Document number: D0050R1
Date: 2015–10-31
Project: ISO/IEC JTC1 SC22 WG21
Programming Language C++,
Library Evolution Working Group
Reply-to: Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

C++ visitation for multiple sum types

Vicente J. Botet Escriba

Experimental visitation for multiple sum types.

This paper presents a proposal for generic visitation of sum types individually or by groups (product of sum types). It is similar to `std::experimental::visit` from [N4542], but works for any model of sum type. In particular the proposed std::experimental `variant<Ts>` and `optional<T>` are models of sum type.

The title of the proposal has changed to better match the propose of the paper (the original proposal [P0050R0] was "C++ generic match function").

Contents

Introduction

This paper presents a proposal for generic visitation of sum types individually or by groups (product of sum types). It is similar to `std::experimental::visit` from [N4542], but works for any model of sum type. In particular the proposed std::experimental `variant<Ts>` and `optional<T>` are models of sum type.

The title of the proposal has changed to better match the propose of the paper (the original proposal [P0050R0] was "C++ generic match function").

Motivation and Scope

Getting the value stored in sum types as `variant<Ts...>` or `optional<T>` needs to know which type is stored in. Using visitation is a common technique that makes the access safer.

While the last variant proposal [N4542] includes visitation; it takes into account only visitation of homogeneous variant types. The accepted optional class doesn't provides visitation, but it can be added easily. Other classes, as the proposed `expected` class, could also have a visitation functionality. The question is if we want to be able to visit at once several sum types, as `variant`, `optional`, (a possible `expected`), and why not smart pointers.

`std::experimental::apply()` [N3915] can be seen as a particular case of visitation of multiple types if we consider that any type can be seen as a sum type with a single alternative type.

Instead of a `visit` function, this proposal uses instead the `match` function that is used to inspect some sum types.

Tutorial

Customizing the `match/first_match` functions

The proposed `inspect/match/first_match` function works for sum types `ST` that have customized the following overloaded functions

```
template <class R, class F>
R match(ST const&, F&& );

template <class R, class F>
R first_match(ST const&, F&& );
```

For example, we could customize `boost::variant` as follows:

```
namespace boost {
  template <class R, class F, class ...Ts >
  R match(variant<Ts...> const& v, F&& f)
  { return apply_visitor(std::forward<F>(f), v); }
}
```

In addition we need to know the sum type alternatives if we want to use `match` with several sum types. This must be done by specializing the meta-function `sum_type_alternatives` as follows

```
template <class ...Ts >
  struct sum_type_alternatives<variant<Ts...>>
{
  using type = ...;
}
```

There must be defined specializations of the tuple-like helper meta-functions

`std::tuple_size` and `std::tuple_element` for the nested typedef `type`.

Using the `match` function to inspect one sum type

Given the `boost::variant` sum type, we could just visit it using the proposed `overload` function (See [P0051]).

```
boost::variant<int, X> a = 2;
boost::apply_visitor(overload(
  [](int i)
    {},
  [](X const& i)
    {
      assert(false);
    }
  ), v);
```

The same applies to the proposed `std::experimental::variant`

```
std::experimental::variant<int, X> a = 2;
std::experimental::visit(overload(
  [](int i)
    {},
  [](X const& i)
    {
      assert(false);
    }
  ), a);
```

We can use in both cases the variadic `match` function

```
boost::variant<int, X> a = 2;
std::experimental::match(a,
  [](int i)
    {},
  [](X const& i)
    {
      assert(false);
    }
  );

std::experimental::variant<int, X> a = 2;
std::experimental::match(a,
  [](int i)
```

```
      {},
    [](X const& i)
      {
        assert(false);
      }
    );
```

We can also use a single matcher

```
std::experimental::variant<int, X> a = 2;
std::experimental::match(a),
  std::experimental::overload(
  [](int i)
    {},
  [](X const& i)
    {
      assert(false);
    }
  ));
```

But the user must take care that passed function object choose exactly the best overload.

