Call

A Library that Will Change the Way You Think about Function Invocations

Target Audience

This session will be most accessible to people who already use tuples and variants in their day-to-day coding, but ***please*** *ask questions if you get lost.*

Disclaimer

This library is currently in research mode.

- Names of things are constantly changing
- Currently only tested on bleeding-edge clang
- Requires several C++17 features
  - `constexpr` lambdas
  - fold expressions
  - `inline` variables
  - `template<auto>`
  - `if constexpr`
  - `std::variant`

Disclaimer

One more disclaimer...

THIS LIBRARY WILL BLOW YOUR MIND!!!

(maybe)

Introduction

What Will This Talk Cover?

- Simplified interaction with tuple and variant
- Facilities to jump between the run-time and the compile-time world
- Future Direction

Standards Paper

- P0376r0 “A Single Generalization of std::invoke, std::apply, and std::visit”
  - Most-recent revision can be found at http://wg21.link/p0376
  - This library is an evolution of the original paper and contains important changes
  - Effort has shifted towards first making this a boost library
  - Paper will not be revised until/unless this implementation gets more usage experience
    - Any revision would be a very tiny, simple version of what will be discussed today
    - May become a language proposal rather than a library proposal

P0376 “A Single Generalization of std::invoke, std::apply, and std::visit”

What is `std::invoke`?

`std::invoke` is a higher-order function that takes an “Invocable” and N arguments, and returns the result of calling the invocable with those arguments.

Why wouldn’t you just call the function directly?

Not all “Invocables” have a function call operator (an Invocable may not be a function object).

What is an “Invocable” in C++

An “Invocable” is a function object *or* a pointer-to-member.

- If a pointer-to-member, treats the first “argument” to the function as the this pointer.

```
class player
{
 public:
  void move_to(point location);
};

std::function<void(player&, point)> fun = &player::move_to;

player hero;
fun(hero, point{2, 4});
```

P0312 “Make Pointers to Members Callable”

- Paper by Barry Revzin
- Idea has also been suggested by Peter Dimov
- Presented but not accepted in Oulu
  - Did not reach consensus

Some Common Vocabulary

<u>Product Type</u>

Description: *Contains the state of each of N types*

Example Model: `std::tuple<T...>`

Additional Details:

- A type may be specified multiple times, representing distinct fields
- A `struct` is a kind of product type

C++17 Facility: `std::apply`

Description: *Calls a function, passing elements of a tuple as individual arguments*.

```
auto args = std::make_tuple(1, '2', 3.0);

// foo(1, '2', 3.0)
std::apply(foo, args);
```

How Do Other Languages Unpack Tuples?

Python

```python
args = (1, 2, 3, 4)

// foo(1, 2, 3, 4)
foo(*args)
```

C++

```cpp
std::tuple args{1, 2, 3, 4};

// foo(1, 2, 3, 4)
std::apply(foo, args);
```

How Do Other Languages Unpack Tuples?

Python

```python
args = (2, 3, 4)

// foo(1, 2, 3, 4)
foo(1, *args)
```

C++

```cpp
std::tuple args{2, 3, 4};

// foo(1, 2, 3, 4)
std::apply([](const auto&... args) {
               return foo(1, args...);
           },
           args);
```

How Do Other Languages Unpack Tuples?

Python

```
args0 = (2, 3)
args1 = (5, 6)

// foo(1, 2, 3, 4, 5, 6)
foo(1, *args0, 4, *args1)
```

C++

```
std::tuple args0{2, 3};
std::tuple args1{5, 6};

// foo(1, 2, 3, 4, 5, 6)
std::apply(
  [&args1](auto... args0_) {
    return std::apply(
      [&args0_...](auto... args1_) {
        foo(1, args0_..., 4, args1_...);
      },
      args1);
  },
  args0);
```

`std::apply` Is Great… but Limited

Limitation: *Can only expand a single tuple, occupying the full argument list.*

```
// Existing function taking a stream and objects to output
auto output =
  [](std::ostream& os, const auto&... args) {
    (os << ... << args);
  };

// Objects to output
std::tuple args{5, 3.5, std::string("Hello")};

// output(std::cout, 5, 3.5, std::string("Hello"))
std::apply([](const auto&... args) {
             output(std::cout, args...);
           },
           args);
```

Some Common Vocabulary

Sum Type

Description: *Contains the state of exactly one of N types (alternatives)*

Example Model: `std::variant<H, T...>`

Additional Details:

- A type may be specified multiple times, representing distinct alternatives
- Also known as a discriminated union

C++17 Facility: `std::visit`

Description: *A higher-order function that takes an Invocable and N* *`std::variants`*, *and returns the result of calling the Invocable with the currently active alternative of each variant as a separate argument.*

```
std::variant<thief, bot, rope> enemy = rope();

// attack(rope())
std::visit(attack, enemy);
```

What Does `std::visit` *Actually* Do?

A `std::visit` is analogous to nested switch statements.

Example:

Return types must be the same

```
std::variant<thief, bot, rope> enemy
  = rope();

// attack(rope())
std::visit(attack, enemy);
```

```
// Simplification of std::visit implementation
switch(enemy.index()) {
 case 0:  // thief
  return attack(std::get<0>(enemy));
 case 1:  // bot
  return attack(std::get<1>(enemy));
 case 2:  // rope
  return attack(std::get<2>(enemy));
}
```

More Advanced `std::visit`

Calling a binary function where exactly one argument comes from a variant.

```
std::variant<thief, bot, rope> enemy = rope();
player hero;

// fight(hero, rope())
std::visit(
  [&hero](const auto& enemy_) { fight(hero, enemy_); },
  enemy);
```

Using `std::visit` in Everyday Code

Situation: *You need to write a generic serialize function for a `std::variant` with all serializable alternatives.*

serialize

```
template<class Archive, class... T>
void serialize(Archive& ar, const std::variant<T...>& v)
  ar << v.index();  // Serialize index of which T

  // Serialize the active alternative
  std::visit([&ar](const auto& elem) { ar << elem; }, v);

}
```

Deserializing a `std::variant`

Situation: *You need to write a* ***deserialize*** *function for a* *`std::variant`*.

deserialize

```
template<class Archive, class... T>
void deserialize(Archive& ar, std::variant<T...>& v) {
  std::size_t index;
  ar >> index;  // Deserialize the index

}
```

A useful facility is missing!

