<Opt> {
              if (cl.pred(v)) {
                  return v;
              }
              return std::nullopt;
          });
  }
} filter;
```

## We defined filter wrong. Let's fix it.

```cpp
constexpr struct filter_t {
  template <typename P> struct closure { P pred; };

  template <typename P>
  constexpr auto operator()(P&& predicate) const {
      return closure{std::forward<P>(predicate)};
  }

  template <fn::some_optional Opt, typename P>
  friend constexpr auto operator|(Opt&& opt, closure<P> const& cl) {
      return std::forward<Opt>(opt).and_then(
          [&](auto&& v) -> std::remove_cvref_t<Opt> {
              if (cl.pred(v)) {
                  return v;
              }
              return std::nullopt;
          });
  }
} filter;
```

## We can omit the and_then:

```
auto get_hostname(
            int argc, char const* const* argv,
            std::string const& default_hostname)
        -> std::string {
    return maybe_hostname_from_args(argc, argv)
          .or_else([]{ return get_env("SERVICE_HOSTNAME")
                                  | filter(nonempty); })
          .value_or(auto(default_hostname));
}
```

## Aside: `some_optional`

We needed a concept for the above:

```
// fastest-to-compile is_specialization_of I know of
template <typename X>
constexpr bool _some_optional_v = false;
template <typename T>
constexpr bool _some_optional_v<fn::optional<T>&>
    = true;
template <typename T>
constexpr bool _some_optional_v<fn::optional<T> const&>
    = true;
// same for std::optional

template <typename X>
concept some_optional = _some_optional_v<X&>;
```

`fn::some_optional` of course already exists; but you will need this for your own types.

Everything called `some_XXX` is a concept like the above.

# Fix the defaulting.

If you don't need to make a choice, why make it?

Just let the caller do it:

```
auto maybe_hostname_from_outside(int argc, char const* const* argv)
        -> std::string {
    return maybe_hostname_from_args(argc, argv)
          .or_else([]{ return get_env("SERVICE_HOSTNAME")
                            | filter(nonempty); });
          // <-- no more default
}
```

## Usage:

```
int main(int argc, char** argv) {
    auto const config_file =
        maybe_config_file_from_params(argc, argv)
       .value_or({});
    auto const target_hostname =
        maybe_hostname_from_params(argc, argv)
       .or_else([&]{return config_file.maybe_get_hostname();})
       .value_or("default_hostname");
}
```

This isn't always true - you might choose to do it in one or the other location depending on the desired semantics.

## Packs

Multiple monadic arguments work as packs!

```
struct config {
  template <typename T>
  auto get_maybe(std::string const& key) -> fn::optional<T>;
};
struct socket_addr { std::string hostname; int port; };
auto connect(config c) -> fn::optional<socket_addr> {
    return
        (c.get_maybe<std::string>("hostname") & c.get_maybe<int>("port"))
        // optional<pack<std::string, int>>
        .transform([](std::string && hostname, int port){
            return socket_addr{std::move(hostname), port};
        });
}
```

## Creating objects is common.

## We name common patterns.

```
template <typename T>
constexpr auto make = []<typename... Ts>(Ts&&... args) {
    return T(std::forward<Ts>(args)...);
};
auto connect(config c) -> std::optional<socket_addr> {
    return (c.get_maybe("hostname")
           & c.get_maybe("port"))
        .transform(make<socket_addr>);
    //         ^^^^^^^^^^^^^^^^^
}
```

## Function "lifts" on `optional<T>`:

```
and_then  :: (T  -> opt<U>) -> (opt<T> -> opt<U>);
transform :: (T  ->     U)  -> (opt<T> -> opt<U>);
or_else   :: (() -> opt<T>) -> (opt<T> -> opt<T>);
value_or  ::  T             -> (opt<T> -> T);
```

`value_or` is an escape operation - final default.

## **Error:** `expected<T, E>`

Instead of `nullopt`, we get E.

The flavor is different. Error is error, maybe is maybe.

There are as many `expected`s as there are Es.

# Function lifts on `expected<T, E>`

```
and_then        :: (T -> exp<U, E>) -> (exp<T, E> -> exp<U, E >);
transform       :: (T ->     U    ) -> (exp<T, E> -> exp<U, E >);
or_else         :: (E -> exp<T, E>) -> (exp<T, E> -> exp<T, E >);
transform_error :: (E ->        E') -> (exp<T, E> -> exp<T, E'>);
value_or        ::  T                -> (exp<T, E> -> T);
```

- ++ `transform_error`
- `or_else` now takes the error type.
- `and_then` cannot change E.
- Neither can `or_else`.

This means we actually got an `optional`-per-E.

`expected` is a domain-based composition type.

It is *not* better exceptions.

# Parsers: a good fit for `expected`

Compositional parser infrastructure.

Every parser's signature is

