Index based coproduct operations on variant, wording
v0.3

Esa Pulkkinen
esa.pulkkinen@iki.fi

December 27, 2024

Revision History

Revision	Date	Author(s)	Description
v0.1	3.10.2024	Esa Pulkkinen	Initial version
v0.2	25.12.2024	Esa Pulkkinen	Add categorical semantics
v0.3	27.12.2024	Esa Pulkkinen	Add examples and loose ends

1 Introduction

There is a known problem with C++ `std::variant`'s visit operations, which are useful when variant's branches do not have multiple uses of same types as branches. When using the overloaded visiting, only differences encoded in types would be allowed, but when multiple branches have same type, the behaviour of the visit operations using an overload-set based matching does not allow distinguishing them.

This came up for class invariants in P3361 [7], which attempted to rely on coproducts in category theory to describe semantics of class invariants and contract checking in general, however current `std::variant` implementation didn't quite support sufficient operations, and therefore linking coproduct semantics with current approach in `std::variant` seemed difficult.

Here are some examples, what you might need to do to use a variant:

```
std::variant<int,int,int> v{std::in_place_index
  <0>(10)}; // OK

// ill-formed, also can't tell which int:
// std::visit(v, [](int i) { ... })
```

```
int x = std::get<0>(v); // works
int y = std::get<1>(v); // throws exception

// ill-formed, and can't tell which int was selected
 :
// int z = std::get<int>(v);

// ugly, inefficient code, but works:
int a;
try {
  a = std::get<0>(v);
} catch (std::bad_variant_access &e) {
  try {
    a = std::get<1>(v);
  } catch (std::bad_variant_access &e) {
    a = std::get<2>(v);
  }
}
// also ugly:
if (int * ap0 = std::get_if<0>(&v)) {
  a = *ap0;
} else if (int *ap1 = std::get_if<1>(&v)) {
  a = *ap1;
} else if (int *ap2 = std::get_if<2>(&v)) {
  a = *ap2;
} else { throw std::bad_variant_access(); }

// ugly, procedural style code, but works, now all
// branch-specific code now is inside the switch-
 case:
switch (v.index()) {
  case 0: a = std::get<0>(v); break;
  case 1: a = std::get<1>(v); break;
  case 2: a = std::get<2>(v); break;
  default: throw std::bad_variant_access();
}
std::variant<std::tuple<int,int>, std::pair<int, std
 ::string> > v{
  std::in_place_index<0>, std::make_tuple(2,3)
};
auto f = [](int x , int y) { return x + y; };
auto g = [](int x, std::string name) { return x +
 name.length(); }
// now arguments to either f and g are in variant,
 but how to call it?
int res = 0;
switch (v.index()) {
  case 0: res = std::apply(f,std::get<0>(v)); break;
  case 1: res = std::apply(g,std::get<1>(v)); break;
  default: throw std::bad_variant_access();
}
```

To a functional programmer, all of the above examples seem very complicated and procedural.

It's possible nonetheless to distinguish different branches using index based lookup from variant, that is, `std::get<I>` for a compile-time index. However these are not particularly useful when the index of the chosen branch is not known at compile-time. This is in particular necessary to support correct pattern matching operations for representing a coproduct in category theory using variant. In particular, a coproduct has "injections", which are represented by already existing variant constructors that use index, and "index based case matching", which are described here.

It's important to see that order of declaration now matters, and the functions to process data from variant must be given in the same order they are declared in the variant. But that's because it's the indexing that is used to match branches of the variant with actual functions.

Here are some examples of how index base case matching could work.

```
std::variant<int,std::string, int> v{std::
 in_place_index<2>, 66};
std::variant<int,std::string, int> w{std::
 in_place_index<1>, "foo"};

// order of branches matters:
auto compute = invoke_cases(
      [](int i) -> int { return i; },
      [](std::string const &s) -> int { return s.
 length(); },
      [](int j) -> int { return j + 100; }
      );

std::cout << "res=" << compute(v) << "," << compute(
 w) << std::endl;

using message = std::tuple<int,int>;
using message2 = std::pair<int,std::string>;

std::variant<message, message2> args{
  std::in_place_index<0>, std::make_tuple(3,4)
};
std::variant<message, message2> args2{
  std::in_place_index<1>, std::make_pair(3,"
 teststring")
};
auto analyze = apply_cases(
       [](int x, int y) -> int { return x + y; },
       [](int x, std::string const &y) -> int {
 return x + y.length(); });

int result = analyze(args);
int result2 = analyze(args2);
int result3 = visit_apply_cases(args,
      [](int x, int y) -> int { return x + y; },
      [](int x, std::string const &y) -> int {
 return x + y.length(); });

 auto tup = std::make_tuple(
  [](int x, int y) -> int { return x + y; },
  [](int x, std::string const &y) -> int { return x
 + y.length(); }
  );
int result4 = visit_apply(args,tup);
int result5 = visit_apply(args2,tup);
```