Summary: C++ Support for Algebraic Datatypes

- Type-templates for Product types
  - `std::tuple`
- Type-template for Sum types
  - `std::variant`
- Minimal facilities for interacting with them
  - `std::get`
  - `std::apply`
  - `std::visit`
  - `std::tuple_cat`

boost C++ LIBRARIES

What Is Call?

What Is Call?

- Call is… a library that allows a user to dynamically generate a function's argument list in whole or in part directly at the call-site.

What Is Call?

- Call is... “`std::apply` on crack.”

`std::apply`	`call`
<pre>// Objects to output std::tuple args{5, 3.5, std::string("Hello")}; // output(std::cout, 5, 3.5, std::string("Hello")) std::apply([](const auto&... args) { output(std::cout, args...); }, args);</pre>	<pre>// Objects to output std::tuple args{5, 3.5, std::string("Hello")}; // output(std::cout, 5, 3.5, std::string("Hello")) call(output, std::cout, unpack(args));</pre>

What Is Call?

- Call is… similar to expansion of Iterables in Python.

Python	C++
``args0 = (2, 3)``` ```args1 = (5, 6)``` ```// foo(1, 2, 3, 4, 5, 6)``` ```foo(1, *args0, 4, *args1)```	```std::tuple args{2, 3};``` ```std::tuple args{5, 6};``` ```// foo(1, '2', 3, '4', 5, 6)``` ~~call(foo, 1, unpack(args0), 4, unpack(args1));~~ ```foo(1, unpack(args0), 4, unpack(args1));```

What Is Call?

- Call is… magic.

Access a Tuple with a Run-Time Index

```
std::tuple<cat, dog> animals;

// User decides which animal to pet at run-time
std::size_t animal_to_pet;
std::cin >> animal_to_pet;

// pet(cat{})
pet(prov::access_tuple(animals, animal_to_pet));
```

What Is Call?

- Call is… `std::visit` on crack.

<table>
<tr><th><code>std::visit</code></th><th><code>call</code></th></tr>
<tr><td>

```
std::variant<rat, bat> enemy = bat{};
player hero;

// fight(hero, bat{})
std::visit(
  [&hero](const auto& enemy_) {
    fight(hero, enemy_);
  },
  enemy);
```

</td><td>

```
std::variant<rat, bat> enemy = bat{};
player hero;

// fight(hero, bat{})
fight(hero, active_alternative_of(enemy));
```

</td></tr>
</table>

What Is Call?

- Call is… *composable* magic.

Access each Active Alternative in a Tuple of Variants

```
using animal = std::variant<cat, dog>;
std::tuple<animal, animal> tup(dog{}, cat{});

// foo(dog{}, cat{})
foo(prov::unpack(tup) | prov::active_alternative_of);
```

What Is Call?

- Call is… a way to create multi-methods for closed-sets of input types.

OOP Multimethod with `call`

```
class collidable { virtual ~collidable(); };
class box : public collidable { /* ... */ };
class sphere : public collidable { /* ... */ };
class pill : public collidable { /* ... */ };

using children = type_list_t<box, sphere, pill>;

bool result = in_collision(
  prov::dynamic_cast_<children>(collidable0),
  prov::dynamic_cast_<children>(collidable1));
```

What Is Call?

- Call is… can create multimethods for `std::any`.

Multimethod with `call`

```
class box { /* ... */ };
class sphere { /* ... */ };
class pill { /* ... */ };

using kinds = type_list_t<box, sphere, pill>;

std::any collidable0 = box{};
std::any collidable1 = pill{};

bool result = in_collision(
  prov::any_cast<kinds>(collidable0),
  prov::any_cast<kinds>(collidable1));
```

What Is Call?

- Call is… an expression-template library.

Expression-Template Library	Expression Represents
Boost.Phoenix/Boost.Lambda	Function
Boost.UBlas/Eigen/Blitz++	Tensor
Boost.Spirit.Qi	Parser
Call	Argument List (Sum Type of)

What Is Call?

- Call is… a natural extension to what we can already do with function calls.

Reasoning About Function Calls

“Normal” Invocations (No Argument Transformation)

```
// foo(1, '2', 3.0)
foo(1, '2', 3.0);
```

“Normal” Argument Transformation

In C++, we can compute, or transform, values at the call-site.

Given: *“next” is a function that returns the argument plus one.*

```
// foo(2, '2', 3.0)
foo(next(1), '2', 3.0);
```

Transform Value of Argument at Call-Site

“Normal” Type-Changing Argument Transformation

In C++, we can compute, or transform, ***types*** *at the call-site.*

```
// foo(std::string("1"), '2', 3.0)
foo(boost::lexical_cast<std::string>(1), '2', 3.0);
```

Transform Type of Argument at Call-Site

Dynamic Transformations in the Call Library

With `call` we can transform a single value to multiple values.

```
std::tuple args{2, 3};
std::tuple args{5, 6};

// foo(1, '2', 3, '4', 5, 6)
foo(1, unpack(args0), 4, unpack(args1));
```

Transform to Multiple Values at Call-Site

Other Dynamic Transformations in the Call Library

With `call` we can transform a value's type to one in a possible set of types.

```
std::variant<rat, bat> enemy = bat{};
player hero;

// fight(hero, bat{})
fight(hero, active_alternative_of(enemy));
```

Transform to One Type in a Set of Possible Types at Call-Site

What's Different?