Using the `match` function to visit several sum types

The variant proposal provides visitation of multiple variants.

```
std::experimental::variant<int, X> a = 2;
std::experimental::variant<int> b = 2;
std::experimental::visit(overload(
  [](int i, int j )
    {
    },
  [](auto i, auto j )
    {
      assert(false);
    }
  ), a, b);
```

The `match` function generalizes the visitation for several instances of heterogeneous sum types, e.g. we could visit `variant` and `optional` at once:

```
std::experimental::variant<int, X> a = 2;
std::experimental::optional<int> b = 2;
std::experimental::match(std::tie_tuple(a, b),
  [](int i, int j )
    {
    },
  [](auto const &i, auto j )
    {
      assert(false);
    }
  );
```

Alternatively we could use an `inspect` factory that would wrap the tuple and provide two functions `match` and `first_match`

```
std::experimental::inspect(a, b).match( // or first_match
```

```
[](int i, int j )
  {
  },
[](auto const &i, auto j )
  {
    assert(false);
  }
);
```

This is the preferred interface for the authors.

Design rationale

Result type of `visitation`

We can consider several alternatives:

- same type: the result type is the type returned by all the overloads (must be the same),
- `common_type`: the result type is the `common_type` of the result of the overloads,
- explicit return type `R`: the result type is `R` and the result type of the overloads must be explicitly convertible to `R`,

Each one of these alternatives would need a specific interface:

- `same_type_match`
- `common_type_match`
- `explicit_type_match`

For a sake of simplicity (and because our implementation needed the result type) this proposal only contains a `match` version that uses the explicit return type. The others can be built on top of this version.

Let `Ri` be the return type of the overloaded functor for the alternative i of the ST.

- `same_type_match:` Check that all `Ri` are the same, let call it `R`, and only then call to the explicit `match<R>(...)`,
- `common_type_match:` Let be `R` the `common_type<Ri...>` and only if it exists call to the explicit `match<R>(...).`

Matt Calabrese has suggested a different approach:"

1. If `match` is given an explicit result type, use that.
2. If `match` is NOT given an explicit result type, use the explicit result type of the function object if one exists.
3. If neither `match` nor the function object have an explicit result type type, but all of the invocations would have the same result type, use that.
4. If none of the above are true, then we have a few final options:
    1. Produce a compile time error
    2. Return void or some stateless type

3. Try to do some kind of common type deduction

To be optimal with compile-times, which is something that becomes a serious consideration with variants having many states and/or doing n-ary visitation for n > 1, users would prefer option 1 or option 2, since otherwise the implied checking, especially if common type deduction is at play, could be considerable.”

Here ”the explicit result type of the function” stands for a nested `result_type` of the function object.

Note that [N4542] `std::experimental::visit` requires that all the overloads return the same type.

Multiple cases or overload

The matching functions accept several functions. In order to select the function to be applied we have two alternatives:

- use `overload` to resolve the function to be called
- use `first_overload` to do a sequential search for the first function that match.

The `match` function uses `overload` and the `first_match` uses `first_overload`.

Grouping with `overload/first_overload`

We could think that group all the functions with the `overload` function and let the variadic part for the sum types as `std::experimental::visit` does has advantages.

```
boost::variant<int, X> a = 2;
boost::optional<int> b = 2;
match<void>(std::tie_tuple(a, b), overload(
  [](int i, int j )
    {
      //...
    },
  [](...)
    {
      assert(false);
    }
  ));
```

This gives the impression that the algorithm is orthogonal, we have a single visit function (no need to make the difference between `match` and `first_match`.

`Order of parameters and function name`

The proposed `visit` function has as first parameter the visitor and then the visited variant.

```
visit(visitor, visited);
```

The proposed `match` function reverse the order of the parameters. The reason is that we consider that it is better to have the object before the subject.

```
match(sumType, matcher);
```

Of course we can also reverse the roles of the object and subject and tell that the object is the visitor.

