`visit<R>`: Explicit Return Type for `visit`

Document #:	D0655R0
Date:	2017-10-14
Project:	Programming Language C++ Library Evolution Group

Reply-to:	Michael Park <mcypark@gmail.com> Agustín Bergé <agustinberge@gmail.com>

1 Introduction

This paper proposes to allow visiting `variant`s with an explicitly specified return type.

2 Motivation and Scope

Variant visitation requires invocation of all combinations of alternatives to result in the same type, such type is deduced as the visitation return type. It is sometimes desirable to explicitly specify a return type to which all the invocations are implicitly convertible to, as if by *INVOKE*`<R>` rather than *INVOKE*:

```
struct process {
  template <typename I>
  auto operator()(I i) -> O<I> { /* ... */ };
};

std::variant<I1, I2> input = /* ... */;

// mapping from a `variant` of inputs to a `variant` of results:
auto output = std::visit<std::variant<O<I1>, O<I2>>>(process{}, input);

// coercing different results to a common type:
auto result = std::visit<std::common_type_t<O<I1>, O<I2>>>(process{}, input);

// visiting a `variant` for the side-effects, discarding results:
std::visit<void>(process{}, input);
```

In all of the above cases the return type deduction would have failed, as each invocation yields a different type for each alternative.

3 Impact on the Standard

This proposal is a pure library extension.

4 Proposed Wording

Modify **§23.7.2 [variant.syn]** of N4687 [1] as indicated:

```
  // 23.7.7, visitation
  template <class Visitor, class... Variants>
    constexpr see below visit(Visitor&&, Variants&&...);
+  template <class R, class Visitor, class... Variants>
+    constexpr R visit(Visitor&&, Variants&&...);
```

Add new paragraphs to **§23.7.7 [variant.visit]** of N4687 [1]:

```
template <class R, class Visitor, class... Variants>
  constexpr R visit(Visitor&& vis, Variants&&... vars);
```

Requires: The expression in the *Effects*: element shall be a valid expression for all combinations of alternative types of all variants. Otherwise, the program is ill-formed.

Effects: Let `is...` be `vars.index()...`. Returns *`INVOKE`*`<R>(forward<Visitor>(vis), get<is>(forward<Variants>(vars))...);`.

Throws: `bad_variant_access` if any `variant` in `vars` is `valueless_by_exception()`.

Complexity: For `sizeof...(Variants) <= 1`, the invocation of the callable object is implemented in constant time, i.e. it does not depend on `sizeof...(Types)`. For `sizeof...(Variants) > 1`, the invocation of the callable object has no complexity requirements.

5 Design Decisions

There is a corner case for which the new overload could clash with the existing overload. A call to `std::visit<Result>` actually performs overload resolution with the following two candidates:

```
template <class Visitor, class... Variants>
constexpr decltype(auto) visit(Visitor&&, Variants&&...);

template <class R, class Visitor, class... Variants>
constexpr R visit(Visitor&&, Variants&&...);
```

The template instantiation via `std::visit<Result>` replaces `R` with `Result` for the first overload, and `Visitor` with `Result` for the second, and we get the following:

```
template <class... Variants>
constexpr decltype(auto) visit(Result&&, Variants&&...);

template <class Visitor, class... Variants>
constexpr Result visit(Visitor&&, Variants&&...);
```

This results in an ambiguity if `Visitor&&` happens to be the same type as `Result&&`. For example, a call to `std::visit<Vis>(Vis{});` would be ambiguous since `Visitor&&` and `Result&&` are both `Vis&&`.

In general, we would first need a self-returning visitor, then an invocation to `std::visit` with the same type **and** value category specified for the return type **and** the visitor argument.

We claim that this problem is not worth solving considering the rarity of such a use case and the complexity of a potential solution.

Finally, note that this is not a new problem since `bind` already uses the same pattern to support `bind<R>`:

```
  template <class F, class... BoundArgs>
    unspecified bind(F&&, BoundArgs&&...);
  template <class R, class F, class... BoundArgs>
    unspecified bind(F&&, BoundArgs&&...);
```

6 Implementation Experience

An implementation of `visit<R>` as proposed here can be found in the `visit-r` branch of `mpark/variant`.

`eggs/variant` provides an implementation of `visit<R>` as `apply<R>`, and also handles the corner case mentioned above.

7 Future Work

There are other similar facilities that currently use *`INVOKE`* and does not provide an accompanying overload that uses *`INVOKE`*`<R>`.

Some examples are `std::invoke`, `std::apply`, and `std::async`.

There may be room for a guideline paper with clear outlines as to when such facilities should have an accompanying overload.

References

[1] 2017. Working Draft, Standard for Programming Language C++. *N4687*. Retrieved from

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4687.pdf