“Normal” Argument Transformations

- May transform one or more values to a different value
- May transform a type to a different type
  - *Exact type* must be known at compile-time
- May transform a single argument to a *single* argument
- Does not require library support in C++

Argument Transformations of the Call Library

- May transform one or more values to a different value
- May transform a type to a different type
  - The *set of possible types* must be known
- May transform a single argument to *many* arguments
- Requires library support in C++

Is the Call Library *Really* that Novel?

- Functionality of Call is trivial or unnecessary in dynamically-typed languages
- Many statically-typed languages have direct support for tuple unpacking
- The Clay programming language can expand variants at the call-site
- Some aspects of the library do seem genuinely novel

History of Algebraic Datatypes in C++

Interest in Algebraic Datatypes Over Time

- Boost.Tuple (2001), Boost.Optional (2002), Boost.Variant (2003)
- `std::tuple` (C++11)
- `std::variant`, `std::optional`, `std::apply`, `std::visit` (C++17)
- Proposals
    - `std::expected` (p0323 - Vicente Botet)
    - Tuple-like variadic template facilities (p0341 - Mike Spertus)
    - Language-level variants (p0095 - David Sankel)
- Library-based pattern matching (github.com/mpark/patterns - Michael Park)

My Exposure to Variant

1. Boost.Variant accepted into boost (2003)
    a. One of many amazing feats of preprocessor metaprogramming in boost
    b. Visitors had to inherit from boost::static_visitor and specify a return type explicitly
2. Sometimes needed to expand just one of N arguments from a variant
    a. Lack of language-level lambdas and base class requirement made things very verbose
    b. Even C++11 wouldn't help (no generic lambdas until C++14)
3. Created a form of visit with implicit pass-through for non-variants (~2005)
    a. Initial Syntax: visit(my_visitor, variant1, non_variant, variant2)
    b. Eventual Syntax: visit(my_visitor, +variant1, non_variant, +variant2)
    c. Also Supported: visit<void>(my_visitor, +variant1, variant_to_not_expand, +variant2)

boost
C++ LIBRARIES

Standardization of `std::variant`

- In 2014 `std::variant` proposals appear
  - Very different from existing variant implementations
  - Introduced an intrinsic empty state
  - Inconsistent semantics with the proposed `std::optional`
- Interactions on the public mailing lists were not helpful
  - Started attending meetings
  - Decided to propose the new form of visit

Proposal of Visit with Pass-Through Support

- Started writing in early 2016
    - Accidentally typo’d in an extra “…” in one of the internal lambdas of the implementation
    - Augmented the paper to directly include tuple support
        - Submitted the paper, only looking for feedback (P0376)

What Is the Status of the Call Library?

- Currently in research-mode
- Expect to propose for Boost
- Long-term goal: pick out essentials and revise p0376
    - Possibly propose a language-level solution instead of or in addition to a library-level solution

User-Facing Concepts of the Call Library

Decomposing the Call-Site

```
auto tup = std::make_tuple(1, '2', 3.0);

// output(std::cout, 1, '2', 3.0)
call(output, std::cout, prov::unpack(tup));
```

TupleLike

Algebraic Datatype Concepts

- Specializable Traits for
  - TupleLike
    - Represents product types
    - Can be modeled by `tuple`, `boost::fusion` containers and adapted structs, etc.
  - UnionLike
    - Represents sum types that might not have a coupled discriminator
    - Logically modeled by C++ unions
    - Usually appears in higher-level code with an explicit or implicit discriminator
  - VariantLike
    - Refines UnionLike by including discriminator access
    - Can be modeled by `variant`, `optional`, `expected`, `boost::gil::any_image`, etc.

Decomposing the Call-Site

```
auto tup = std::make_tuple(1, '2', 3.0);

// output(std::cout, 1, '2', 3.0)
call(output, std::cout, prov::unpack(tup));
```

ArgumentProvider

Argument Provision Concepts

ArgumentProvider

Description: *A representation of a sum type of argument lists*

Example Model: *The return value of `prov::unpack(tup)`*

Additional Details:

- Most models do not actually contain a concrete sum type
  - The return value of `prov::unpack(tup)` only *represents* a sum type of one argument list
  - The return value of `prov::active_alternative_of(v)` does *not* contain a variant

boost C++ LIBRARIES

Decomposing the Call-Site

```
auto tup = std::make_tuple(1, '2', 3.0);

// output(std::cout, 1, '2', 3.0)
call(output, std::cout, prov::unpack(tup));
```

ArgumentProviderGenerator

Argument Provision Concepts

ArgumentProviderGenerator

Description: *An Invocable that returns an ArgumentProvider*

Example Model: *The function object* `prov::active_alternative_of`

Additional Details:

- N-ary concept over the Invocable object and its parameter types
- No further semantic requirements ("concept alias" of a constrained Invocable)

Example ArgumentProviderGenerators

ArgumentProviderGenerator	Parameter Type(s)	Represented Argument List	Argument Count	Possibility Count
`prov::unpack`	`TupleLike`	Reference to each element	Element count	`1`
`prov::active_alternative_of`	`VariantLike`	Reference to the active alternative	`1`	Alternative count
`prov::access_tuple`	`TupleLike, Integral`	Reference to the n^{th} element	`1`	Element count
`prov::value_of`	`Objects...`	Value of each argument	`sizeof...(Objects)`	`1`
`prov::reference_to`	`References...`	Reference to each argument	`sizeof...(References)`	`1`

Core Argument Provision Concepts in Detail

ArgumentProvider

Description: *Representation of a sum type of argument lists that may be expanded out as a series of actual function arguments*

Example: *The returned value of `prov::unpack(some_tuple)`*

Associated Function Templates

- `provide(Self, ArgumentReceiver)`

Note: Users do not normally invoke `provide` directly (`call` does).

