P3561: Index based coproduct operations on variant, and library wording

P3561R2

Esa Pulkkinen
esa.pulkkinen@iki.fi

April 6, 2025

Revision History

Revision	Date	Author(s)	Description
P3561R0	8.1.2025	Esa Pulkkinen	Initial version for C++ standardization
P3561R1	15.1.2025	Esa Pulkkinen	Clarify `valueless_by_exception() = true` scenario
P3561R2	6.4.2025	Esa Pulkkinen	Describe semantics of `valueless_by_exception() = true` scenario. Many fixes to indices in the description of the categorical semantics. Expanded description of loose ends.

1 Introduction

This paper is intended for WG21, C++ standardization committee, in particular, should be reviewed by SG18 LEWGI and LEWG.

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

1.1 Motivation and scope

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 determine 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 based 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].

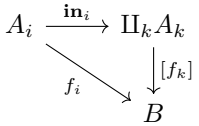
As summary, the plan is to support following operations:

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.



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

Notice the indices i and k vary independently. This is the source of content of the coproduct commutative diagram. For every i it must be that the *same* set of indices k satisfies $f_i = [f_k] \circ \mathbf{in}_i$. Thus $[f_k]$ must include all f_i for $i \in [0, n-1]$. So it's the coproduct injection $\mathbf{in}_i$ (a.k.a. variant constructor) which chooses which function from f_k the composition represents.

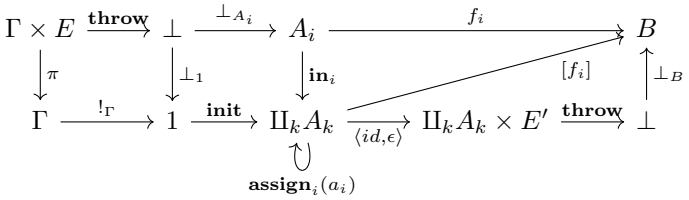
The parameter to f_i has type $x_i : 1 \to A_i$, and since the type depends on the index, I've used index for x_i as well.

The `visit_invoke` operation described above doesn't support functions where the 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 functions to deconstruct the variant are not limited to functions of 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.

Therefore, 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`$(f_0, ..., f_{n-1})(\mathbf{in}_i(\langle x_{i0}, ..., x_{i(k_i-1)}\rangle)) = f_i(x_{i0}, ..., x_{i(k_i-1)})$, where $i \in [0, n-1]$ and $j \in [0, k_i - 1]$.

Notice that it is not required that the functions take same number of parameters, and thus the number of parameters of each function, k_i, is indexed by i.

In C++, the variant has a special case of `valueless_by_exception`, which describes the scenario when move-assignment, copy-assignment, type changing assignment or type changing `emplace` operation raises exception. In such cases, the variant is initialized with the special value. To describe semantics of that special scenario as extension to the normal coproduct semantics, I add following commutative diagrams:

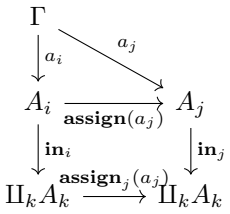


The `valueless_by_exception` scenario can be identified as the case where $\Gamma = \amalg_k A_k$ and a_i in $\mathbf{assign}_i(a_i)$ is $a_i = \bot_{A_i} \circ \mathbf{throw} \circ \langle id_\Gamma, e \rangle$ for some exception object constructor e.

Here **init** is a special operation to construct the empty variant describing the `valueless_by_exception` scenario and $\epsilon \in E'$ is the corresponding `bad_variant_access` exception object thrown if the variant is valueless. The point here is to say that any exception that occurs when constructing the variant object may leave the variant in an unique valueless by exception state, where observing the variant's state will produce exceptions. $\bot_B$ are the initial morphisms and $!_\Gamma$ is the terminal morphism. I do assume the categorical model of C++ has terminal objects, initial objects, binary products, products, coproducts and exponentials. When one of the variant's branches is the terminal object, $A_i = 1$, it's important to be able to distinguish this empty scenario from the $\mathbf{in}_i$ scenario, so the `init` operation should be identified by an (unspecified) index, that is not present for any of the variants' non-empty branches.

