Argot

Simplifying Variants, Tuples, and Futures

Matt Calabrese
Twitter: @CppSage

C++Now 2018

My Run-Time Polymorphism Philosophy

- *variants* and *type erasure* are generally **superior** to inheritance
  - Polymorphism only when needed
  - Non-intrusive
  - Value semantics by default
  - Direct, efficient multi-method support (sum types)

Why Don't "Non-Experts" Use Sum Types in C++?

- A variant is a new concept to many existing users
- Sum types are not yet well supported
  - Few existing facilities for working with variants
  - std::visit is cumbersome
  - The language does not provide pattern matching

Though often "better" than inheritance, variants are too foreign and have too steep of a learning curve to be practical.

I Aim to Make Variants More Usable!

C++ Standard Paper

- "A Single Generalization of std::invoke, std::apply, and std::visit" [Calabrese] http://wg21.link/P0376
  - Simplifies the usage tuples and variants
  - Shelved until Argot is "complete"
  - Will likely propose as language features rather than library features

Where Can I Try It Out?

- Library is on github at https://github.com/mattcalabrese/argot/
  - Header-only (it's templates all the way down)
- Disclaimer:
  - Hobby project
  - Requires bleeding-edge clang
    - Uses most C++17 language features

Algebraic Datatypes in C++

Tuples, variants, and their associated functions

Product Types in C++

- Tuple-Like types
  - std::tuple
  - std::array
  - struct

Unpacking Product Types

- std::apply
  - Given an Invocable **f** and a tuple **t**, invokes **f** with each element of **t**.

```
template<class T, class... P>
void print(T& stream, P const&... args);

int main()
{
  std::tuple<int, float, char> tup = /*...*/;
  std::apply([](auto const&... args) { print(std::cout, args...); }, tup);
}
```

- Structured bindings

```
std::tuple<int, float, char> bar();

int main()
{
  auto&& [a, b, c] = bar();
}
```

Wouldn't It Be Nice If This Just Worked?

```
std::tuple<int, float, char> tup = /*...*/;

// Ideally, this would unpack the tuple.
print(std::cout, tup...);
```

Expand the tuple

```
std::tuple<int, float, char> tup = /*...*/;

// With a library, could we maybe do *this*?
print(std::cout, *tup);
```

Sum Types Currently in C++

- std::variant
- std::optional
- std::expected (proposed)

Standard Variant Utilies/Algorithms

- std::visit
  - Given an Invocable **f** and a variant **v**, invokes **f** with the active alternative of **v**.

```
using shape = std::variant<circle, square>;

void draw(circle const&) { std::cout << "()"; }
void draw(square const&) { std::cout << "[]"; }

void draw_shapes(std::vector<shape> const& shapes)
{
  for(shape const& s : shapes)
    std::visit([](auto const& s_) { draw(s_); }, s);
}
```

Could Variant Visitation Be Better?

- Plenty of languages offer pattern matching
- The Clay programming language offers a "dispatch" operator

```
variant Shape (Circle, Square);

[S | VariantMember?(Shape, S)]
define draw(s:S) : ;

overload draw(s:Circle) { println("()"); }
overload draw(s:Square) { println("[]"); }

drawShapes(ss:Vector[Shape]) {
    for (s in ss)
        draw(*s);
}
```

Pass the active alternative

Wouldn’t It Be Nice If This Just Worked?

```
std::variant<circle, square> shape = /*...*/;

draw(*shape);
```

Visit the variant

Why Are Tuples and Variants Easier to Use in Other Languages?

- Language-level algebraic datatypes
- Facilities for "generating" argument lists
  - Python's *iterable "generates" an argument list from elements
  - Clay's *variant "generates" an argument corresponding to the active alternative
- Community Culture
  - Functional communities tend to more fully embrace higher level concepts

How Can We Improve Working with Tuples and Variants in C++?