For the coproduct index based case matching, the important consideration is that it allows combining several functions indexed by a compile-time integer into one function, whose input is a variant whose branches are indexed similarly by the integer. Such operation is often considered primitive in functional programming languages such as Haskell [1].

As summary, the plan is to support following operations:

operation	multi-parameter functions	variant first parameter	multiple functions
visit_invoke	No	Yes	Tuple
visit_invoke_cases	No	Yes	Variadics
invoke_cases	No	No	Variadics
visit_apply	Yes	Yes	Tuple
visit_apply_cases	Yes	Yes	Variadics
apply_cases	Yes	No	Variadics

The idea in naming these operations is that "invoke" is used if functions have one parameter, "apply" for cases where more than one parameter is supported. The "cases" suffix is used if the functions are listed as variadic arguments. The "visit" prefix is used if the first argument is the variant to be analyzed. If not, then to call the operation, you need two calls, where first call is for a list of functions, and second call passes in the variant.

2 Categorical semantics

The appropriate category theory commutative diagram is the definition of the coproduct. The definitions for these are well known [2] [3]. Notice that these are intended to represent coproduct of arbitrary number of alternatives, not the binary coproducts. The extension to more than two alternatives is a standard construction.

$$\begin{tikzcd}
A_i \arrow[r, "\mathbf{in}_i"] \arrow[dr, "f_i"'] & \amalg_i A_i \arrow[d, "{[f_i]}"] \\
 & B
\end{tikzcd}$$

There is a simple mechanism for visiting a variant which allows matching by index. The $\mathbf{in}_i$ is the constructor for variant indexed by compile-time constant

i. The $[f_i] = [f_i]_{i \in [0,n-1]}$ will be called `invoke_cases`$(f_0, ..., f_{n-1})$ and satisfies the principle that `invoke_cases`$(f_0, ..., f_{n-1})(\mathbf{in}_i(x_i)) = f_i(x_i)$.

The parameter to f_i has type $x_i : 1 \to A_i$, and since the type depends on the index, I've used index for x_i as well.

I allow each A_i to be a product. The C++ implementation will support separate functions to support functions f_i that take multiple parameters, as a generalization of the one-parameter case. The corresponding function to `invoke_cases` will be called `apply_cases`. That satisfies the same principle than `invoke_cases`, except that it allows multiple parameter functions, and requires the arguments to be wrapped in a tuple. So each A_i in that case is considered to be of form $\Pi_j U_{ij}$, and corresponding functions $f_i : (\Pi_j U_{ij}) \to B$. So `apply_cases`$(f_0, ..., f_{n-1})(\mathbf{in}_i(\langle x_{i0}, ..., x_{i(k-1)} \rangle)) = f_i(x_{i0}, ..., x_{i(k-1)})$, where $i \in [0, n-1]$ and $j \in [0, k-1]$.

The C++ implementation uses functions that take tuples of functions instead of a variadic function with multiple function parameters. Those are useful primitives to construct more complicated cases. In C++ the case where $n = 2$ is special, because `std::pair` and `std::tuple` with two parameters provide similar (isomorphic) functionality. The `std::get` and `std::apply` already provide sufficient abstraction for that.

The case where $n = 0$ is also special in that coproduct of zero alternatives should be the initial object, but operation to extract values always fails. The case where $n = 1$ has the special property that operations to extract the values cannot fail.

To represent an indexed set of functions, a tuple containing function objects is natural, however for ease of use, it's useful to support operations where user doesn't need to explicitly construct the tuple. A tuple can be described as following commutative diagram, which is dual to the one for coproduct.

$$\begin{tikzcd}
\Gamma \arrow[r, "\langle a_i \rangle"] \arrow[dr, "a_i"'] & \Pi_i X_i \arrow[d, "\pi_i"] \\
& X_i
\end{tikzcd}$$

The operation π_i is the `std::get<I>` operation, and $\langle a_i \rangle = \langle a_i \rangle_{i \in [0,n-1]}$ is `make_tuple`$(a_0, ..., a_{n-1})$.