The distinction between the empty variant scenario and valid values is by existence of the index i for variant created using the index base variant constructors. To distinguish these cases from the empty scenario, a special index value is allocated. The standard has `variant_npos` for that purpose, as result of `index` query operation on the variant produced by `init`.

The assignment can be described as follows:



Implied here is the idea that assignment preserves the storage location of the assigned value. This notion is not reflected in the categorical model, and adding it would imply modelling pointers, then modelling which data is stored in what locations of memory, and then abstracting that using some appropriate functor. However, that is outside the scope of this paper. Here it's sufficient to require $\mathbf{addressof}(a_i) = \mathbf{addressof}(a_i.\mathbf{assign}(a_j))$. This implies that we want to require the corresponding $\mathbf{addressof}(in_i(a_i)) = \mathbf{addressof}(in_i(a_i).\mathbf{assign}_j(a_j))$.

The reason for requiring these is the following: There is an alternative implementation for `variant`-like type which allocates storage for all branches of the variant separately, and keeps track which branches are filled, basically a fancy version of a typed set that models $\Pi_k(1 + X_k)$:

```
template <class... T> using typed_set = tuple<optional<T>...>;
```

Then composing $\mathbf{assign}_i(a_i) \circ \mathbf{assign}_j(a_j)$ for $i \neq j$ would fill two branches

to produce a value that cannot be produced by a single invocation of variant constructor $\mathbf{in}_i$, violating the above commutative diagrams. To disallow that scenario for `variant`, the address of each branch of the variant should be allocated from same storage. The variant assignment must destroy the previously assigned value.

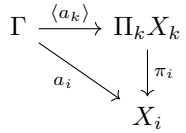
Notice here the underlying assignment can change the type of the assigned value, which is why $\mathbf{assign}(a_j)$ is not an endomorphism unless $i = j$. Normally state changes would be modelled as endomorphisms that are right actions of the operation identified by the method of a class, but that notion requires the types to be preserved. The variant assignment however preserves types. Anyway, it's necessary for the variant assignment to select which branch of variant contains the new value.

The variant can also be constructed using type-indexed constructors $\mathbf{in}_{A_i}$ when the types of branches are unique, and corresponding assignment operations exist. However, it's important to require $\mathbf{in}_{A_i} \equiv \mathbf{in}_i$ when there is an unique index i that identifies the A_i branch. If $A_i \cong A_j$ for $i \neq j$, the type indexed constructors cannot be used, and corresponding coproduct operations would not be able to choose an unique index.

To represent an indexed set of functions, a tuple containing function objects is natural. 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, and it's expected that other coproduct-like types should also support these operations.

The case where $n = 0$ is also special in that coproduct of zero alternatives should be the initial object, in such way that the operation to extract values always fails, and constructing the variant is ill-formed. However, the valueless-by-exception scenario means that such variant has a possible value, so the variant with zero alternatives is not quite the initial object. However, extracting values from such variant raises the `bad_variant_access` exception, which is almost as good. However comparison operations for two variants and the index query operation may be able to tell the difference and they are required not to raise exceptions.

For ease of use, it's useful to support operations where user doesn't need to explicitly construct the tuple $\langle f_k \rangle$. The tuple can be described as following commutative diagram, which is dual to the one for coproduct.

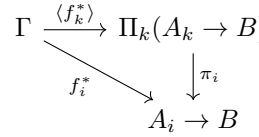


The operation π_i is the `std::get<I>` operation, and $\langle a_k \rangle = \langle a_k \rangle_{k \in [0,n-1]}$ is `make_tuple`$(a_0, ..., a_{n-1})$.

