Reflections on Completion Signatures

Document Number: P4357R0
Date: 2026-09-09
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: SG1, LEWG

Abstract

This paper proposes changes and additions to better integrate `std::meta::info` into the calculation of completion signatures in `std::execution`.

Background

Senders advertise the ways in which they complete via a `get_completion_signatures` static, consteval template member function. This function returns an instance of an instantiation of `std::execution::completion_signatures` whose template arguments indicate the way in which the corresponding asynchronous operation can complete.

When the template arguments used to instantiate a certain `get_completion_signatures` member function are invalid the sender rejects by throwing an exception [1]. Even though such instantiations throw unconditionally (since their throwing-ness is a product of their template arguments) they advertise a return type (despite the fact they'll never generate an instance thereof). In this situation `get_completion_signatures` advertises that it will return `std::exception::completion_signatures<>`.

Discussion

`std::execution::get_completion_signatures`, and, transitively, the `get_completion_signatures` member function of senders, reports the result of the computation it performs in an odd way: Success is reported via the return type, whereas failure is reported by throwing an exception. This is not only awkward, because implementers thereof must advertise a dummy type (e.g. `std::execution::completion_signatures<>`) when they know the computation will fail, but is error prone, because consumers need to ensure the function is actually called.

For support of the above consider the definition of `std::execution::completion_signatures_of_t`:

```
template<class Sndr, class... Env>
    requires sender_in<Sndr, Env...>
```

```
  using completion_signatures_of_t =
    decltype(get_completion_signatures<Sndr, Env...>());
```

One might casually think that the `std::execution::sender_in` constraint is not particularly load-bearing, but this is wrong. Without that constraint `std::execution::completion_signatures_of_t` would erroneously report `std::execution::completion_signatures<>` as the completion signatures of a sender which cannot compute its completion signatures in the given environment. The final constraint of `std::execution::sender_in` therefore becomes incredibly relevant:

```
template<class Sndr, class... Env>
  concept sender_in =
    sender<Sndr> &&
    (sizeof...(Env) <= 1) &&
    (queryable<Env> &&...) &&
    is-constant<get_completion_signatures<Sndr, Env...>()>;
```

Given the manipulation of types required to synthesize the completion signatures of an asynchronous operation C++26 reflection [2] immediately seems to be a good fit for such computation. Integrating with the multimodal communication mechanisms of `std::execution::get_completion_signatures`, however, adds friction.

A naïve attempt to integrate the two protocols might look like this:

```
struct sender {
  using sender_concept = std::execution::sender_tag;
  template<typename Sender, typename... Env>
  static consteval std::meta::info get_completion_signatures_() {
    // ...
  }
  template<typename Sender, typename... Env>
  static consteval auto get_completion_signatures() {
    return typename [:get_completion_signatures_<Sender, Env>():]();
  }
  // ...
};
```

Which actually works in the “happy” case (i.e. where the completion signatures can be computed). In the “unhappy” case, however, the spliced expression throws, is not therefore a constant expression, and compilation fails (rather than propagating the exception as is desired/intended). Therefore the following must be written:

```
struct sender {
  using sender_concept = std::execution::sender_tag;
  template<typename Sender, typename... Env>
```

```
  static consteval std::meta::info get_completion_signatures_() {
    // ...
  }
  template<typename Sender, typename... Env>
  static consteval auto get_completion_signatures() {
    if constexpr ([&]() {
      try {
        (void)get_completion_signatures_<Sender, Env>();
      } catch (...) {
        return false;
      }
      return true;
    }()) {
      return typename [:get_completion_signatures_<Sender, Env>():]();
    } else {
      (void)get_completion_signatures_<Sender, Env>();
      return std::execution::completion_signatures<>{};
    }
  }
  // ...
};
```

Which is outrageous.

Rather than forcing users to write, or perhaps worse understand, code such as the above, the standard library should simply take care of this for them. Therefore this paper proposes:

- Enriching `std::execution::get_completion_signatures` to support `get_completion_signatures` member functions on senders which return `std::meta::info` directly, and
- Adding `std::execution::get_completion_signatures_type` which exposes a `std::meta::info` which reflects the completion signatures of the given sender

Together these provide a bridge from (first bullet) and to (second bullet) reflection-native sender implementations.

Proposal

[execution.syn]

[...]

```
// [exec.getcomplsigs], get completion signatures
template<class Sndr, class... Env>
  consteval auto get_completion_signatures() ->
    valid-completion-signatures auto;

template<class Sndr, class... Env>
    requires sender_in<Sndr, Env...>
  using completion_signatures_of_t =
    decltype(get_completion_signatures<Sndr, Env...>());

// [exec.getcomplsigstiype], get reflection of completion signatures
template<meta::reflection_range Range = std::initializer_list<meta::info>>
  consteval meta::info get_completion_signatures_type(meta::info sndr,
                                                      Range&& range = {});

// [exec.connect], the connect sender algorithm
struct connect_t;
inline constexpr connect_t connect{};
```

[...]