- Better language support
  - Language level variants and pattern matching *may* eventually come
    - "Pattern Matching and Language Variants" [Sankel] http://wg21.link/p0095
    - Michael Park Is also working on efforts in this area
  - A Clay-like dispatch operator can hypothetically be adopted (not proposed)
- Better library support
  - Can greatly improve the experience without modifying the language
  - Boost is a breeding ground for libraries in this area
    - Boost.Hana
    - Boost.Fusion

Limitations of std::visit

It's more than a syntax problem

Observation

1. std::variant<T...> can contain multiple Ts of the same type
2. std::visit forwards a reference to the currently active alternative
3. If we have duplicate types in the variant, we lose information with visit
   - A call to var.index() can tell us more, but in the form of a run-time value

Using Variants in Practice

- Situation
  - You have *N* ranges of **T** and wish to view them as though they were a single range, without actually copying the data into a container (identity matters)

```
// Two or more ranges with the same value type
auto range1 = /*...*/;
auto range2 = /*...*/;

// A view as though they were a single range
auto combined_view = concatenate(range1, range2);

// Get an iterator to the largest element.
auto max_it = max_element(combined_view);
```

Concatenated View Synopsis

```
template<class... Subranges>
class concat_t
{
 public:
  // Iterator defined in later slides
  using iterator = concat_iterator<Subranges...>;

  // ...

 private:
  std::tuple<Subranges...> subranges;
};
```

Implement the Iterator with Existing Facilities

```
template<class... Subranges>
struct concat_iterator
{
  concat_iterator& operator++()
  {
    auto increment_impl = [this](auto& curr_it)
    {
      ++curr_it; // ... And then what?

    };

    std::visit(increment_impl, underlying_it);
    return *this;
  }

  std::variant<typename Subranges::iterator...> underlying_it;
  concat_t<Subranges...>* full_range;
};
```

Summary of the Problem

- std::visit forwards a reference to the active alternative, but does not tell us the **index** of that alternative
  - v.index() gives us a **run-time** index that we can't use to index into our tuple of subranges

We Need an Additional Facility

argot::call

An algorithm for expanding symbolic argument list representations

First, What Is std::invoke?

- std::invoke(function, args...);
  - Does a superset of "function(args...)"
  - Works with generalized Invocables (pointer-to-member, etc.)

```
auto print = [](auto& stream, auto const&... args) { /*...*/ };

// Equivalent to print(std::cout, 1, 2.0, '3')
std::invoke(print, std::cout, 1, 2.0, '3');
```

What Is argot::call?

- A generalization of std::invoke that can **expand symbolic argument list placeholders** wherever they appear
  - For "simple" calls, it does exactly the same thing as std::invoke

```
auto print = [](auto& stream, auto const&... args) { /*...*/ };

// Equivalent to print(std::cout, 1, 2.0, '3')
call(print, std::cout, 1, 2.0, '3');
```

An "Unpack This Tuple" Placeholder

- prov::unpack(tup) results in a placeholder telling "call" to **expand a tuple**, wherever it may appear in the argument list

```
std::tuple<int, float, char> tup = /*...*/;

// Call print with each tuple element
call(print, std::cout, prov::unpack(tup));
```

ArgumentProvider

What *Exactly* Is argot::call Doing?

1. For each argument that is an ArgumentProvider, expand it in-place.
2. For each argument that is *not* an ArgumentProvider, perfect-forward it.

```
// Call print with each tuple element
call(print, std::cout, prov::unpack(tup));
```

ArgumentProvider

- Equivalent using std::apply

```
// Pass the tuple elements as separate arguments to print.
std::apply([](auto const&... args){ print(std::cout, args...); }, tup);
```

What Other ArgumentProviders Can Be Useful?

- Use prov::alternative_of to tell "call" to **expand a variant**
  - Equivalent to a "visit" at the individual argument level

```
std::variant<int, float, char> var = /*...*/;

// Pass the active alternative as an argument to print.
call(print, std::cout, prov::alternative_of(var));
```

ArgumentProvider

- Equivalent using std::visit

```
std::variant<int, float, char> var = /*...*/;

// Pass the active alternative as an argument to print.
std::visit([](auto const& arg){ print(std::cout, arg); }, var);
```

What Exactly Can an ArgumentProvider Represent?