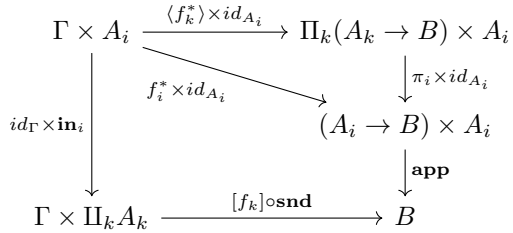
Here there is similar consideration for commutative diagram that $\langle a_k \rangle$ must have access to every value stored to the tuple because π_i must be able to choose any one of them, and k and i vary independently.

To specialize this to represent an indexed set of functions for coproduct, I set $X_k = A_k \to B$ and use exponentials $(-^*, \mathbf{app}, \eta_A, \phi_A) : - \times A \dashv A \to -$, a.k.a.

lambda abstraction, to produce the function based on the coproduct definition. This produces following commutative diagram:



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



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 means the variant is usable as messages in situations where data is sent from one context to another. However, the processing logic cannot be in such way separated from its context.

However, the indices k for the coproduct operation $[f_k]$ and the corresponding indices for the tuple of functions associated with each branch $\langle f_k^* \rangle : \Pi_k(A_k \to B)$ must match. This is expressed in C++ as a concept requirement involving `sizeof...` operator.

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 global operations

This should be added after 22.6.7 [variant.visit] so that these will be included in `<variant>` header.

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)
noexcept(/* described below */)
requires (I >= 0
  && I < tuple_size_v<Tuple>
  && sizeof...(Alts) == tuple_size_v<Tuple>);
```

Throws: `std::bad_variant_access` if `v.index() == std::variant_npos`.

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

```
invoke(
  get<LIFT(v.index())>(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<Tuple >
&& sizeof...(Alts) == std::tuple_size_v<Tuple>);
```

Throws: `std::bad_variant_access` if `v.index() == std::variant_npos`. 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));
```

Throws: `std::bad_variant_access` if `v.index() == std::variant_npos`. 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); }
```

The returned function throws: `std::bad_variant_access` if `v.index() == std::variant_npos`.

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));
```

Throws: `std::bad_variant_access` if `v.index() == std::variant_npos`.
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); };
```

The returned function throws: `std::bad_variant_access` if `v.index() == std::variant_npos`.

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.

Such approach would introduce a helper function as follows, and then use this `invoke_or_apply` instead of `invoke` or `apply` inside implementation of `visit_invoke`.

```
template <class F, class... Args>
constexpr auto
invoke_or_apply( F&& f, Args && ... args)
  noexcept(std::is_nothrow_invocable_v<F, Args...>)
  -> std::invoke_result_t<F,Args...>
  requires std::invocable<F, Args...>
{
  return std::invoke(std::forward<F>(f), std::forward<Args>(args)...);
}

template <class F, class... Args>
constexpr auto
invoke_or_apply( F&& f, const std::tuple<Args...> & args)
  noexcept(std::is_nothrow_invocable_v<F, Args...>)
  -> std::invoke_result_t<F,Args...>
  requires std::invocable<F, Args...>
{
  return std::apply(std::forward<F>(f), args);
}
```

Following names and signatures may be possible for `visit_invoke` operation. However, adding these can impact include dependencies between `<tuple>` and `<variant>` headers.

```
  template <class... F, class ... Alts>
  auto std::tuple<F...>::operator()(std::variant<Alts...> const &v)
    requires (sizeof...(F) == sizeof...(Alts))
  { return visit_invoke(v,*this); }

  template <class... Alts, class... F>
  auto std::variant<Alts...>::visit(std::tuple<F...> const &t)
    requires (sizeof...(Alts) == sizeof...(F))
  { return visit_invoke(*this, t); }

