Remove return type deduction in std::apply

Aaryaman Sagar
aary@meta.com

Eric Niebler
eniebler@nvidia.com

April 3, 2024

Document number: P1317R1

Library Evolution Working Group

1. Introduction

```cpp
#include <tuple>

template <class Func, class Tuple>
concept applicable =
    requires(Func&& func, Tuple&& args) {
        std::apply(std::forward<Func>(func), std::forward<Tuple>(args));
    };

int main() {
    auto func = []({});
    auto args = std::make_tuple(1);

    static_assert(!applicable<decltype(func), decltype(args)>);
}
```

The code above should be well formed. However, since `std::apply` uses return type deduction to deduce the return type, we get a hard error as the substitution is outside the immediate context of the template instantiation.

This paper proposes a new public trait instead of `decltype(auto)` in the return type of `std::apply`

2. Impact on the standard

This proposal is a pure library extension.

3. std::apply_result

`std::apply_result` (and the corresponding alias `std::apply_result_t`) is the proposed trait that should be used in the return type of `std::apply`. With the new declaration being:

```cpp
template <class F, class Tuple>
constexpr std::apply_result_t<F, Tuple> apply(F&& f, Tuple&& t);
```

This fixes hard errors originating from code that tries to employ commonly-used SFINAE patterns with `std::apply` that could have otherwise been well-formed. It is backwards compatible with well-formed usecases of `std::apply`

4. Implementation

`std::apply_result` can be defined using the existing `std::invoke_result` trait to avoid duplication in implementations:

```cpp
namespace std {
    // exposition only
    template <size_t I, class T>
    using element_at = decltype(get<I>(declval<T>()));

    // exposition only
    template <class F, class T, std::size_t... I>
```
constexpr auto apply-impl(F&& f, T&& t, std::index_sequence<I...>)
  noexcept(is_nothrow_invocable_v<F, element-at<I, T>...>)
  -> invoke_result_t<F, element-at<I, T>...> {
    return invoke(std::forward<F>(f), get<I>(std::forward<T>(t))...);
  }

template <class F, class Tuple>
using apply_result_t = decltype(
  apply-impl(
    declval<F>(),
    declval<Tuple>(),
    make_index_sequence<
      tuple_size_v<remove_reference_t<Tuple>>>()));

// 18.7.8, concept applicable
template<class F, class Tuple>
concept applicable = see below;

// 18.7.9, concept regular_applicable
template<class F, class Tuple>
concept regular_applicable = see below;

[Editorial note: Add the following two subsections at the end of [concepts.callable], right after [concept.strictweakorder]. — end note]

18.7.8 Concept applicable [concept.applicable]

- The applicable concept specifies a relationship between a callable type ([func.def]) F and a tuple-like type ([tuple.like]) Tuple that can be evaluated by the library function apply ([tuple.apply]).
template<class F, class Tuple>
concept applicable = requires(F&& f, Tuple&& tup) {
    apply(std::forward<F>(f), std::forward<Tuple>(tup)); // not required to be
    // equality-preserving
};

- [Example 1]: A function that generates random numbers can model applicable, since the apply function call expression is not required to be equality-preserving ([concepts.equality]). — end example]

18.7.9 Concept regular_applicable [concept.regularapplicable]

template<class F, class Tuple>
concept regular_applicable = applicable<F, Tuple>;

- The apply function call expression shall be equality-preserving ([concepts.equality]) and shall not modify the function object or the arguments.
[Note 1]: This requirement supersedes the annotation in the definition of applicable. — end note]
- [Example 1]: A random number generator does not model regular_applicable. — end example]
- [Note 2: The distinction between applicable and regular_applicable is purely semantic. — end note]
[Editorial note: Add the following to the listing in [meta.type.synop]. — end note]

21.3.3 [meta.type.synop]

// all freestanding
namespace std {
    [...]}

// 21.3.7, type relations
[...]}
template<class Fn, class... ArgTypes> struct is_invocable;
template<class R, class Fn, class... ArgTypes> struct is_invocable_r;

template<class Fn, class... ArgTypes> struct is_nothrow_invocable;
template<class R, class Fn, class... ArgTypes> struct is_nothrow_invocable_r;

template<class Fn, class Tuple> struct is_applicable;
template<class R, class Fn, class Tuple> struct is_applicable_r;

template<class Fn, class Tuple> struct is_nothrow_applicable;
template<class R, class Fn, class Tuple> struct is_nothrow_applicable_r;