ArgumentReceiver

Description: *Object that can operate on an argument list along with a compile-time list of alternative argument list types*

Note: *Akin to a visitor that's also passed the list of alternative types*

Associated Function Templates

- `receive(Self, Args...)`
- `branch(Self, APTypes0, ArgumentProvider, APTypes1)`

Note: Users do not normally interact with ArgumentReceivers directly (ArgumentProviders do).

Peeking into the Internals of `call`

- First, assume that we have the following function...

```
// Invoke "fun" with the expanded arguments
template<class Fun, class Provider>
decltype(auto) call_impl(Fun&& fun, Provider&& provider);
```

Create Call Using the Aforementioned Function

- Now, assume that all `args...` model ArgumentProvider
  - No support for pass-through of “normal” arguments just yet

```
template<class Fun, class... Args>
decltype(auto) call(Fun&& fun, Args&&... args)
{
  return call_impl(std::forward<Fun>(fun),
                   prov::group(std::forward<Args>(args)...));
}
```

ArgumentProviderGenerator: `prov::group`

Description: *Takes a series of ArgumentProviders and returns an ArgumentProvider representing their concatenation*

`prov::group` Expression	**Value of Represented Arguments**
`prov::group(prov::value_of(1, '2', 3),` `            prov::value_of(4, 5, 6.0))`	`(1, '2', 3, 4, 5, 6.0)`
`prov::group(` `  prov::value_of('A', 0xB),` `  prov::unpack(std::make_tuple('C', 0xD)))`	`('A', 0xB, 'C', 0xD)`

Aside

Pop Quiz: *`prov::group` is an associative operation taking two ArgumentProviders and returning an ArgumentProvider. What else is required for the ArgumentProvider concept to form something monoid-like with `prov::group`?*

Answer: *An identity value "`I`" such that*

```
prov::group(I, a) == a
prov::group(a, I) == a
```

prov::nothing

Description: *An ArgumentProviderGenerator representing an empty argument list*

Example:

```
// foo()
call(foo, prov::nothing);
```

Common Usage: *An argument to higher-order ArgumentProviders and as a terminating case in variadic functions*

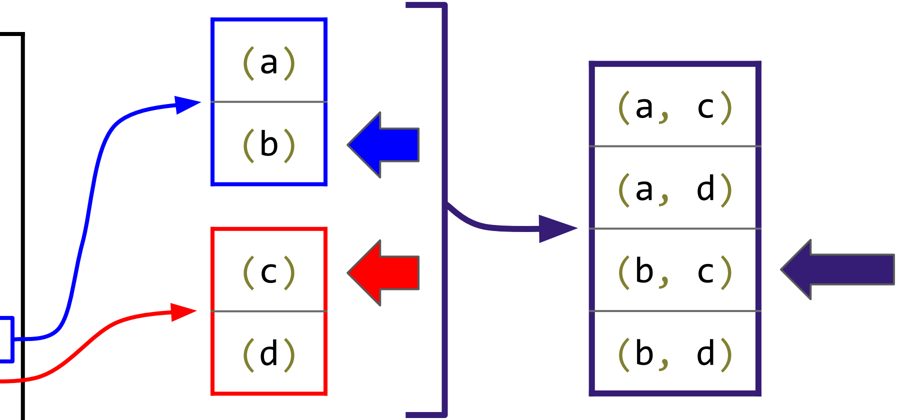
Understanding the Semantics of `prov::group`

What sum type of argument lists does the following `prov::group` represent?

```
struct a {}; struct b {};
std::variant<a, b> var = b();

struct c {}; struct d {};
std::tuple<c, d> tup;

prov::group(
  prov::active_alternative_of(var),
  prov::access_tuple(tup, 0));
```



Back to Implementing Call...

- Now, assume that all `args...` model ArgumentProvider
    - No support for pass-through of “normal” arguments just yet

```
template<class Fun, class... Args>
decltype(auto) call(Fun&& fun, Args&&... args)
{
  return call_impl(std::forward<Fun>(fun),
                   prov::group(std::forward<Args>(args)...));
}
```

How Might We Implement Pass-Through?

- Option 1:
  - Any valid parameter type is an ArgumentProvider that provides itself (unless overridden).
- Option 2:
  - Internally wrap each argument that isn't ArgumentProvider in `prov::reference_to`.

What are the benefits and drawbacks of each of the approaches?

`prov::default_to_reference_to`

Description: *An ArgumentProviderGenerator that takes any argument “arg”. If “arg” is an ArgumentProvider it returns “arg”, otherwise it returns* `prov::reference_to(arg)`*.*

How Might `call` with Pass-Through Support Look?

```
template<class Fun, class... Args>
decltype(auto) call(Fun&& fun, Args&&... args)
{
  return call_impl(std::forward<Fun>(fun),
                   ~~prov::group(prov::default_to_reference_to(std::forward<Args>(args))...));~~
                   prov::default_to_reference_to(std::forward<Args>(args)...));
}
```

boost C++ LIBRARIES

Additional ArgumentProviderGenerators

Total of 30+ ArgumentProviders and ArgumentProviderGenerators including...

