D2532R0

Removing exception_ptr from the Receiver Concepts

Draft Proposal, 2022-02-01

Author:
Eric Niebler

Source:
GitHub

Issue Tracking:
GitHub

Project:
ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Audience:
LEWG

Table of Contents

§ 1. Introduction

This paper proposes a refactorization of the receiver concepts of [P2300R4] to address concerns raised by LEWG during its design review related to the requirement of an error channel that accepts `exception_ptr`. The change to `receiver_of` proposed herein enables a corresponding change to the `sender_to` concept that strengthens type checking and removes some need to constrain customizations of the `connect` customization point.

§ 1.1. Motivation

In [P2300R4], the receiver concepts are currently expressed as follows:

```
template <class T, class E = exception_ptr>
concept receiver =
  move_constructible<remove_cvref_t<T>> &&
  constructible_from<remove_cvref_t<T>, T> &&
  requires(remove_cvref_t<T>&& t, E&& e) {
    { execution::set_stopped(std::move(t)) } noexcept;
    { execution::set_error(std::move(t), (E&&) e) } noexcept;
  };

template<class T, class... An>
concept receiver_of =
  receiver<T> &&
  requires(remove_cvref_t<T>&& t, An&&... an) {
    execution::set_value(std::move(t), (An&&) an...);
  };
```

During the design review of P2300, LEWG raised the following concerns about the form of these concepts:

1. Since `set_value` is permitted to be potentially throwing, and since the receiver type is not known when a sender is asked to compute its completion signatures, most senders will need to pessimistically report that they can complete exceptionally, when that may in fact not be true. This may cause the instantiation of expensive error handling code that is effectively dead.
2. No receiver `R` can satisfy the `receiver<R>` or `receiver_of<R, As...>` concepts without providing an error channel for `exception_ptr`. This has the following problems:

— `exception_ptr` is a relatively heavy-weight error type, not unlike a `shared_ptr`. Requiring the presence of this channel is likely to cause needless code generation.

— It makes it questionable whether any of P2300 can be reasonably expected to work in freestanding environments, which often lack exception handling support.

Although the design of P2300 is sound, LEWG nevertheless wanted an investigation into these issues and a recommendation to be made.

This paper makes a recommendation to change the receiver concepts to address these concerns.

§ 1.2. Design Summary

This paper proposes to make the following changes, summarized here without commentary. Commentary is provided below.

— Remove the default implementation of the `get_env` receiver query.

— The `receiver_of` concept takes a receiver and an instance of the `completion_signatures<>` class template.

— A receiver's customization of `set_value` is required to be `noexcept`.

— The `sender_to<Sndr, Rcvr>` concept requires `Rcvr` to accept all of `Sndr`'s completions.

— `connect(sndr, rcvr)` also requires `rcvr` to accept all of `sndr`'s completions.

— `get_completion_signatures` is required to return an instantiation of the `completion_signatures` class template; the `value_types_of_t` and `error_types_of_t` template aliases remain unchanged.

— The `make_completion_signatures` design is slightly tweaked to be more general.

§ 1.3. Design Rationale

The author believes these are all reasonable adjustments to the design of P2300, but one may wonder why they were not considered before now.

The fourth revision of P2300 brought with it some notable changes, the two most significant of which are:

1. Support for dependently-typed senders, where a sender's completions can depend on information that isn't known independently of the execution environment within which the sender will be initiated. For instance, a `get_scheduler()` sender which queries the receiver for the current scheduler and then sends it through the value channel, cannot possibly know the type of the scheduler it will send until it has been connected to a receiver.

2. Dropping of support for "untyped" senders, which do not declare their completion signatures. Untyped senders were supported because of the lack of dependently-typed senders, which ceased to be an issue with R4. At the direction of LEWG, "untyped" senders were dropped, greatly simplifying the design.

Taken together, these two changes open up a huge piece of the design space. The implication is that **a sender is *always* able to provide its completion signatures.** This is new, and P2300R4 is not taking advantage of this extra type information.

The author realized that the extra type information can be leveraged to accommodate LEWGs requests regarding the receiver interface, while at the same time simplifying uses of `std::execution` by permitting the library to take on more of the type checking burden.

The `sender_to` concept, which checks whether a sender and a receiver can be connected, now has perfect information: it can ask the receiver for the execution environment; it can ask the sender how it will complete when initiated in that environment; and it can ask the receiver if it is capable of receiving all of the sender's possible completions. This was not possible before R4.

Below we look at each of the changes suggested in the summary and explain its rationale in light of the extra information now available to the type system.

§ 2. Design Details

§ 2.1. Remove the default implementation of the `get_env` receiver query.

The presence of a customization of `get_env` becomes the distinguishing feature of receivers. A "receiver" no longer needs to provide any completion channels at all to be considered a receiver, only `get_env`.