To specialize this to represent an indexed set of functions for coproduct, I set $X_i = A_i \to B$ and use exponentials $(-^*, \mathbf{app}) : - \times A \dashv A \to -$ to produce the function based on the coproduct definition. This produces following commutative diagram:

$$\begin{tikzcd}
\Gamma \arrow[r, "\langle f_i^* \rangle"] \arrow[dr, "f_i^*"'] & \Pi_i (A_i \to B) \arrow[d, "\pi_i"] \\
& A_i \to B
\end{tikzcd}$$

Using exponentials in reverse, and using the coproduct, we can turn this into commutative diagram:

$$\begin{tikzcd}
\Gamma \times A_i \arrow[r, "\langle f_i^* \rangle \times id_{A_i}"] \arrow[dr, "f_i^* \times id_{A_i}"'] \arrow[dd, "id_\Gamma \times \mathbf{in}_i"'] & \Pi_i(A_i \to B) \times A_i \arrow[d, "\pi_i \times id_{A_i}"] \\
 & (A_i \to B) \times A_i \arrow[d, "\mathbf{app}"] \\
\Gamma \times \Pi_i A_i \arrow[r, "{[f_i] \circ \mathbf{snd}}"] & B
\end{tikzcd}$$

Notice that when constructing the coproduct, the context is not used, whereas when going through the lambda, the function used to deconstruct the variant may depend on the context.

This paper is intended to formalize changes to C++ standard for introducing index-based coproduct visit operations for variant to C++.

The `visit_invoke` operation described above doesn't support functions where function takes more than one parameter. To support that case, I will add a new operation which uses `std::apply` to apply the function on the variant branch to multiple arguments that are described in a tuple. This provides a generalization where parameters to variant are not limited to one argument. So in that case each branch of the variant should be a tuple of parameters, which would be passed on to the functions in the tuple.

3 Changes to standard

The changes are against [5]. An example implementation which doesn't need compiler extensions is given in the appendix.

3.1 coproduct operations

This should be added after 22.6.7 [variant.visit]:

3.1.1 visit_invoke

```
  template <class Tuple, size_t I = 0, class... Alts>
constexpr auto visit_invoke(std::variant<Alts...>
   const &v, Tuple const &tup)
requires (I >= 0
  && I < tuple_size_v<remove_reference_t<Tuple> >
  && sizeof...(Alts) == tuple_size_v<
   remove_reference_t<Tuple> >);
```

The semantics should be same as returning the results of following expression:

```
invoke(
  get<LIFT(v.index())>(forward<Tuple>(tup)),
  get<LIFT(v.index())>(v)
);
```

Where `LIFT(v.index())` is a constant expression referring to currently chosen branch of variant in `v`. The lifting of the run-time index of the chosen branch of the variant to compile-time expression should use an implementation-defined mechanism, which is indicated as "LIFT" here. Practical library implementations could use a switch statement with appropriate mechanisms to prevent ill-formed constructs together with a default branch that invokes the operation recursively, or a more sophisticated implementation-defined mechanism.

3.1.2 visit_apply

```
template <class Tuple, size_t I = 0, class ... Alts>
constexpr decltype(auto)
visit_apply(std::variant<Alts ...> const &v, const
 Tuple &t)
requires (I >= 0
&& I < std::tuple_size_v<std::remove_reference_t<
 Tuple> >
&& sizeof...(Alts) == std::tuple_size_v<std::
 remove_reference_t<Tuple> >);
```

The semantics should be same as returning the results of following:

```
return std::apply(get<LIFT(v.index())>(t), std::get<
 LIFT(v.index())>(v));
```

Where again the "LIFT" is as above.