- `prov::conditional(condition, provider_if_true, provider_if_false)`
- `prov::for_(state, predicate, step, provider_generator_taking_state)`

Composition of Argument Providers

Making Your Own Function Object Call-Like

CRTP Base: `call_object_base<fight>`

```
struct fight : call_object_base<fight> {
  template<class Enemy>
  static void run(fight, Enemy) { /* ... */ }
};

std::variant<rat, bat> enemy = bat{};
player hero;

// fight(hero, bat{})
fight(hero, prov::active_alternative_of(enemy));
```

Argument Transformation: `prov::transform`

Description: *Given an ArgumentProvider "pro" of arguments "args..." and an ArgumentProviderGenerator "gen", returns an ArgumentProvider of gen(args)....*

Example Usage: Unpack a tuple of tuples

```
struct a {}; struct b {}; struct c {}; struct d {};

auto tup = std::make_tuple(std::make_tuple(a(), b()),
                           std::make_tuple(c(), d()));

// foo(a(), b(), c(), d())
foo(prov::transform(prov::unpack(tup), prov::unpack));
```

Argument Transformation: `prov::transform`

Chained transformations do not need to be homogeneous

Example: Get the active alternative of each element of a tuple of variants

```
struct a {}; struct b {}; struct c {}; struct d {};

auto tup = std::make_tuple(std::variant<a, b>(a()),
                           std::variant<c, d>(d()));

// foo(a(), d())
foo(prov::transform(prov::unpack(tup),
                    prov::active_alternative_of));
```

Argument Transformation: `prov::transform`

`prov::transform` is variadic

Example: unpack a tuple of tuple of variants (because why not?)

```
struct a {}; struct b {}; struct c {}; struct d {};

auto tup = std::make_tuple(
             std::make_tuple(std::variant<a, b>(b()),
                             std::variant<c, d>(d())),
             std::make_tuple(std::variant<b, d>(b()),
                             std::variant<c, b>(c())));

// foo(b(), d(), b(), c())
foo(prov::transform(prov::unpack(tup),
                    prov::unpack,
                    prov::active_alternative_of));
```

prov::squash

Description: *Given an ArgumentProvider "a" of ArgumentProviders "provs...", returns the equivalent of* `prov::group(provs...)`.

Example:

```
struct a {}; struct b {}; struct c {}; struct d {};

auto tup0 = std::make_tuple(a{}, b{});
auto tup1 = std::make_tuple(c{}, d{});

// foo(a(), b(), c(), d())
foo(prov::squash(
      prov::value_of(
        prov::unpack(tup1),
        prov::unpack(tup2)));
```

Monad-Like Operations

Unit Operation	Map Operation	Join Operation	Bind Operation
`prov::value_of`	`prov::lift_transform`	`prov::squash`	`prov::transform`

prov::lift_call

Description: *Given an Invocable and a series of arguments and/or ArgumentProviders, returns an ArgumentProvider that provides the result of the call.*

```
class box { /* ... */ };
class sphere { /* ... */ };
class pill { /* ... */ };

using kinds = type_list_t<box, sphere, pill>;

std::any collidable0 = box{};
std::any collidable1 = pill{};

auto result_provider = prov::lift_call(
  in_collision,
  prov::any_cast<kinds>(collidable0),
  prov::any_cast<kinds>(collidable1));
```

Argument Provision: `prov::bind`

Description: *Given an Invocable "fun" and a series of arguments "args", returns the equivalent of* `prov::squash(prov::lift_call(fun, args...))`

Creating Your Own ArgumentProviders

Let’s Recreate an ArgumentProvider We’ve Used

List of some interesting ArgumentProvider’s we’ve seen so far:

- `prov::unpack`
- `prov::active_alternative_of`
- `prov::access_tuple`

prov::active_alternative_of

Description: *An ArgumentProviderGenerator that takes a VariantLike and returns an ArgumentProvider of the currently active alternative.*

Example:

```
struct triangle {}; struct square {};

std::variant<triangle, square> poly = square();

// draw(square())
draw(prov::active_alternative_of(poly));
```

Creating `prov::active_alternative_of`

What building blocks might we need?

- A way to ***access a VariantLike using a compile-time index*** represented in a function argument
- A way to ***access the value of the currently active index of a VariantLike***
- A way to transform an argument from a ***run-time index to a compile-time index***
- A way to represent an ArgumentProvider that is a ***composition*** of the transformation and the actual access

variant_traits::get

Description: *Access an element of a VariantLike type using a compile-time index.*

Standard Variant Access	Generalized VariantLike Access
`std::variant<cat, dog, rabbit> animal = cat();` `cat& cat_ = std::get<0>(animal);`	`std::variant<cat, dog, rabbit> animal = cat();` `cat& cat_ = variant_traits::get(animal, std::integral_constant<std::size_t, 0>());`

Differences from `std::get`:

- Takes the index as a `std::integral_constant` function parameter
- Explicitly intended for customization
- Is a function object (can be easily passed to higher-order functions)

boost C++ LIBRARIES

Creating `prov::active_alternative_of`

What building blocks might we need?

- A way to ***access a VariantLike using a compile-time index*** represented in a function argument
- A way to ***access the value of the currently active index of a VariantLike***
- A way to transform an argument from a ***run-time index to a compile-time index***
- A way to represent an ArgumentProvider that is a ***composition*** of the transformation and the actual access

prov::variant_index<VariantLike>

Description: *Given a VariantLike type and a run-time index, provide an instance of a `std::integral_constant` that corresponds to the index value.*

Precondition: The input value is a *valid* index for the VariantLike.

```
// Given a std::variant instantiation type "V"
using V = std::variant<cat, dog, cat>;

// and an index in the range
//   [0, std::variant_size_v<V>)
std::size_t I = 2;

// foo(std::integral_constant<std::size_t, 2>())
call(foo, prov::variant_index<V>(I));
```

Creating `prov::active_alternative_of`

What building blocks might we need?