`std::experimental::apply` function follows the same pattern

```
apply(fct, tpl);
```

If uniform function syntax is adopted we would have

```
visitor.visit(visited);
sumType.match(matcher);
visitor.accept(visited);
fct.apply(tpl);
```

const aware matchers

A matcher must match the exact type. The following will assert false

```
const boost::variant<int, X> a = 2;
std::experimental::match(a,
  [](int& i) { ++i; },
  [](auto const&)    { assert(false); }
  );
```

and the following will even not compile

```
const boost::variant<int, X> a = 2;
std::experimental::match(a,
  [](int& i) { ++i; }
  [](X const& i)
    {
      assert(false);
    },
  );
```

Variadics

The two parameters of the match function can be variadic. We can have several sum types and several functions overloading a possible match.

If we had a language type pattern matching feature (see [PM]) the authors guess it would be something like:

```
boost::variant<int, X, float> a = 2;
boost::optional<int> b = 2;
inspect (a, b) {
  when { int i, int j } :
    //...
  when { int i, auto j } :
    //...
```

```
    default:
      assert(false);
  }
```

The sum types would be grouped in this case using the inspect (a, b) and the cases would be the variadic part. The cases would be matched sequentially.

This is a major motivation to place the sum type variables as the first parameter and let the matchers variadic since the second parameter.

In addition, the match statement would allow to const-aware cases on some of the types

```
const boost::variant<int, X, float> a = 2;
boost::optional<int> b = 2;
inspect (a, b) {
  when {int i, int j } :
    //...
  when { X& x, auto j } :
    //...
  default:
    assert(false);
}
```

This is not possible with the variadic interface proposed in [N4542] as all the variants must be either const or not const. However the multiple sum types uses a const tuple as parameter, but the tuple types can be const and non-const and can contain references or not depending on whether we use `std::make_tuple` or `std::tie_tuple`.

```
const boost::variant<int, X, float> a = 2;
boost::optional<int> b = 2;
std::experimental::match(std::tie_tuple(a, b),
  [](int i, int j )
    {
    },
  [](auto const &i, auto j )
    {
      assert(false);
    }
  );
```

Customization point

A customization point must be defined for any sum type ST as an overload of this function

```
template <class ST, class F>
  match(ST & st, F&& f);

template <class ST, class F>
  match(ST const& st, F&& f);
```

It is required that any customization respect the following, but it can be more restrictive.

Requires: The invocation expression of `f` with for all the alternative types of the sum type `ST` must be a valid expression.

Effects: Calls f with the current contents of the sum type.

Throws: doesn't throws any other exception that the invocation of the callable.

Remark: While the `std::experimental::visit` function proposed in [] requires the invocation of the callable must be implemented in O(1), i.e. it must not depend on `sum_type_size<ST>`, this proposal suggest to leverage this requirement and considered it as a QOI.

Grouping with `overload`

We could also group all the functions with the `overload` function and let the variadic part for the sum types as `std::experimental::visit` does.

```
boost::variant<int, X> a = 2;
boost::optional<int> b = 2;
match<void>(std::tie_tuple(a, b), overload(
  [](int i, int j )
    {
      //...
    },
  [](...)
    {
      assert(false);
    }
  ));
```

This has the advantage of been orthogonal, we have a single match function (no need to have match and first_match. The liability is much a question of style, we need to type more.

Considering any type as a sum type with a single alternative

We could try to apply the language-like match with types that have as single alternative, themselves

```
int a = 2;
boost::optional<int> b = 2;
inspect (a, b) {
  when {int i, int j ) }:
    //... make use of i and j
  when {int i, auto j } :
    assert(false);
}
```

This seems not natural as we are able to use directly the variable a inside each match case.

```
int a = 2;
boost::optional<int> b = 2;
inspect (b) {
  when (int j ) :
    //... make use of a and j
  default:
    assert(false);
}
```

Using the library solution each case is represented by a function and the function would not have direct access to the variable a

```
int a = 2;
boost::optional<int> b = 2;
auto x = inspect(b).match(
  [a] ( int j ) {
    return sum(a,j);
  },
  [a] (auto const &j ) {
    assert(false);
  },
  [ ] (auto){
    assert(false);
}
```

