visit<R>: Explicit Return Type for visit

1 Introduction

This paper proposes allowing visiting variants 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. This 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 \texttt{INVOKE}\texttt{<R>} rather than \texttt{INVOKE}:

```cpp
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

Add to §19.7.2 [variant.syn] of N4762 [1]:

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

Add to §19.7.7 [variant.visit] of N4762 [1]:

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

1 Let \( n \) be sizeof...(Variants). Let \( m \) be a pack of \( n \) values of type size_t. Such a pack is called valid if \( 0 \leq m_i < \text{variant.size_v<remove_reference_t<\text{Variants}_i>}> \) for all \( 0 \leq i < n \). For each valid pack \( m \), let \( e(m) \) denote the expression:

```cpp
INVOKE\left(\text{std::forward<Visitor>(vis)}, \text{get}<m>(\text{std::forward<Variants>(vars)})\ldots\right) \quad /\!/ \text{see 19.14.3}
```

for the first form and

```cpp
INVOKE<R>\left(\text{std::forward<Visitor>(vis)}, \text{get}<m>(\text{std::forward<Variants>(vars)})\ldots\right) \quad /\!/ \text{see 19.14.3}
```

for the second form.

2 Requires: For each valid pack \( m \), \( e(m) \) shall be a valid expression. All such expressions shall be of the same type and value category; otherwise, the program is ill-formed.

3 Returns: \( e(m) \), where \( m \) is the pack for which \( m_i \) is \( \text{vars}_i.index() \) for all \( 0 \leq i < n \). The return type is \( \text{decltype}(e(m)) \) for the first form.

4 Throws: bad_variant_access if any variant in \( \text{vars} \) is valueless_by_exception().

5 Complexity: For \( n \leq 1 \), the invocation of the callable object is implemented in constant time, i.e., for \( n = 1 \), it does not depend on the number of alternative types of \( \text{Variants}_0 \). For \( n > 1 \), 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:

```cpp
template <class Visitor, class... Variants>
constexpr decltype(auto) 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 Visitor with Result for the first overload, R with Result for the second, and we end up with 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 Result&& happens to be the same type as Visitor&&. For example, a call to std::visit<Vis>(Vis{}); would be ambiguous since Result&& and Visitor&& 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>:  

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

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

6 Implementation Experience

- **MPark.Variant** implements visit<R> as proposed in the visit-r branch.
- **Eggs.Variant** has provided an implementation of visit<R> as apply<R> since 2014, and also handles the corner case mentioned in Design Decisions.

7 Future Work

There are other similar facilities that currently use INVOKE, and do 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 paper with clear guidelines as to if/when such facilities should have an accompanying overload.

References