§ 2.2. The `receiver_of` concept takes a receiver and an instance of the `completion_signatures<>` class template.

The `receiver_of` concept, rather than accepting a receiver and some value types, is changed to take a receiver and an instance of the `completion_signatures<>` class template. A sender uses `completion_signatures<>` to describe the signals with which it completes. The `receiver_of` concept ensures that a particular receiver is capable of receiving those signals.

Notably, if a sender only sends a value (i.e., can never send an error or a stopped signal), then a receiver need only provide a value channel to be compatible with it.

§ 2.3. A receiver's customization of `set_value` is required to be `noexcept`.

This makes it possible for many senders to become "no-fail"; that is, they cannot complete with an error. `just(1)`, for instance, will only ever successfully send an integer through the value channel. An adaptor such as `then(sndr, fun)` can check whether `fun` can ever exit exceptionally when called with all the sets of values that `sndr` may complete with. If so, the `then` sender must add `set_error_t(exception_ptr)` to its list of completions. Otherwise, it need not.

§ 2.4. The `sender_to<Sndr, Rcvr>` concept requires `Rcvr` to accept all of `Sndr`'s completions.

The `sender_to` concept, which checks whether a sender and a receiver can be connected, now enforces that the sender's completion signatures can in fact be handled by the receiver. Previously, it only checked that `connect(sndr, rcvr)` was well-formed, relying on sender authors to properly constrain their `connect` customizations.

§ 2.5. `connect(sndr, rcvr)` also requires `rcvr` to accept all of `sndr`'s completions.

For good measure, the `connect` customization point also checks whether a receiver can receive all of the sender's possible completions before trying to dispatch via `tag_invoke` to a `connect` customization. This often entirely frees sender authors from having to constrain their `connect` customizations at all. It is enough to customize `get_completion_signatures`, and the type checking is done automatically.

Strictly speaking, with this change, the change to `sender_to` is unnecessary. The change to `sender_to` results in better diagnostics, in the author's experience.

§ 2.6. `get_completion_signatures` is required to return an instantiation of the `completion_signatures` class template.

`get_completion_signatures` was added in R4 in response to feedback that authoring sender traits was too difficult/arcane. Rather than defining a struct with `template template` aliases, a user can simply declare a sender's completions as:

```
execution::completion_signatures<
  execution::set_value_t(int),
  execution::set_error_t(std::exception_ptr),
  execution::set_stopped_t()>
```

In R4, `completion_signatures` generated the `template template` aliases for you. The proposed change is to take it further and *require* `get_completion_signatures` to return an instance of the

`completion_signatures` class template. With this change, the last vestige of the old sender traits design with its unloved `temlate template` alias interface is swept away. `completion_signatures` entirely replaces sender traits, further simplifying the design.

The `sender` concept enforces the new requirement.

§ 2.7. The `value_types_of_t` and `error_types_of_t` template aliases remain.

It can still be helpful sometimes to *consume* the old `template template`, say, for generating a variant of the tuples of all the sets of a sender's value types. For that reason, the alias templates `value_types_of_t` and `error_types_of_t` retain the same interface and semantic as before. For instance, generating the variant of tuples of value types, you would use the following:

```
execution::value_types_of_t<
  Sndr,
  Env,
  std::tuple,
  std::variant>;
```

Additionally, these two alias joined by a `sends_stopped<Sndr, Env>` Boolean variable template to complete the set.

§ 2.8. The `make_completion_signatures` design is slightly tweaked to be more general.

In the proposed design, `completion_signatures` plays a much larger role. Accordingly, the job of specifying the completion signatures of custom sender adaptors also becomes more important, necessitating better tools. The `make_completion_signatures`, new to R4, narrowly misses being that better tool.

In R4, `make_completion_signatures` has the following interface:

```
template <
  execution::sender Sndr,
  class Env = execution::no_env,
  class OtherSigs = execution::completion_signatures<>,
  template <class...> class SetValue = default-set-value,
  template <class> class SetError = default-set-error,
  bool SendsStopped = execution::completion_signatures_of_t<Sndr, Env>::sends_s
    requires sender<Sndr, Env>
using make_completion_signatures =
  execution::completion_signatures</* see below */>;
```

In the R4 design, `SetValue` and `SetError` are alias templates, instantiations of which are required to name function types whose return types are `excecution::set_value_t` and `execution::set_error_t`, respectively. This is overly-restrictive. The problems with it are:

1. It is not possible to map one kind of completion into a different kind. For instance, the `upon_error(sndr, fun)` maps error completions into value completions.
2. It is not possible to map a single completion signature into multiple different completions. For instance, the `let_value(sndr, fun)` sender adaptor needs to map a set of `sndr`'s value types into the set of completions of whatever sender that is returned from `fun(values...)`, which is likely more than one.