Allowing types with a single alternative makes the code more homogeneous

```
int a = 2;
boost::optional<int> b = 2;
inspect(a, b).match(
  [] (int i, int j) {
    return sum(i,j);
  },
  [] (int i, auto const &j ) :
    assert(false);
  [] (auto, auto) {
    assert(false);}
);
```

sum_type_alternatives

[N4542] variant proposal specialize `tuple_element` and `tuple_size` for `variant<Types...>`. The author think that as variant is not a product type, specializing these mete-functions is weird. However the alternatives of a sum type are a product type and so we can specialize `tuple_element` and `tuple_size` on `sum_type_alternatives<ST>::type`.

The current implementation uses a pattern matching approach and it requires just `sum_type_alternatives<ST>::type` to be a variadic template instantiation. As for example

```
template < class ...Ts >
struct type_list {};
```

Note that his proposal doesn't proposes this class, this is just an example.

This restriction shouldn't be part of the standard proposal.

Open Points

The authors would like to have an answer to the following points if there is at all an interest in this proposal:

- **`Would the visitation be done using the best match or the first match? or both?`**
- **`Which strategy for the match return type?`**
- **`Would the customization/mapping be done using ADL, traits, ...?`**
- **`Which complexity for the customization point?`**
- **`match` versus `visit versus inspect`**

  N proposes a `visit` function to visit variants that takes the arguments in the reverse order.

  What do we prefer `visit` or `match`?

  Which order of arguments do we prefer?

  Do we want a variadic function of overloads or just an overloaded visitor functor?

  Do we want inspect?
- **Seen a type T as a sum type with a unique alternative.**

  Do we want to support this case?
- **Do we want `sum_type_alternatives`?**
- **Do we want `type_list`?**

Technical Specification

Header <experimental/functional> Synopsis

```
namespace std {
namespace experimental {
inline namespace fundamental_v2 {

  template <class ST>
    struct sum_type_alternatives; // undefined
  template <class ST>
    struct sum_type_alternatives<ST&> : sum_type_alternatives<ST> {}
  template <class ST>
    struct sum_type_alternatives<ST&&> : sum_type_alternatives<ST> {}
  template <class ST>
    struct sum_type_alternatives<const ST> : sum_type_alternatives<ST> {}
  template <class ST>
    struct sum_type_alternatives<volatile ST> : sum_type_alternatives<ST> {}
  template <class ST>
    using sum_type_alternatives_t = typename sum_type_alternatives<ST>::type;
  template <class ST, int N>
    using sum_type_alternative = tuple_element<sum_type_alternatives<ST>, N>;
```

```
  template <class ST>
    using sum_type_size = tuple_size<sum_type_alternatives<ST>>;
  template <class T, class F>
    'see below' match(const T &that, F && f);
  template <class R, class ST, class... Fs>
    'see below' match(const ST &that, Fs &&... fcts);
  template <class R, class... STs, class... Fs>
    'see below' match(const std::tuple<STs...> &those, Fs &&... fcts);

}}}
```

Type trait sum_type_alternatives

The nested type `sum_type_alternatives<ST>::type` must define the tuple-like helper meta-functions `std::tuple_size` and `std::tuple_element` and be a variadic template instance.

Template function match

```
template <class R, class ST, class... Fs>
'see below' match(ST const& that, Fs &&... fcts);

template <class R, class ST, class... Fs>
'see below' match(ST& that, Fs &&... fcts);
```

Requires: Let R`i` be `decltype(overload(forward<Fs>(fcts)...)(declval<sum_type_alternative<ST,i>()) for i in 1...sum_type_size<ST>.` `Ri` must be explicitly convertible to `R`.

Returns: the result of calling the overloaded functions `fcts` depending on the type stored on the sum type using the customization point match as if

```
return match(that, overload(forward<Fs>(fcts)...));
```

Remarks: This function will not participate in overload resolution if ST is a tuple type.