```
struct ParseError { std::span<char const> where; };
auto subgrammar(std::span<char const>)
     -> std::expected<
            pair<SomeGrammarNode, std::span<char const>>,
            ParseError
        >;
```

## For instance, parse_int

```
template <typename T>
using parser_for = fn::expected<std::pair<T, std::span<char const>>, Par
auto parse_int(std::span<char const> src)
        -> parser_for<int> {
    if (src.empty()) { return unexpected(ParseError(src)); }
    bool negative = src[0] == '-';
    if (negative || src[0] == '+') { src = src.subspan(1); }
    if (src.empty()) { return unexpected(ParseError(src)); }
    int result = 0;
    if ('0' <= src[0] && '9' <= src[0]) {
        result = src[0] - '0'; src = src.subspan(1);
    }
    while (!src.empty() && '0' <= src[0] && '9' <= src[0]) {
        // not checking for overflow
        result = result * 10 + (src[0] - '0'); src = src.subspan(1);
    }
    return std::pair{result * (-1 * negative), src};
}
```

EEEEwww.

## We could rework this to reduce the number of visible branches:

```
auto parse_digit(std::span<char const> src) -> parser_for<int> {
  return (!src.empty() && '0' <= src[0] && '9' <= src[0])
         ? parser_for<int>(std::pair{src[0] - '0', src.subspan(1)});
         : unexpected(ParseError(src));
}

auto parse_positive_digit(std::span<char const> src) -> parser_for<int>
    return parse_digit(src).and_then([&](auto r) {
        return r.first > 0 ? parser_for<int>(r)
                           : unexpected(ParseError(src));
    });
}
```

## Rework the "integral" portion...

```
auto parse_unprefixed_int(std::span<char const> src)
    -> fn::expected<std::pair<int, std::span<char const>>, ParseError> {
    return parse_positive_digit(src)
           .transform([&](auto p){
        auto&& [result, rest] = p;
        while (auto d = parse_digit(rest)) {
            // not checking for overflow
            result = result * 10 + d.value().first;
            rest = d.value().second;
        }
        return std::pair{result, rest};
    });
}
```

## The sign portion...

```cpp
auto parse_maybe_sign(std::span<char const> src) -> parser_for<int> {
    if (src.empty()) { return unexpected(ParseError(src)); }
    bool negative = src[0] == '-';
    if (negative || src[0] == '+') {
        return std::pair{negative:-1:1, src.subspan(1)};
    }
    return parse_digit(src) // just check it's a digit
           .transform([&](auto&&){return std::pair{1, src};});
}
```

## And finally put it all together:

```
auto parse_int(std::span<char const> src) -> parser_for<int> {
    return parse_maybe_sign(src)
        .and_then([](auto sign) {
            return parse_unprefixed_int(sign.second)
                .transform([&](auto p){
                    return std::pair{sign.first * p.first, p.second};
                });
        });
}
```

Seems limited.

It is.

A graded `expected` is better - we'll see it later.

# Error Packs

```
auto parse_response(json const& doc)
    -> fn::expected<Response, ParseError>
{
    return (
        (doc.get_maybe("version")
            | or_else([]{return ParseError("version is required");})
            | parse_version
            | filter(eq(version{3, 14}), make<ParseError>)
        ) // expected<Version, ParseError>
        & (doc.get_maybe("id")
            | or_else([]{return ParseError("id is required");})
            | parse_int
            | transform(make<Id>)
        ) // expected<Id, ParseError>
        & /*...*/
        ) // expected-pack<Version, Id, ..., ParseError>
        | transform(make<Response>);
}
```

- We didn't forget error checking.
- No way to make a `Response` on failure.
- We validate everything due to strong types.

## Strong types

- Version is just an integer
- constructor validates
- Response takes a Version
    - We can't make a `Response` without a `Version`

Only plausible business logic should compile.

Encode things in types until that is true.

```
Price x, y;
x + y; // does not compile
PriceDelta d;
Price r = x + d; // compiles
```

`Price` is a point-space over the integral module of `PriceDelta`.

`std::chrono` does this too.

## Dealing with multiple Error Types

We're going to need a better `variant`. We'll call it `sum`.

If `sum` is a compositional context, what is the core lift?

```
transform  :: {(Ti -> Ui)...} -> (sum<Ti...> -> sum<Ui...>);
```

## Example: a state machine:

```
using State = sum<Initialized, Connecting,
                  Connected, Disconnected, Fail>;
using Message = sum<Connect, Stop, Data, UnexpectedDisconnect>;
template <typename X>
concept a_live_state = std::same_as<X, Initialized>
                    || std::same_as<X, Connecting>
                    || std::same_as<X, Connected>;
```