In addition, the final Boolean `SendsStopped` parameter merely controls whether or not the completion `execution::set_stopped_t()` should be added to the resulting list of completion signatures. This doesn't help a sender adaptor such as `let_stopped(sndr, fun)`, which needs to transform a stopped signal into the set of completions of the sender that `fun()` returns.

This design proposes to change the three final template arguments as follows:

— **`template <class...> class SetValue;`** Instantiations of this alias template must name an instantiation of the `completion_signatures` class template.

— **`template <class> class SetError;`** Instantiations of this alias template must name an instantiation of the `completion_signatures` class template.

— **`class SetStopped;`** Must name an instantiation of the `completion_signatures` class template. If the sender `Sndr` can complete with `set_stopped`, then these signatures are included in the resulting list of completions. Otherwise, this template parameter is ignored.

The semantics of `make_completion_signatures` is likewise simplified: The three template arguments, `SetValue`, `SetError`, and `SetStopped`, are used to map each of a sender's completions into a list of completions which are all concatenated together, along with any additional signatures specified by the `OtherSigs` list, and made unique.

§ 3. Considerations

§ 3.1. Implications of `noexcept set_value`

The role of `execution::set_value` is to execute a continuation on the success of the predecessor. A continuation is arbitrary code, and surely arbitrary code can exit exceptionally, so how can we require `execution::set_value` to be `noexcept`?

The answer has two parts:

1. `execution::set_value` always has the option of accepting arguments by forwarding reference and executing any potentially throwing operations within a `try/catch` block, routing any exceptions to `set_error(std::exception_ptr)`.
2. A sender knows what types it will send and with what value category. The `sender_to` concept checks that none of the `set_value` expression(s) it will execute are potentially throwing. This doesn't necessitate that all receivers accept all arguments by forwarding reference, however. For instance, if a sender knows it will pass an rvalue `std::string` to the receiver's `set_value`, and if the sender is connected to a receiver whose `set_value` takes a `std::string` by value, that will type-check. The `sender_to` concept will essentially be enforcing this constraint:

   ```
   requires (Receiver rcvr) {
     { execution::set_value(std::move(rcvr), std::string()) } noexcept;
   }
   ```

   Since `std::string`'s move constructor is `noexcept`, this constraint is satisfied regardless of whether `rcvr`'s `set_value` customization accepts the string by value or by reference.

§ 3.2. Diagnostics

On the whole, the authors of P2300 feel that this design change is the right one to make to meet LEWG's requirements. It comes with one drawback, however: The satisfaction checking of the `receiver_of` concept, which must now check against a set of signatures specified in a type-list, now requires metaprogramming in addition to `requires` clauses. As a result, diagnostics can suffer.

During the implementation experience, the author was able to surface a relatively suscinct and accurate error for, say, the lack of a particular completion channel on a receiver, by employing several tricks. While regrettable that such tricks are required, we do not feel that the issue of mediocre diagnostics is dire enough to offset the many advantages of the design presented here.

In addition, the author has discovered a way that an implementation may choose to extend the `connect` customization point in a way that permits users to bypass the constraint checking entirely, thus generating a deep instantiation backtrace that often greatly assists the debugging of custom sender/receiver-based algorithms. This mechanism can be enshrined in the standard as "recommended practice."

§ 4. Open questions

§ 4.1. Weasel wording for `-fno-exceptions`

We may need to add some weasel wording to the effect that:

> ... if an implementation is able to deduce that all of its operations are not potentially throwing, a conforming implementation of the algorithms in <section> may omit `set_error_t(exception_ptr)` from any sender's list of completion signatures.

If an implementation doesn't support exceptions, e.g., if the user is compiling with `-fno-exceptions`, it can safely assume that an expression `expr` is not going to exit exceptionally regardless of the value of `noexcept(expr)`. An implementation shouldn't be required to report that it can complete with an exception in that case.

§ 4.2. Error channel of allocating algorithms

An interesting question is what to do on freestanding implementations for those algorithms that necessarily must allocate. Those algorithms, as P2300 stands today, will always have a `set_error_t(exception_ptr)` completion signature. The possibilities I see are:

— Permit implementations to omit the exceptional completion signature when it knows allocations can't fail with an exception (see above),

— Replace the exceptional completion signature with `set_error_t(std::error_code)`, and call the receiver with `std::make_error_code(std::errc::not_enough_memory)` on allocation failure.

— Replace the exceptional completion signature with `set_error_t(std::bad_alloc)`; that is, pass an instance of the `std::bad_alloc` exception type through the error channel by value. (From what the author can infer, freestanding implementations are required to provide the `std::bad_alloc` type even when actually throwing exceptions is not supported.)