- An ArgumentProvider is a type that *represents* a **sum type of possible argument lists**
  - There must be some finite set of possibilities
  - prov::unpack(tup) represents a single possible argument list of N arguments
  - prov::alternative_of(var) represents N possible argument lists, each of 1 argument

Let's Make Things Easier

- It'd be better if we could just write "print(std::cout, prov::unpack(tup))"
- argot::as_call_object(function) makes a function that works like "call"

```
auto print
  = as_call_object([](auto& stream, auto const&...) { /*...*/ });


std::tuple<int, float, char> tup = /*...*/;
std::variant<int, float, char> var = /*...*/;

// No need for callers to use higher-order functions!
print(std::cout, prov::unpack(tup), prov::alternative_of(var));
```

But Again, Wouldn't It Be Even *Nicer* if We Could Just Overload "..."?

```
std::tuple<int, float, char> tup = /*...*/;

// Ideally, this would unpack the tuple.
print(std::cout, tup...);
```

The DWIW Operator

- "Do what I want!" (aka the expansion operator)
  - Akin to a hypothetical "operator..." overload
  - Intentionally in namespace argot::expansion_operator **not found by ADL**

```
std::tuple<int, float, char> tup = /*...*/;

// Apply the "default" provision for a tuple.
print(std::cout, +tup);
```

```
std::variant<int, float, char> var = /*...*/;

// Apply the "default" provision for a variant.
print(std::cout, +var);
```

Operands that Work with +

- TupleLike types
  - Each element becomes a separate argument (prov::unpack)
- VariantLike types
  - The active alternative becomes the argument (prov::alternative_of)
- ArgumentProviders of things that work with +
  - The logical concatenation of each expanded argument list
- User-customizable (without manually overloading +)

ArgumentProvider of Expandables

- Use the + operator *N* times in a row to expand *N* levels deep
  - Convenient when dealing with nested algebraic datatypes

```
variant<tuple<int, float>, tuple<char, double>> var = /*...*/;

// Expand each tuple element in a variant of tuples.
print(std::cout, "The active tuple contains", ++var);
```

Case Study: Range Concatenation

View a series of ranges as though they were a single range

Problem Restatement

- Situation
  - You have *N* ranges of **T** and wish to view them as though they were a single range, without actually copying the data into a container.

```
// Two or more ranges with the same value type
auto range1 = /*...*/;
auto range2 = /*...*/;

// A view as though they were a single range
auto combined_view = concatenate(range1, range2);

// Get an iterator to the largest element.
auto max_it = max_element(combined_view);
```

Concatenated View Synopsis

```
template<class... Subranges>
class concat_t
{
 public:
  // Iterator defined in later slides
  using iterator = concat_iterator<Subranges...>;

  // ...

 private:
  std::tuple<Subranges...> subranges;
};
```

Where We Ended Earlier...

```
template<class... Subranges>
struct concat_iterator
{
  concat_iterator& operator++()
  {




    // We need to get the variant index as a constant!




    return *this;
  }

  std::variant<typename Subranges::iterator...> underlying_it;
  concat_t<Subranges...>* full_range;
};
```

prov::index_of(variant)

- Use prov::index_of to retrieve a **std::integral_constant variant index**

```
std::variant<int, float, int> var(std::in_place_index<2>);

// foo(std::integral_constant<std::size_t, 2>())
foo(prov::index_of(var));
```