// 21.3.8.2, const-volatile modifications
[...]}

// 21.3.8.7, other transformations
[...]}
template<class T> struct underlying_type;
template<class Fn, class... ArgTypes> struct invoke_result;
template<class Fn, class Tuple> struct apply_result;
template<class T> struct unwrap_reference;
template<class T> struct unwrap_ref_decay;
[...]}

}
template< class Fn , class... ArgTypes >
  using invoke_result_t = typename invoke_result<Fn , ArgTypes...>::type;

template< class Fn , class Tuple>
  using apply_result_t = typename apply_result<Fn , Tuple>::type;

template< class T>
  using unwrap_reference_t = typename unwrap_reference<T>::type;

// 21.3.7, type relations
... as before...

template< class R , class Fn , class... ArgTypes >
  constexpr bool is_nothrow_invocable_r_v
  = is_nothrow_invocable_r<R , Fn , ArgTypes...>::value;

template< class Fn , class Tuple>
  inline constexpr bool is_applicable_v = is_applicable<Fn , Tuple>::value;

template< class R , class Fn , class Tuple>
  inline constexpr bool is_applicable_r_v = is_applicable_r<R , Fn , Tuple>::value;

template< class Fn , class Tuple>
  inline constexpr bool is_nothrow_applicable_v = is_nothrow_applicable<Fn , Tuple>::value;

template< class R , class Fn , class Tuple>
  inline constexpr bool is_nothrow_applicable_r_v
      = is_nothrow_applicable_r<R , Fn , Tuple>::value;

// 21.3.9, logical operator traits
[...]

[Editorial note: Change [meta.rel] as follows. — end note]

1. The templates specified in Table 49 may be used to query relationships between types at compile time.

2. Each of these templates shall be a Cpp17BinaryTypeTrait [meta.rqmts] with a base characteristic of true_type if the corresponding condition is true, otherwise false_type.

3. Let ELEMS-OF(T) be the parameter pack get<N>(declval<T>()) , where N is the pack of size_t template arguments of the specialization of index_sequence denoted by make_index_sequence<tuple_size_v<T>>.

[Editorial note: At the end of Table 49, add the following. — end note]

Table 49: Type relationship predicates
<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class R, class Fn, class... ArgTypes&gt; struct is_nothrow_invocable_r;</td>
<td>is_invocable_r_v&lt;R, Fn, ArgTypes...&gt; is true and the expression \texttt{INVOKE}&lt;R&gt;(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...) is known not to throw any exceptions ([expr.unary.noexcept]).</td>
<td>Fn, R, and all types in the template parameter pack \texttt{ArgTypes} shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class Fn, class Tuple&gt; struct is_applicable;</td>
<td>The expression \texttt{INVOKE}(declval&lt;Fn&gt;(), \texttt{ELEMS-OF}(Tuple)...)) is well-formed when treated as an unevaluated operand.</td>
<td>Fn and Tuple shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class R, class Fn, class Tuple&gt; struct is_applicable_r;</td>
<td>is_applicable_v&lt;Fn, Tuple&gt; is true and the expression \texttt{INVOKE}(declval&lt;Fn&gt;(), \texttt{ELEMS-OF}(Tuple)...)) is known not to throw any exceptions ([expr.unary.noexcept]).</td>
<td>Fn and Tuple shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class Fn, class Tuple&gt; struct is_nothrow_applicable;</td>
<td>is_applicable_v&lt;Fn, Tuple&gt; is true and the expression \texttt{INVOKE}(declval&lt;Fn&gt;(), \texttt{ELEMS-OF}(Tuple)...)) is known not to throw any exceptions ([expr.unary.noexcept]).</td>
<td>Fn and Tuple shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
</tbody>
</table>

[Editorial note: Change [meta.trans.other] as follows. — end note]