```cpp
auto transition(State s, Message m) {
    return (s & m).transform(fn::overload{ // double dispatch!
        [](Initialized s, Connect c)
            -> sum<Connecting, Fail> {...},
        [](Connected auto s, Data d)
            -> sum<Connected, Fail> {...},
        [](a_live_state auto s, Stop)
            -> Disconnected {...},
        [](a_live_state auto s, UnexpectedDisconnect)
            -> sum<Connecting, Fail> {...},
        [](auto s, auto m)
            -> Fail { return sum<Fail>{}; },
    }); // -> <Connecting, Fail, Connected, Disconnected>
}
```

`transform` unions result types.

What's the type of `(s & m)`?

What's the type of `(s & m)`?

it's `pack<State, Message>`

## Making returning and calling symmetric

`sum` is special:

`transform` finds the common return type (it's a sum)

```
sum<A, B, C>{b} | transform(overload{
    [](B) {...} -> sum<E, F>
    [](A) {...} -> sum<pack<C, D>, E>,
    [](C) {...} -> G,
}); // sum<E, F, G, pack<C, D>>
```

## pack **unpacks into arguments**

```
pack{1, A{}, B{}} | transform([](int, A, B) {});
```

## pack of sums does multiple dispatch

```
pack<sum<A, B>, sum<X, Y>>{A{}, B{}}
| transform{overload{
    [](A, X) {...},
    [](A, Y) {...},
    [](B, X) {...},
    [](B, Y) {...}
}}:
```

... and it also finds the common type.

## The algebra

For ONE T:

```
sum<T> = pack<T> = sum<pack<T>> = pack<sum<T>>
```

## For multiple:

```
pack<sum<A, B>, sum<X, Y>>
=
sum<pack<A, X>,
    pack<A, Y>,
    pack<B, X>,
    pack<B, Y>>
```

(works for more than one `sum` in a pack)

## Flattening

`sum<sum<T, U>>` -> `sum<T, U>`

`pack<pack<T, U>, V>` -> `pack<T, U, V>`.

These equivalences let us define calling overload sets.

An overload set is a function defined "in parts".

Overload set calls don't compose; you can call it once, but you can't call one with a return value (you only get one type).

This makes pipelines not work.

```
transition(current_state, message) -> pack<state, effect> {...};
pack{current_state, message}
   | transform([&](auto s, auto effect) {
    environment.dispatch(effect);
    return fn::pack{s, environment.read_next_message()};
   })
   | transform(transition) ...;
```

## Back to our expected

`expected<T, sum<Es...>>`

Let's try to use our error monad with this improved `sum` on the `Error` side:

```
open(path) // expected<File, sum<DoesNotExist, PermError>>
    | and_then(
        read_line // expected<std::string, IOError> - oops
    )
```

Can't.

## Need to try this again:

```
using AllErrors = sum<DoesNotExist, PermError, IOError>;
open(path) // expected<File, sum<DoesNotExist, PermError>>
    | transform_error(match(make<AllErrors>))
    // blech
    | and_then(
        read_line // expected<std::string, IOError>
        | transform_error(make<AllErrors>) // ok, now
    )
    | and_then(
        parse_version // expected<Version, SyntaxError> oh, nonono.
    )
```

We need something less annoying that accumulates error types.

# Enter graded monads

Graded monads are... relaxed.

```
auto version_or_error
    = open(path)          // fn::expected<File, sum<DoesNotExist, PermEr
    | and_then(read_line  // expected<std::string, IOError>
    ) // expected<std::string, sum<DoesNotExist, PermError, IOError>>
    | and_then(parse_version);
    // expected<Version, sum<DoesNotExist, PermError, IOError, SyntaxErr
```

## `expected<T, sum<...>>` is a closed-polymorphic equivalent to c++ exceptions.

## We can also handle the errors generically.

```
version_or_error
    | on_error( // may return void
        overload{
        [](any_of<DoesNotExist, PermError, IOError> auto&& e) {
            std::print("could not read file {}", e.filename);
        },
        [](SyntaxError const& e) {
            std::print("syntax error near column {}", e.col);
        }
    });
```

## Back to the get_maybe...

Using sum, we can drop the requirement to provide the type of get_maybe:

```
struct config {
    auto get_maybe(std::string const& key)
      -> fn::optional<fn::sum_for<std::string, int, float>>;
};
auto connect(config c) -> fn::optional<socket_addr> {
  return (c.get_maybe("hostname") & c.get_maybe("port"))
       //            ^ <std::string> no longer here
       // optional<pack<sum<...>, sum<...>>
      .and_then(
          fn::overload{
            [](std::string && hostname, int port) {
               return fn::optional(socket_addr{std::move(hostname), port
            },
            // fail if types weren't correct
            [](auto const&...) { return std::nullopt; }
```

## Conclusion

In order to name common control flow patterns, we reach for named compositional contexts, some of which are monads.

We explored the following contexts today:

- `optional` or "Maybe"
- `expected` or "Error"
- `sum`

And we touched on what it takes to make function call composition work the same way going out as going in.

Happy hacking!

CITADEL | Securiti

(Interested in careers?)

CITADEL | Securities
Build, deploy and iterate at the speed of markets.