Now We Can Proceed

```
template<class... Subranges>
struct concat_iterator
{
  concat_iterator& operator++()
  {
    auto increment_impl = [this](auto index)
    {
      ++std::get<index.value>(underlying_it);
      advance_to_next_nonempty_range<index.value>();
    };

    call(increment_impl, prov::index_of(underlying_it));

    return *this;
  }

  std::variant<typename Subranges::iterator...> underlying_it;
  concat_t<Subranges...>* full_range;
};
```

```cpp
template<class... Subranges>
struct concat_iterator
{
  //...

  template<std::size_t I>
  void advance_to_next_nonempty_range()
  {
    if constexpr(I != sizeof...(Subranges) - 1)
      if(    std::get<I>(underlying_it)
          == std::end(std::get<I>(full_range->subranges)))
      {
        constexpr std::size_t new_i = I+1;
        underlying_it.template emplace<new_i>(
          std::begin(std::get<new_i>(full_range->subranges)));

        advance_to_next_nonempty_range<new_i>();
      }
  }

  std::variant<typename Subranges::iterator...> underlying_it;
  concat_t<Subranges...>* full_range;
};
```

And Our Range Type Is Done!

Application to Concurrency

Seamlessly assembling future continuations

std::async

- Intent: *Easily execute functions asynchronously*

```
// Call "foo" with the values 1, 2, 3 and return a future
std::future<int> result = std::async(foo, 1, 2, 3);
```

Problem with std::future

- No way to specify an operation to be performed once the value is ready
  - Must **block** in order to retrieve the value of the future
  - Not a good building block for asynchronous computation graphs

Futures with Continuation Support

- boost::futures and stlab::futures support continuations using ".then"

```
stlab::future<int> fut = /*asynchronous call*/;

// Whenever "fut" is ready, call foo (non-blocking)
std::move(fut).then(exec, [](int arg) { foo(arg, 2, 3); });
```

This higher-order function looks *sort of* like std::apply or std::visit

Idea

- Can we make an ArgumentProvider that calls "print" when "some_future" is ready?

```
print(stream, "The future's underlying value is", +some_future)
```

Only if we block

What Went Wrong?

- An ArgumentProvider must **directly** produce arguments to the function
  - This would imply **blocking** (calling .get())
- The result will **not be a future**

argot::async_call

Concurrent execution with symbolic argument list representations

The Purpose of argot::async_call

- A well-behaved replacement for std::async
- An equivalent of argot::call for the asynchronous world
  - Takes an **Executor** in addition to the Invocable and its symbolic arguments
  - **Returns a future** to the result of the Invocable
  - Supports passing futures once they become ready (**non-blocking**)

… But First, argot::async_forgetful_call

- Exactly like argot::async_call, except it returns **void** instead of a future
  - Simpler, and we can efficiently build async_call with it

```
// Calls "print" through the Executor "exec".
async_forgetful_call(exec, print, std::move(stream), 1, 2, 3);
```

Executor

- An Executor controls **where/how an Invocable is run**
- Work is ongoing in the committee on **standard Executor concepts**
- Argot's Executors are most similar to the **stlab Executors**
  - Single associated function "execute" that takes a **nullary Invocable**
  - executor::immediate is an executor that executes a function synchronously
  - executor::stlab(*model-of-stlab-executor*) converts an stlab::executor to Argot

```
// Use an Executor directly (uncommon)
executor_traits::execute(
  executor::immediate, nullary-invocable);
```

conc::when_ready

- Use conc::when_ready to pass an underlying value once ready

```
stlab::future<int> fut = /*...*/;

// Calls "print" through "exec" once "fut" is ready.
async_forgetful_call(
  exec, print, std::move(stream), conc::when_ready(std::move(fut)));
```

ConcurrentArgumentProvider

ArgumentProvider Vs. ConcurrentArgumentProvider

	ArgumentProvider	ConcurrentArgumentProvider
Used with	argot::call	argot::async_forgetful_call argot::async_call
Symbolizes	Sum type of argument lists	Future to sum type of argument list