3.1.3 visit_apply_cases

```
template <class... Alts, class... F>
constexpr decltype(auto)
visit_apply_cases(variant<Alts> const &v,
F && ... funcs)
requires (sizeof...(Alts) == sizeof...(F));
```

The semantics should be:

```
return visit_apply(v, std::make_tuple(std::forward<F
 >(funcs)...));
```

3.1.4 apply_cases

```
template <class... F>
constexpr auto apply_cases(F && ... funcs);
```

The semantics should be:

```
return [fs = std::make_tuple(std::forward<F>(funcs)
 ...)]
      <class... Alts>(std::variant<Alts...> const &
 v)
requires (sizeof...(Alts) == sizeof...(F)) { return
 visit_apply(v,fs); }
```

3.1.5 visit_invoke_cases

```
template <class... Alts, class... F>
constexpr auto
visit_invoke_cases(
std::variant<Alts> const &v, F && ... funcs)
-> common_type_t<invoke_result_t<F, Alts>...>
requires (sizeof...(Alts) == sizeof...(F));
```

The semantics should be:

```
return visit_invoke(v, std::make_tuple(std::forward<
 F>(funcs)...));
```

3.1.6 invoke_cases

```
template <class... F>
constexpr auto invoke_cases(F && ... funcs);
```

Semantics should be:

```
return [fs = std::make_tuple(std::forward<F>(funcs)
 ...)]
   <class... Alts>(std::variant<Alts...> const &v)
-> std::common_type_t<std::invoke_result_t<F, Alts
 >...>
requires (sizeof...(F) == sizeof...(Alts))
{ return visit_invoke(v,fs); };
```

4 Loose ends

The following known issues have not had careful analysis:

1. Naming of the operations. It's possible to support std::variant member functions with similar semantics for the visit operations that take variant as first argument. Also various overloaded operations which might automatically choose between, say, `visit_invoke_cases` and `visit_apply_cases` based on whether the functions used are multi-argument functions are possible. However to avoid overloading-based ambiguities on their semantics I've used distinct name.
2. Interaction with pattern matching proposals [6] [4].
3. Interaction with `std::expected`, `std::optional` or `std::any`, or other existing coproduct-like types.

5 Appendix

These functions have been implemented. As an example, the `visit_invoke` operation can be implemented as follows:

```
template < class Tuple , size_t I = 0, class... Args>
constexpr decltype(auto) visit_invoke(std::variant<
   Args...> const &v, const Tuple &t)
  requires (I >= 0 && I < std::tuple_size_v<std::
   remove_reference_t<Tuple> >
           && sizeof...(Args) == std::tuple_size_v<
   std::remove_reference_t<Tuple> >)
{
  constexpr std::size_t SZ = std::tuple_size_v<std::
   remove_reference_t<Tuple> >;
  switch (v.index()) {
  case I: if constexpr (I < SZ)   { return std::invoke
   (std::get<I>(t), std::get<I>(v)); }
  case I+1: if constexpr (I+1<SZ) { return std::invoke
   (std::get<I+1>(t),std::get<I+1>(v)); }
  case I+2: if constexpr (I+2<SZ) { return std::invoke
   (std::get<I+2>(t),std::get<I+2>(v)); }
  case I+3: if constexpr (I+3<SZ) { return std::invoke
   (std::get<I+3>(t),std::get<I+3>(v)); }
  case I+4: if constexpr (I+4<SZ) { return std::invoke
   (std::get<I+4>(t),std::get<I+4>(v)); }
  case I+5: if constexpr (I+5<SZ) { return std::invoke
   (std::get<I+5>(t),std::get<I+5>(v)); }
  default: if constexpr (SZ > I+6) {
      return visit_invoke<Tuple,I+6, Args...>(v,t);
    } else {
      throw std::bad_variant_access();
    }
  }
}
```

The `visit_apply` function is similar, but uses `std::apply` instead of `std::invoke`. The other functions' implementation is relying on these two, and are as described in their semantics.

References

[1] The Haskell 2010 committee. The haskell 2010 report. Technical report, 2010.

[2] Maarten M. Fokkinga. A gentle introduction to category theory. 1994.

[3] Joseph A. Goguen. A categorical manifesto. 1989.

[4] Bruno Cardoso Lopes, Sergei Murzin, Michael Park, David Sankel, Dan Sarginson, and Bjarne Stroustrup. Pattern matching. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r3.pdf.

[5] N4950: Working draft, standard for programmning language c++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4950.pdf.

[6] Michael Park. Pattern matching: match expression. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2688r1.pdf.

[7] Esa Pulkkinen. Class invariants and contract checking philosophy. https://esapulkkinen.github.io/cifl-math-library/C++/contracts.pdf, 2024.