- A way to ***access a VariantLike using a compile-time index*** represented in a function argument
- A way to ***access the value of the currently active index of a VariantLike***
- A way to transform an argument from a ***run-time index to a compile-time index***
- A way to represent an ArgumentProvider that is a ***composition*** of the transformation and the actual access

Decomposing `prov::active_alternative_of`

No Decomposition

```cpp
auto element_provider
  = prov::active_alternative_of(v);

call(fun, element_provider);
```

Some Decomposition

```cpp
auto index_provider
  = prov::variant_index<Variant>(v.index());

auto element_provider
  = prov::lift_call(
      variant_traits::get,
      prov::reference_to(v),
      index_provider);

call(fun, element_provider);
```

More Decomposition

```cpp
constexpr std::size_t size = std::variant_size_v<Variant>;

auto index_provider
  = prov::value_in_range<std::size_t, 0, size>(v.index());

auto element_provider
  = prov::lift_call(
      variant_traits::get,
      prov::reference_to(v),
      index_provider);

call(fun, element_provider);
```

Rebuilding `prov::active_alternative_of`

Some Decomposition

```
auto index_provider
  = prov::variant_index<Variant>(v.index());

auto element_provider
  = prov::lift_call(
      variant_traits::get,
      prov::reference_to(v),
      index_provider);
```

```
template<class Variant>
auto active_alternative_of(const Variant& v)
{
  return prov::lift_call(
    variant_traits::get,
    prov::reference_to(v),
    prov::variant_index<Variant>(v.index()));
}
```

We've just implemented something like `std::visit` in a single statement!

Improving `prov::active_alternative_of`

ArgumentProviderGenerators are function objects such that they can easily be used with higher-order functions, so let's make this a function object...

```
template<class Variant>
auto active_alternative_of(const Variant& v)
{
  return prov::lift_call(
    variant_traits::get,
    prov::reference_to(tup),
    prov::variant_index<Variant>(v.index()));
}
```

```
struct active_alternative_of_t {
  template<class Variant>
  auto operator ()(const Variant& v) const
  {
    return prov::lift_call(
      variant_traits::get,
      prov::reference_to(v),
      prov::variant_index<Variant>(v.index()));
  }
} inline constexpr active_alternative_of = {};
```

boost C++ LIBRARIES

Improving `prov::active_alternative_of`

Variant access should perfect-forward the element, preserving cv-qualifiers and value category.

```cpp
struct active_alternative_of_t {
  template<class Variant>
  auto operator ()(const Variant& v) const
  {
    return prov::lift_call(
      variant_traits::get,
      prov::reference_to(v),
      prov::variant_index<Variant>(v.index()));
  }
} inline constexpr active_alternative_of = {};
```

```cpp
struct active_alternative_of_t {
  template<class Variant>
  auto operator ()(Variant&& v) const
  {
    return prov::lift_call(
      variant_traits::get,
      prov::reference_to(std::forward<Variant>(v)),
      prov::variant_index<unqualify_t<Variant>>(v.index()));
  }
} inline constexpr active_alternative_of = {};
```

What Have We Learned from This Example?

- `std::visit` can be implemented as a one-line ArgumentProvider
- ArgumentProviders can be easily composed
  - `prov::lift_call` can create an ArgumentProvider from a “normal” function and arguments from one or more ArgumentProviders
  - The library provides low-level ArgumentProviders for converting from run-time values to compile-time values

Let’s Try Implementing Another

List of some interesting ArgumentProvider’s we’ve seen so far:

- `prov::unpack`
- ~~`prov::active_alternative_of`~~
- `prov::access_tuple`

Dispelling Magic: `prov::access_tuple`

Description: *An ArgumentProviderGenerator that takes a TupleLike and a run-time index as parameters, and returns an ArgumentProvider of the tuple element.*

Example:

```
std::tuple<cat, dog> tup;
std::size_t which = 1;  // Not a constant

// output(dog{})
call(output, prov::access_tuple(tup, which));
```

Creating `prov::access_tuple`

What building blocks might we need?

- A way to ***access a TupleLike using a compile-time index*** represented in a function argument
- A way to transform an argument from a ***run-time value to a compile-time index***
- A way to represent an ArgumentProvider that is a ***composition*** of the transformation and the actual access

This is very similar to what is needed for active_alternative_of!

Frankensteining `prov::access_tuple`

```
struct active_alternative_of_t {
  template<class Variant>
  auto operator ()(Variant&& v) const
  {
    return prov::lift_call(
      variant_traits::get,
      prov::reference_to(std::forward<Variant>(v)),
      prov::variant_index<unqualify_t<Variant>>(
        variant_traits::index(v)));
  }
} inline constexpr active_alternative_of = {};
```

```
struct access_tuple_t {
  template<class Tuple>
  auto operator ()(Tuple&& tup, std::size_t index) const
  {
    return prov::lift_call(
      tuple_traits::get,
      prov::reference_to(std::forward<Tuple>(tup)),
      prov::tuple_index<unqualify_t<Tuple>>(index));
  }
} inline constexpr access_tuple = {};
```

What Does `prov::access_tuple` *Actually* Do?

Description: *Creates an ArgumentProvider that represents the Nth element of the tuple.*

Example:

~~Return types must be the same~~

```
std::tuple<thief, bot, rope> enemies;
std::size_t which = 1;  // Variable

// attack(bot())
attack(std::access_tuple(enemies, which));
```

```
// Simplification of the eventual provision
switch(which) {
 case 0:  // thief
  return attack(std::get<0>(enemies));
 case 1:  // bot
  return attack(std::get<1>(enemies));
 case 2:  // rope
  return attack(std::get<2>(enemies));
}
```

Additional Features of `prov::access_tuple`

- Deals with the tuple's natural `index_type` rather than `std::size_t`
- Supports passing a `std::integral_constant` as an index
    - Prevents any run-time branching/dynamic dispatch
    - Useful when composing ArgumentProviders
- Properly constrained for valid tuples and indices
- Supports customizable fall-back for when the index is out of range
- Is `constexpr`

What Have We Learned from This Example?

- `prov::access_tuple` is not magic!
- Converting a runtime value to a compile-time value is easy
  - You may have already encountered situations where this would be useful

Let’s Try Implementing That Last One

List of some interesting ArgumentProvider’s we’ve seen so far:

- `prov::unpack`
- ~~`prov::active_alternative_of`~~
- ~~`prov::access_tuple`~~

prov::unpack

Description: *An ArgumentProviderGenerator that takes a TupleLike and returns an ArgumentProvider that represents each element as a separate argument.*

```
struct a {}; struct b {}; struct c {};

std::tuple<a, b, c> tup;

// output(a(), b(), c())
output(prov::unpack(tup));
```

Creating `prov::unpack`

What building blocks might we need?