The CDWIW Operator

- Analogous to +, but for the concurrent world
  - Syntax: ~concurrent_expandable
  - Intentionally in namespace argot::concurrent_expansion_operator to avoid ADL

```
stlab::future<int> fut = /*...*/;

// Calls "print" through "exec" once "fut" is ready.
async_forgetful_call(
  exec, print, std::move(stream), ~std::move(fut));
```

Expanding Many Futures Is Equivalent to "when_all"

```
stlab::future<int> fut1 = /*...*/;
boost::shared_future<int> fut3 = /*...*/;

// Async call removes the need for explicit when_all
async_forgetful_call(
  exec, print, std::move(stream), ~fut1, 2, ~fut3);
```

Back to argot::async_call

- Works just like argot::async_forgetful_call, but **returns a future**
- Argot *does not* provide a general-purpose future type, so which you want **must be specified**
  - Pass a FuturePackager to async_call to inform the library what to produce
  - FuturePackagers can hypothetically be deduced in some cases, but not yet

```
// Calls "print" through the Executor "exec".
stlab::future<void> result
  = async_call<packager::stlab>(
      exec, print, std::move(stream), 1, 2, 3);
```

FuturePackager

- A FuturePackager is a type with an associated function that takes an **Executor** and an **Invocable**, runs the Invocable through the Executor, and returns an stlab-like **task and future pair**.
  - Replaces the need for a promise type
  - Allows a user to control what kind of future comes out of an asynchronous call
    - packager::stlab
    - packager::boost_future
    - packager::boost_shared_future

```
// Use a FuturePackager directly (uncommon)
auto [task, fut]
  = packager_traits::package<packager::stlab>(
      some_executor, []{ /*...*/ } );

task(); // Issue the call, eventually fulfilling "fut"
```

Expanding Tuples and Variants

- conc::unpack_by_value or conc::alternative_of_by_value
  - Similar to prov::unpack and prov::alternative_of, but usable with async_call and **capture by value** instead of by reference
  - The ~ operator can be used as a short-hand

```
std::tuple<int, float, char> tup = /*...*/;

async_call<packager::stlab>(exec, print, std::move(stream), ~tup);
```

Overall Benefits of argot::async_call

- Whether a "bare" asynchronous invocation or one done via continuations, argot::async_call may be used (never need to chain ".then" calls)
- The eventual function to call is not obscured
  - Directly express "execute *this* function with *these* arguments once ready"
- Argument capture has sensible defaults (reduced risk of dangling references)
- Can *hypothetically* use a when_all behind the scenes (not currently)

Future-Related Concepts

Why these concepts, specifically

Core Concepts

- Executor
- Future
- FuturePackager
- ConcurrentArgumentProvider

How Were the Concepts Derived?

- Concepts were **lifted** from the necessities of async_call and async_forgetful_call
  - Though **Executor** is similar to stlab's, that is not where it came from
  - Executor concept is **minimal** when compared to Executors TS and Networking TS
  - FuturePackager is the **glue** between a Future and an Executor
- Additional backend complexity arises due to the interface that existing future types choose to expose

Thenable

- SemiRegular
  - Movable, Destructible
- Associated value_type
- Associated "then" function taking an Executor and a FuturePackager

ForgetfulThenable

- Similar to capabilities of one-way executors of the Executors TS
- Same as Thenable, but with "fire and forget" semantics
  - More efficient when a Future result is not needed
- Efficient building-block when used with FuturePackager
  - Can build the Future-returning "then"
  - Can build an efficient "join" (Future<Future<T>> -> Future<T>)
  - Can build an efficient Future-kind converter (boost::future<T> -> stlab::future<T>)

"Future" Direction

Suggested direction for implementation of futures

A Future Concept Is Inevitable

- A single future template cannot feasibly handle all situations
- High-level code can work across multiple future kinds
- A future concept does not imply a lack of a "vocabulary" future
  - Container/range concepts are useful even with std::vector
  - The Invocable concept is useful even with std::function
- Argot's concepts were lifted for the needs of *my* generic code and are likely not sufficient for arbitrary algorithms

Questions

or comments poorly disguised as questions