Throws: Any exception thrown during the construction any internal object or thrown by the call of the selected overloaded function.

```
template <class R, class... STs, class... Fs>
'see below' match(const std::tuple<STs...> &those, Fs &&... fcts);
```

Requires: Let `{i,j, …}` one element of the cartesian product `1...sum_type_size<STs>....`

`decltype(overload(forward<Fs>(fcts)...)(declval<sum_type_alternative<STs,i>>(), declval<sum_type_alternative<STs,j>>(), ...)` must be explicitly convertible to `R.`

Returns: the result of calling the overloaded functions `fcts` depending on the type stored on the sum types `STs....`

Throws: Any exception thrown during the construction any internal object or thrown by the call of the selected overloaded function.

Customization point `match`

A customization point must be defined for any sum type ST as an overload of this function

```
template <class ST, class F>
  match(ST & st, F&& f);

template <class ST, class F>
  match(ST const& st, F&& f);
```

It is required that any customization respect the following, but it can be more restrictive.

Requires: The invocation expression of `f` with for all the alternative types of the sum type `ST` must be a valid expression.

Effects: Calls f with the current contents of the sum type.

Throws: doesn't throws any other exception that the invocation of the callable.

Remark: While the `std::experimental::visit` function proposed in [N4542] requires the invocation of the callable must be implemented in O(1), i.e. it must not depend on `sum_type_size<ST>`, this proposal suggest to leverage this requirement and considered it as a QOI.

Note: If we accept that any type can be seen as a sum type with it as single alternative we can add the following overload:

```
template <class T, class F>
auto match(T &const that, F&& f) -> decltype(f(that))
{ return f(that); }

template <class T, class F>
auto match(T &that, F&& f) -> decltype(f(that))
{ return f(that); }
```

Header <experimental/optional> Synopsis

```
namespace std {
namespace experimental {
inline namespace fundamental_v2 {
      template <class T >
        struct sum_type_alternatives<optional<T>>
      {
        using type = __type_list<nullopt_t, T>; // implementation defined
      }
      template <class R, class F, class ...Ts >
      R match(optional<T> const& v, F&& f)
      {
        if (v)
          return f(*v);
        else
          return f(nullopt);
      }
      template <class R, class F, class ...Ts >
      R match(optional<T>& v, F&& f)
```

```
      {
        if (v)
          return f(*v);
        else
          return f(nullopt);
      }
}}}
```

Header <experimental/variant> Synopsis

```
namespace std {
namespace experimental {
inline namespace fundamental_v2 {
      template <class ...Ts >
        struct sum_type_alternatives<variant<Ts...>> // implementation defined
      {
        using type = __type_list<Ts...>;
      }
      template <class R, class F, class ...Ts >
      R match(variant<Ts...> const& v, F&& f)
      { return visit(std::forward<F>(f), v); }
      template <class R, class F, class ...Ts >
      R match(variant<Ts...> & v, F&& f)
      { return visit(std::forward<F>(f), v); }
}}}
```

Implementation

There is an implementation at https://github.com/viboes/tags including a customization for `boost::variant` and `std::experimental::optional`.

Further work

The multiple sum type matching not only can be used with tuples created with `std::make_tuple`, we should be able to use also `std::tie_tuple` and `std::forward_as_tuple`.

Acknowledgements

Many thanks to Matt Calabrese for its insightful suggestions on the result type approach.

References

- [boost.variant] apply_visitor

  http://www.boost.org/doc/libs/1_59_0/doc/html/variant/reference.html#header.boost.variant.apply_visitor_hpp
- [N4542] N4542 - Variant: a type-safe union (v4) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4542.pdf

- [N3915] N3915 - apply() call a function with arguments from a tuple (V3)
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3915.pdf
- [P0051] P0051 - C++ generic overload function
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0051r0.pdf
- [PM] Open Pattern Matching for C++
  http://www.stroustrup.com/OpenPatternMatching.pdf