- A way to ***access a TupleLike using a compile-time index*** represented in a function argument
- A way to ***generate all of the valid indices of a Tuple*** in the form of separate
- A way to represent an ArgumentProvider that is a ***composition*** of the transformation and the actual access

prov::tuple_indices<TupleLike>

Description: *Given TupleLike type passed as an explicit template argument, is an ArgumentProvider of all valid tuple indices each as a std::integral_constant.*

```
struct a {}; struct b {}; struct c {};
using Tup = std::tuple<a, b, c>;

// foo(std::integral_constant<std::size_t, 0>{},
       std::integral_constant<std::size_t, 1>{},
       std::integral_constant<std::size_t, 2>{})
foo(prov::tuple_indices<Tup>());
```

Composing prov::unpack

```
struct unpack_t {
  template<class Tuple>
  auto operator()(const Tuple& t) const {
    return prov::transform(
      prov::tuple_indices<Tuple>,
      [&tup](auto i) { return access_tuple(tup, i); });
  }
} inline constexpr unpack = {};
```

Alternative ArgumentProvider Creation Method

Create a type with a *concept map* (traits) to the ArgumentProvider concept

- Often produces code that is faster to compile and easier to debug
- Drawback: More code and direct interaction with ArgumentReceivers
- Recommended method for commonly-used, library-level ArgumentProviders

Solving a Motivating Case

Serializing a `std::variant`... *Take 2!*

Situation: *Write a generic serialize function for `std::variant`*

Idea: *Serialize the variant index followed by the active alternative*

serialize

```
template<class Archive, class... T>
void serialize(Archive& ar, const std::variant<T...>& v)
{
  ar << v.index();  // Serialize index

  // Serialize the active alternative
  call([&ar](const auto& elem) { ar << elem; },
       prov::active_alternative_of(v));
}
```

Deserializing a `std::variant`

Situation: *Write a generic deserialize function for `std::variant`*

Difficulty: *Which type to deserialize depends on the deserialized index*

deserialize

```
template<class Archive, class... T>
void deserialize(Archive& ar, std::variant<T...>& v) {
  using V = std::variant<T...>;
  std::size_t i;
  ar >> i;  // Deserialize the index

  // Deserialize the expected alternative
  // Assume deserialize_impl takes an integral_constant index
  call(deserialize_impl, ar, v, prov::variant_index<V>(i));
}
```

deserialize_impl

```
auto deserialize_impl
  = [](auto& ar, auto& v, auto index) {
      ar >> v.template emplace<index>();
    };
```

Deserializing a `std::variant`

Is there a safety concern here?

deserialize

```
template<class Archive, class... T>
void deserialize(Archive& ar, std::variant<T...>& v) {
  using V = std::variant<T...>;
  std::size_t i;
  ar >> i;  // Deserialize the index

  // Deserialize the expected alternative
  // Assume deserialize_impl takes an integral_constant index
  call(deserialize_impl, ar, v, prov::variant_index<V>(i));
}
```

```
// Throw if out-of-range
prov::to_variant_index<V>(
  i,
  prov::default_.fail([]{ throw an_exception(); }))
```

```
// ...or provide fall-back arguments
prov::to_variant_index<V>(
  i,
  prov::default_(prov::value_of(args, you, choose)))
```

Where There's `default_`, There's Also `case_`

Most ArgumentProviderGenerators may be created via composition. A handful of low-level ArgumentProviderGenerators are included as tools.

- `prov::value_in_range<ValueType, Begin, End>`
- `prov::value_in_set<ValueType, Values...>`
- `prov::switch_`
- *… and several others …*

prov::switch_

Description: *A mini EDSL for generating arguments through an embedded switch statement.*

```
enum class option { a, b, c };

option my_option = option::a;

auto s = prov::switch_(
  my_option,

  prov::case_<option::a>
    (prov::value_of(foo())),

  prov::case_<option::b>
    (prov::value_of(bar())),

  prov::default_(prov::nothing)
);
```

boost C++ LIBRARIES

Use of prov::default_ with Other ArgumentProviders

Having a fallback when a run-time to compile-time conversion cannot succeed is a common desire for many ArgumentProviders

- prov::value_in_range
- prov::access_tuple
- prov::access_union
- prov::access_variant

Use-Case Exploration:
Collision Detection

Situation: Collision Detection

Situation: *Application has various primitive shape types…*

triangle	circle	rectangle

Values of these types contain vertex offsets, radius, orientation, etc.

Overload set:

```
bool in_collision_(triangle, triangle);
bool in_collision_(triangle, circle);
bool in_collision_(triangle, rectangle);
bool in_collision_(circle, circle);
bool in_collision_(circle, rectangle);
bool in_collision_(rectangle, rectangle);
// ...
```

Shapes with Run-time Polymorphism

Situation: *For some of our shapes, we don't know which we'll have until run-time.*

triangle	circle	rectangle

```
std::variant<triangle, circle, rectangle>
```

Checking Collision with a Variant of Shapes

Situation: *Given a square "square_" and a variant of shapes "shape", see if the square and the active shape of "shape" are in collision.*

Single Function Object to Call

```
auto in_collision = [](auto s0, auto s1) {
  return in_collision_(s0, s1);
}
```

```
square square_ = get_square();
std::variant<triangle, circle, square> shape = get_shape();

bool res = in_collision(square_, prov::active_alternative_of(shape));
```