[exec.getcomplsigs]

```
template<class Sndr, class... Env>
  consteval auto get_completion_signatures() ->
    valid-completion-signatures auto;
```

Let *except* be an rvalue subexpression of an unspecified class type *Except* such that `move_constructible<Except> && derived_from<Except, exception>` is `true`. Let *`CHECKED-COMPLSIGS`*`(e)` be `e` if `e` is a core constant expression whose type satisfies *`valid-completion-signatures`*; otherwise, it is the following expression:

```
(e, throw except, completion_signatures())
```

Let *`get-complsigs`*`<Sndr, Env...>()` be expression-equivalent to `remove_reference_t<Sndr>::template get_completion_signatures<Sndr, Env...>()`. Let `NewSndr` be `Sndr` if `sizeof...(Env) == 0` is `true`; otherwise, `decltype(s)` where `s` is the following expression:

```
transform_sender(
  declval<Sndr>(),
  declval<Env>()...)
```

Constraints: `sizeof...(Env) <= 1` is `true`.

Effects: Equivalent to: `return e;` where `e` is expression-equivalent to the following:

- *except* if `sender<Sndr>` is `false`.
- Otherwise, `typename [:get_complsigs<NewSndr, Env...>():]()` if `get_complsigs<NewSndr, Env...>()` is well-formed, has type `meta::info`, and its evaluation does not exit via an exception.
- Otherwise, `(get_complsigs<NewSndr, Env...>(), completion_signatures())` if that expression is well-formed, has type `meta::info`, and its evaluation exits via an exception.
- `typename [:get_complsigs<NewSndr>():]()` if `get_complsigs<NewSndr>()` is well-formed, has type `meta::info`, and its evaluation does not exit via an exception.
- Otherwise, `(get_complsigs<NewSndr>(), completion_signatures())` if that expression is well-formed, has type `meta::info`, and its evaluation exits via an exception.
- Otherwise, `CHECKED-COMPLSIGS(get_complsigs<NewSndr, Env...>())` if `get_complsigs<NewSndr, Env...>()` is a well-formed expression.
- Otherwise, `CHECKED-COMPLSIGS(get_complsigs<NewSndr>())` if `get_complsigs<NewSndr>()` is a well-formed expression.
- Otherwise,
  ```
  completion_signatures<
    SET-VALUE-SIG(await_result_type<NewSndr, env_promise<Env>...>),
    set_error_t(exception_ptr),
    set_stopped_t()>
  ```
  if `is-awaitable<NewSndr, env-promise<Env>...>` is `true`.
- Otherwise, `(throw dependent-sender-error(), completion_signatures())` if `sizeof...(Env) == 0` is `true`, where `dependent-sender-error` is `dependent_sender_error` or an unspecified type derived publicly and unambiguously from `dependent_sender_error`.
- Otherwise, `(throw except, completion_signatures())`.

Given a type `Env`, if `completion_signatures_of_t<Sndr>` and `completion_signatures_of_t<Sndr, Env>` are both well-formed, the former shall be a superset of the latter, with the difference corresponding to error or stopped completion operations.

Let `rcvr` be an rvalue whose type `Rcvr` models `receiver`, and let `Sndr` be the type of a sender such that `sender_in<Sndr, env_of_t<Rcvr>>` is `true`. Let `Sigs...` be the template arguments of the `completion_signatures` specialization named by `completion_signatures_of_t<Sndr, env_of_t<Rcvr>>`. Let `CSO` be a completion function. If sender `Sndr` or its operation state cause the expression `CSO(rcvr, args...)` to be potentially evaluated then there shall be a signature Sig in `Sigs...` such that

`MATCHING-SIG(decayed-typeof<CSO>(decltype(args)...), Sig)`

is `true`.

[exec.getcomplsigstype]

Note: This is a new section.

```
template<meta::reflection_range Range = std::initializer_list<meta::info>>
  consteval meta::info get_completion_signatures_type(meta::info sndr,
                                                      Range&& range = {});
```

Effects: Let *`check-completion-signatures`* be the following exposition-only template function:

```
template <class Sender, class... Env>
  consteval void check-completion-signatures() {
    get_completion_signatures<Sender, Env...>();
  }
```

Equivalent to:

```
vector<meta::info> args{sndr};
args.insert(args.end(), ranges::begin(range), ranges::end(range));
extract<void(*)()>(substitute(^^check-completion-signatures, args))();
return dealias(substitute(^^completion_signatures_of_t, args));
```

Open Design Questions

- If a sender's `get_completion_signatures` member function returns `std::meta::info`, rather than an instantiation of `std::execution::completion_signatures`, should that function instead be required to be named something else (e.g. `get_completion_signatures_type`)?

Implementation Experience

The author has implemented this in Nvidia's stdexec [3].

Acknowledgements

The author would like to thank Daniel Katz for assistance in assessing whether his code was wrong or GCC had a bug, and for advice on designing idiomatic reflection-based APIs.

The author would like to thank Eric Niebler for input on how to better handle non-dependent senders.

References

[1] E. Niebler. High-Quality Sender Diagnostics with Constexpr Exceptions P3557R3
[2] W. Childers et al. Reflection for C++26 P2996R13
[3] https://github.com/NVIDIA/stdexec/pull/2194