Tuple application traits

Aaryaman Sagar (aary@instagram.com)

October 8, 2018
Document #: p1318
Library Evolution Working Group

1. Introduction

The previous paper (p1317) proposes removal of return type deduction with `std::apply` and adds in an explicit trait `std::apply_result` to serve as the return type. Making `std::apply` a suitable candidate for conditionally evaluated expressions that leverage SFINAE.

This paper introduces traits to complement `std::apply_result` that tend to come in handy. These are now consistent with `std::invoke` traits.

2. Impact on the standard

This proposal is a pure library extension.

3. `std::is_applicable`

`std::is_applicable` (and the corresponding template variable `std::is_applicable_v`) is a unary type trait that inherits from `std::true_type` if the passed tuple type can be expanded into an invocation of the passed function type, each with the same value categorizations.

3.1. Implementation

```
template <typename F, typename Tuple, typename = std::void_t<>>
struct is_applicable : std::false_type {};
template <typename F, typename Tuple>
struct is_applicable<F, Tuple, std::void_t<std::apply_result_t<F, Tuple>>>
    : std::true_type {};

template <typename F, typename Tuple>
constexpr auto is_applicable_v = is_applicable<F, Tuple>::value;
```

4. `std::is_applicable_r`

`std::is_applicable_r` (and the corresponding template variable `std::is_applicable_r_v`) is a unary type trait that inherits from `std::true_type` if the passed tuple type can be expanded into an invocation of the passed function type, each with the same value categorizations to yield a result that is converible to `R`.

4.1. Implementation

```
template <typename R, typename F, typename Tuple, typename = std::void_t<>>
struct is_applicable_r : std::false_type {};
template <typename R, typename F, typename Tuple>
struct is_applicable_r<R, F, Tuple, std::void_t<std::apply_result_t<F, Tuple>>>
    : std::is_convertible<std::apply_result_t<F, Tuple>, R> {};
```

```
template <typename F, typename Tuple>
constexpr auto is_applicable_r_v = is_applicable<F, Tuple>::value;
```

5. std::is_nothrow_applicable

`std::is_nothrow_applicable` (and the corresponding template variable `std::is_nothrow_applicable_v`) is a unary type trait that inherits from `std::true_type` if the passed tuple type can be expanded into an noexcept invocation of the passed function type.

5.1. Implementation

```
// apply_impl is for 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<F&&, decltype(std::get<I>(std::declval<T>()))...>{})
    -> invoke_result_t<F&&, decltype(std::get<I>(std::declval<T>()))...> {
  return invoke(std::forward<F>(f), std::get<I>(std::forward<T>(t))...);
}

template <typename F, typename Tuple, typename = std::void_t<>>
struct is_nothrow_applicable : std::false_type {};
template <typename F, typename Tuple>
struct is_nothrow_applicable<
    F,
    Tuple,
    std::void_t<std::apply_result_t<F, Tuple>>>
    : std::bool_constant<noexcept(apply_impl(
          std::declval<F>(),
          std::declval<Tuple>(),
          std::make_index_sequence<
              std::tuple_size_v<std::decay_t<Tuple>>>{}))> {};

template <typename F, typename Tuple>
constexpr auto is_nothrow_applicable_v = is_nothrow_applicable<F, Tuple>::value;
```

6. std::is_nothrow_applicable_r

`std::is_nothrow_applicable_t` (and the corresponding template variable `std::is_nothrow_applicable_r_v`) is a unary type trait that inherits from `std::true_type` if the passed tuple type can be expanded into an noexcept invocation of the passed function type and yield a result that is convertible to `R`.

6.1. Implementation

```
// apply_impl is for 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<F&&, decltype(std::get<I>(std::declval<T>()))...>{})
    -> invoke_result_t<F&&, decltype(std::get<I>(std::declval<T>()))...> {
  return invoke(std::forward<F>(f), std::get<I>(std::forward<T>(t))...);
}

template <typename R, typename F, typename Tuple, typename = std::void_t<>>
struct is_nothrow_applicable_r : std::false_type {};
template <typename R, typename F, typename Tuple>
struct is_nothrow_applicable_r<
    R,
```

```
    F,
    Tuple,
    std::void_t<std::apply_result_t<F, Tuple>>>
    : std::conjunction<
          std::is_convertible<std::apply_result_t<F, Tuple>, R>,
          std::is_nothrow_applicable<F, Tuple>> {};

template <typename F, typename Tuple>
constexpr auto is_nothrow_applicable_r_v =
    is_nothrow_applicable_r<F, Tuple>::value;
```

7. Changes to the current standard

7.1. Section 23.15.2 ([meta.type.synop])

```
// 23.15.6, 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 R, class Tuple> struct is_applicable;
template <class R, class Tuple> struct is_applicable_r;

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

// 23.15.6, type relations

template <class Fn, class... ArgTypes>
  inline constexpr bool is_invocable_v = is_invocable<Fn, ArgTypes...>::value;
template <class R, class Fn, class... ArgTypes>
  inline constexpr bool is_invocable_r_v = is_invocable_r<R, Fn, ArgTypes...>::value;
template <class Fn, class... ArgTypes>
  inline constexpr bool is_nothrow_invocable_v = is_nothrow_invocable<Fn, ArgTypes...>::value;
template <class R, class Fn, class... ArgTypes>
  inline 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;
```

7.2. Section 23.15.6 ([meta.rel])

7.2.1. std::is_applicable

Template

```
template <class F, class Tuple>
struct is_applicable;
```

Condition
The expression `std::apply(std::declval<Fn>(), std::declval<Tuple>())` is well-formed when treated as an unevaluated expression

Comments
`Fn` and `Tuple` shall be complete types, cv void or arrays of unknown bound

7.2.2. `std::is_applicable_r`

Template

```
template <class R, class F, class Tuple>
struct is_applicable_r;
```

Condition
The expression `std::apply(std::declval<Fn>(), std::declval<Tuple>())` is well-formed when treated as an unevaluated expression and yields a result that is convertible to `R`

Comments
`R`, `Fn` and `Tuple` shall be complete types, cv void or arrays of unknown bound

7.2.3. `std::is_nothrow_applicable`

Template

```
template <class R, class F, class Tuple>
struct is_nothrow_applicable;
```

Condition
`is_applicable_v<F, Tuple>` is true and the expression `std::apply(std::declval<F>(), std::declval<Tuple>())` is known not to throw any exceptions.

Comments
`Fn` and `Tuple` shall be complete types, cv void or arrays of unknown bound

7.2.4. `std::is_nothrow_applicable_r`

Template

```
template <class R, class F, class Tuple>
struct is_nothrow_applicable_r;
```

Condition
`is_applicable_r_v<F, Tuple>` is true and the expression `std::apply(std::declval<F>(), std::declval<Tuple>())` is known not to throw any exceptions.

Comments
`R`, `Fn` and `Tuple` shall be complete types, cv void or arrays of unknown bound