Situation: Detailed Collision Information

Situation: *We would like our collision functions to return more detailed information.*

```
struct triangle_triangle_collision {
  explicit operator bool() const;  // Converts to true if in collision
  // ... details specific to triangle-triangle collision
};
```

```
triangle_triangle_collision   in_collision_(triangle, triangle);
triangle_circle_collision     in_collision_(triangle, circle);
triangle_rectangle_collision  in_collision_(triangle, rectangle);
circle_circle_collision       in_collision_(circle, circle);
circle_rectangle_collision    in_collision_(circle, rectangle);
rectangle_rectangle_collision in_collision_(rectangle, rectangle);
// ...
```

Using `call` with the New Collision Functions

Situation: *We have a shape “s” and a variant of shapes “v” and need to see if “s” and the active shape of “v” are in collision.*

```
square square_ = get_square();
std::variant<triangle, circle, square> shape = get_shape();

bool res = in_collision(square_, prov::active_alternative_of(shape));
```

All possible overloads must have the same return type

Invocation When Return Types Differ

Problem: *Different overloads return different values, but we can only return an instance of a single type.*

Solution: *User-specified strategy for reducing different return types to a single return type.*

Argument Provision Concepts

ReturnValueReducer

Description: *An object capable of describing how to map a list of types to a single type “s”, and how to convert a value of one of the types in the list to a value of “s”.*

Associated Function Template:

```
template<class H, class F, class T>
decltype(auto) reduce(your_reducer, H types0, F&& fun, T types1);
```

Note: Users do not normally invoke `reduce` directly (`call` does).

How to Specify a ReturnValueReducer

Using `call` with the default ReturnValueReducer:

```
call(foo, prov::active_alternative_of(bar));
```

Using `call` with a custom ReturnValueReducer:

```
call[some_custom_reducer](foo, prov::active_alternative_of(bar));
```

Must be a model of ReturnValueReducer

The Default ReturnValueReducer

`reducer::same_type_or_fail`

Description: *Directly returns the result of the call without changing any types or values (substitution failure if not all possible return types are the same).*

Usage:

```
call(foo, prov::active_alternative_of(bar));
// ... is equivalent to ...
call[reducer::same_type_or_fail](foo, prov::active_alternative_of(bar));
```

Example ReturnValueReducers

(Default ReturnValueReducer) (call_<T> is shorthand for call[reducer::to<T>])

ReturnValueReducer	Reduced Type	Reduced Value
reducer::same_type_or_fail	Return type of overloads (all must be same)	Result
reducer::to<T>	T	Result (**void**, **bool**, and implicit conversions)

boost
C++ LIBRARIES

Example ReturnValueReducers

ReturnValueReducer	Reduced Type	Reduced Value
`reducer::same_type_or_fail`	Return type of overloads (all must be same)	Result
`reducer::to<T>`	`T`	Result (**`void`**, **`bool`**, and implicit conversions)
`reducer::transform(conversion)`	Return type of conversion (all must be same)	`conversion(`*`result`*`)`

Example ReturnValueReducers

ReturnValueReducer	Reduced Type	Reduced Value
`reducer::same_type_or_fail`	Return type of overloads (all must be same)	Result
`reducer::to<T>`	`T`	Result (`void`, `bool`, and implicit conversions)
`reducer::transform(conversion)`	Return type of conversion (all must be same)	`conversion(result)`
`reducer::to_variant`	`std::variant<return-types-of-overloads...>`	Result as the corresponding variant alternative

reducer::to_variant

Description: *Reduces N different possible overloads to be called to a* `std::variant` *of N alternatives.*

```
square square_ = get_square();
std::variant<triangle, circle, square> shape = get_shape();

// The type of “res” is a variant of all possible collision result types.
auto res = in_collision[reducer::to_variant](square_, prov::active_alternative_of(shape));
```

Example ReturnValueReducers

ReturnValueReducer	Reduced Type	Reduced Value
`reducer::same_type_or_fail`	Return type of overloads (all must be same)	Result
`reducer::to<T>`	`T`	Result (**`void`**, **`bool`**, and implicit conversions)
`reducer::transform(conversion)`	Return type of conversion (all must be same)	`conversion(`*`result`*`)`
`reducer::to_variant`	`std::variant<`*`return-types-of-overloads...`*`>`	Result as the corresponding variant alternative
`reducer::to_heterogeneous_variant`	`std::variant<`*`unique-return-types...`*`>`	Result as the corresponding variant alternative

Example ReturnValueReducers

ReturnValueReducer	Reduced Type	Reduced Value
`reducer::same_type_or_fail`	Return type of overloads (all must be same)	Result
`reducer::to<T>`	`T`	Result (**`void`**, **`bool`**, and implicit conversions)
`reducer::transform(conversion)`	Return type of conversion (all must be same)	`conversion(result)`
`reducer::to_variant`	`std::variant<return-types-of-overloads...>`	Result as the corresponding variant alternative
`reducer::to_heterogeneous_variant`	`std::variant<unique-return-types...>`	Result as the corresponding variant alternative
`reducer::provide_result_to(receiver)`	Return type of provision of result to receiver	Return value of provision of result to receiver

Recapitulation

Uses for the Call Library

- Simplified interaction with tuples and variants
  - Reduce the amount of lambdas/utility functions that need to be created
  - High-level uses make code easier to write and to read
- Development of embedded domain-specific languages
  - Ability to introspect branches
  - Libraries can be easily extended

State of the Call Library

- Project is still in research mode
- High-level and low-level details are changing all of the time
- Boost.Quickbook docs are currently in development

Future Possibilities

- Implement conditionally-noexcept clauses throughout the library
- Implement automatic continuation chaining (prov::when_ready(fut))
- Investigate making this a language feature
    - Possibly a user-overloadable `operator...`
    - May remove some captures/move operations (compose via expression aliases)
    - Can be more optimizer-friendly for small, trivial types (removes some forwarding-references)
    - Would make debugging simpler
    - Would likely improve compile-times
    - Wouldn’t need to use so many advanced templates (they scare the normal folk)

Questions