  template <class... F, class... Alts>
  auto std::invoke(std::tuple<F...> const &t, std::variant<Alts...> const &v)
    requires (sizeof...(F) == sizeof...(Alts))
  { return visit_invoke(v,t); }
```

The member functions might possibly be implemented in terms of explicit object member functions to support classes derived from `tuple` or `variant`.

OTOH, `std::invoke` overload seems redundant if tuple has member `operator()`. Also since `invoke` is used internally by `visit_invoke` a possibility for recursive calls occurs in case of nested tuples. This could be intentional and useful.

And variant's `visit` overload would produce ambiguities with normal use of `visit`, in particular if tuple's `operator()` were introduced. So these seem mutually exclusive approaches.

The version of `variant<Alts...>::visit` that would use variadic template parameters for the functions called would be even more obviously be ambiguous with normal use of `visit`.

2. Interaction with pattern matching proposals [6] [4]. It's not obvious what is good syntax for pattern matching for the index based operations. Either the index of the matched branch should be provided by user in the syntax, or order of branches of the pattern matches should automatically determine the indices.

3. Interaction with `std::expected`, `std::optional` or `std::any`, or other existing coproduct-like types.

   Consider for example an overloaded operation with following signature:

   ```
   template < class Tuple, size_t I = 0, class T, class E>
   constexpr decltype(auto) visit_invoke(std::expected<T,E> const &v, const Tuple &t)
   requires (I >= 0 && I < 2 && std::tuple_size_v<Tuple> == 2);
   ```

   This would extend the usefulness of `visit_invoke` beyond the variant. Similar can be done for `optional`. However, the functions, e.g. `invoke_cases` which are just used to package an indexed set of functions need to include such extensions into the overload set of `operator()` of the returned function object. This can be done only for a fixed number of coproduct-like types inside the returned function object, and the default mechanism is not very extensible to additional types. So there you might need to introduce another extension point to such extensions. For that scenario returning a minimal tuple of functions is expected to be a good solution.

   `std::any` is more difficult since it's not by default obvious how many branches should be included, however maybe those can be deduced from the function parameter types.

4. If it were not for the `valueless_by_exception() == true` scenario, The noexcept specification for `visit_invoke` would look as follows.

   ```
   noexcept(noexcept([]<std::size_t... J>(std::
    integer_sequence<std::size_t, J...>) constexpr
                      noexcept((std::
    is_nothrow_invocable_v<std::tuple_element_t<J,
    Tuple>,
                                std::
    variant_alternative_t<J, std::variant<Alts...>
    > > && ...))
     {}(std::index_sequence_for<Alts...>{})))
   ```

   However it seems important to throw an exception if the variant is valueless, and this information is only available at run-time. So declaring these functions `noexcept(false)` seems better. Also my experiments indicated that this long noexcept specification seemed to always return false, which was surprising.

5. Optimization: instead of throwing `bad_variant_access`, it's possible to use `__builtin_unreachable()` since concepts are used for checking the compile-time index, and run-time indices requested from an initialized variant should already be checked by the switch-case.

   However, `valueless_by_exception() == true` case would become unsafe. In that case, a precondition `pre(not v.valueless_by_exception())` would be needed. For users that explicitly specify starting index to avoid specifying functions for some branches, the branches skipped would also produce such errors. This optimization is not proposed.

6. Analyzing a variant with one or more `std::monostate` branch cannot be used to call functions with empty parameter list. Instead monostate needs to declared as parameter to the function.

7. It's possible to support operations that turn a run-time index into a compile-time index by mapping `v.index()` into `std::in_place_index<I>` arguments to functions. This may be useful in building function templates that analyze a variant with an overload-set of functions.

8. the functions depend on both tuple and variant, so using it requires dependency to `std::apply` and `tuple_size_v` from `<tuple>` and `variant` from `<variant>` headers, and the implementation requires access to `std::invoke` from `<functional>`. Due to this, it's not obvious which header these facilities should be added to. It should be possible to use these when both `<tuple>` and `<variant>` have been included. For two-parameter case, even `std::pair` from `<utility>` is sufficient instead of tuple.

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... Alts>
constexpr decltype(auto) visit_invoke(std::variant<
   Alts...> const &v, const Tuple &t)
requires (I >= 0 && I < std::tuple_size_v<Tuple>
            && sizeof...(Alts) == std::tuple_size_v<
   Tuple>)
{
  constexpr std::size_t SZ = std::tuple_size_v<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)); }
  case std::variant_npos:
  default: if constexpr (SZ > I+6) {
      return visit_invoke<Tuple,I+6, Alts...>(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.