21.3.8.7 Other transformations [meta.trans.other]

1. The templates specified in Table 55 perform other modifications of a type.

[Editorial note: Add the following to Table 55. — end note]

Table 55 — Other transformations [tab:meta.trans.other]
Remove return type deduction in std::apply

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>... as before ...</td>
<td></td>
</tr>
</tbody>
</table>
| template<class Fn
class... ArgTypes>
struct invoke_result; | If the expression `INVOKE` (declval<Fn>(),
declval<ArgTypes>()...) (22.10.4) is well-formed when treated as
an unevaluated operand (7.2.3), the member typedef `type` denotes
the type `decltype(INVOKE)` (declval<Fn>(),
declval<ArgTypes>()...)); otherwise, there shall be no member
type. Access checking is performed as if in a context unrelated to Fn
and ArgTypes. Only the validity of the immediate context of the
expression is considered.

[Note 2: The compilation of the expression can result in side effects such as
the instantiation of class template specializations and function template
specializations, the generation of implicitly-defined functions, and so on.
Such side effects are not in the “immediate context” and can result in the
program being ill-formed. — end note]
Preconditions: Fn and all types in the template parameter pack
ArgTypes are complete types, cv void, or arrays of unknown bound. |
| template<class Fn
class Tuple>
struct apply_result; | If the expression `INVOKE` (declval<Fn>(),
elems-OF(Tuple)...)
(22.10.4) is well-formed when treated as
an unevaluated operand (7.2.3), the member typedef `type` denotes
the type `decltype(INVOKE)` (declval<Fn>(),
elems-OF(Tuple)...)); otherwise, there shall be no member
type. Access checking is performed as if in a context unrelated to Fn
and Tuple. Only the validity of the immediate context of the
expression is considered.

[Note 3: The compilation of the expression can result in side effects such as
the instantiation of class template specializations and function template
specializations, the generation of implicitly-defined functions, and so on.
Such side effects are not in the “immediate context” and can result in the
program being ill-formed. — end note]
Preconditions: Fn and Tuple are complete types, cv void, or arrays
of unknown bound. |
| template<class T>
struct unwrap_reference; | If T is a specialization `reference_wrapper<X>` for some type X, the
member typedef type of `unwrap_reference<T>` denotes X&,
otherwise type denotes T. |
| ... as before ... |

[Editorial note: Change the listing in [tuple.syn] as follows. — end note]

Section 23.5.2 ([tuple.syn])

[...]

template<tuple-like... Tuples>
constexpr tuple<CTypes...> tuple_cat(Tuples&&...);

// 23.5.3.5, calling a function with a tuple of arguments
template<class F, tuple-like Tuple>
constexpr decltype(auto) apply_result_t<F, Tuple> apply(F&& f, Tuple&& t)
noexcept(see below if noexcept applicable_v< F, Tuple>);

[...]

[Editorial note: Change [tuple.apply] as follows. — end note]
22.4.6. Calling a function with a tuple of arguments [tuple.apply]

```
template<class F, tuple-like Tuple>
constexpr decltype(auto) apply_result_t<F, Tuple> apply(F&& f, Tuple&& t)
    noexcept(see below is noexcept_possible_v<F, Tuple>);
```

1. **Effects:** Given the exposition-only function template:

```cpp
class E { public: void f(int); void g(char); }; using V = std::vector<int>; using T = std::tuple<int, V, E>; int main() { std::tuple<int, V, E> t = std::make_tuple(0, std::vector<int>{}, std::unique_ptr<E>(new E{})); assert(apply(apply_result_t<F, Tuple>(std::forward<F>(f), std::forward<Tuple>(t)))), 0); return 0; }
```

Equivalent to:

```
return apply_impl(std::forward<F>(f), std::forward<Tuple>(t), make_index_sequence<tuple_size_v<remove_reference_t<Tuple>>>{})();
```

2. **Remarks:** Let I be the pack 0, 1, ..., (tuple_size_v<remove_reference_t<Tuple>> - 1). The exception specification is equivalent to:

```
noexcept(invoke(std::forward<F>(f), get<I>(std::forward<Tuple>(t))...))
```