§ 5. Implementation experience

The design described above has been implemented in a branch of the reference implementation which can be found in the following GitHub pull request: https://github.com/brycelelbach/wg21_p2300_std_execution/pull/410.

The change, while somewhat disruptive to the reference implementation itself, had the benefits described above; namely:

— Stricter type-checking "for free". Sender authors need only report the completion signatures, and the concepts and customization points of the library do all the heavy lifting to make sure the capabilities of receivers match the requirements of the senders.

— More "no-fail" senders. Many fewer of the senders need an error channel at all, and the ones that do generally need it only conditionally, when working with potentially-thrwoing callables or types whose special operations can throw. Only those few senders that must dynamically allocate state necessarily need a `set_error_t(exception_ptr)` channel, and we may even choose to change those to use something like `set_error_t(bad_alloc)` instead.

— No required `set_error_t(exception_ptr)` or `set_stopped_t()` channels at all.

In addition, in the author's opinion, the reference implementation got significantly *simpler* for the change, and the pull request removes more lines than it adds, while adding functionality at the same time.

§ 6. Proposed wording

The following changes are relative to [P2300R4].

§ 6.1. Header <execution> synopsis

In [exec.syn], apply the following changes:

```
namespace std::execution {
  // [exec.recv], receivers
  template <class T, class E = exception_ptr>
    concept receiver = see-below;

  template <class T, class... An>class Completions>
    concept receiver_of = see-below;

  ...

  template<class S>
    concept has-sender-types = see-below; // exposition only

  ...
  template <class E> // arguments are not associated entities ([lib.tmpl-heads]
    struct dependent_completion_signatures;

  ...
  template<class S,
           class E = no_env,
           template <class...> class Tuple = decayed-tuple,
           template <class...> class Variant = variant-or-empty>
      requires sender<S, E>
    using value_types_of_t =
```

```
  using value_types_of_t =
    typename completion_signatures_of_t<S, S>::template value_types<Tuple, Va
    see below;

template<class S,
         class E = no_env,
         template <class...> class Variant = variant-or-empty>
    requires sender<S, E>
  using error_types_of_t =
    typename completion_signatures_of_t<S, S>::template error_types<Variant>
    see below;

template<class S, class E = no_env>
    requries sender<S, E>
  inline constexpr bool sends_stopped = see below;

...
// [exec.utils.cmplsigs]
template <completion-signature... Fns> // arguments are not associated entiti
  struct completion_signatures {};

template <class... Args> // exposition only
  using default-set-value =
    completion_signatures<set_value_t(Args...)>;

template <class Err> // exposition only
  using default-set-error =
    completion_signatures<set_error_t(Err)>;


template <class Sigs, class E> // exposition only
  concept valid-completion-signatures = see below;

// [exec.utils.mkcmplsigs]
template <
  sender Sndr,
  class Env = no_env,
  classvalid-completion-signatures<Env> AddlSigs = completion_signatures<>,
  template <class...> class SetValue = /* see below */default-set-value,
  template <class> class SetError = /* see below */default-set-error,
  bool SendsStopped = completion_signatures_of_t<Sndr, Env>::sends_stopped>
  valid-completion-signatures<Env> SetStopped = completion_signatures<set_sto
    requires sender
using make_completion_signatures = completion_signatures</* see below */>;
```

§ 6.2. execution::get_env

Change [exec.get_env] as follows:

1. get_env is a customization point object. For some subexpression r, get_env(r) is expression-equivalent to

    1. tag_invoke(execution::get_env, r) if that expression is well-formed.

        — *Mandates:* The decayed type of the above expression is not no_env.

    2. Otherwise, ~~empty_env~~{} get_env(r) is ill-formed .

§ 6.3. Receivers

In [exec.recv], replace paragraphs 1 and 2 with the following:

1. A *receiver* represents the continuation of an asynchronous operation. An asynchronous operation may complete with a (possibly empty) set of values, an error, or it may be cancelled. A receiver has three principal operations corresponding to the three ways an asynchronous operation may complete: set_value, set_error, and set_stopped. These are collectively known as a receiver's *completion-signal operations*.
2. The receiver concept defines the requirements for a receiver type with an unknown set of completion signatures. The receiver_of concept defines the requirements for a receiver type with a known set of completion signatures.

```
template<class T>
  concept receiver =
    move_constructible<remove_cvref_t<T>> &&
    constructible_from<remove_cvref_t<T>, T> &&
    requires(const remove_cvref_t<T>& t) {
      execution::get_env(t);
    };

template <class Signature, class T>
  concept valid-completion-for = // exposition only
    requires (Signature* sig) {
        []<class Ret, class... Args>(Ret(*)(Args...))
            requires nothrow_tag_invocable<Ret, remove_cvref_t<T>, Args...>
        {}(sig);

    };

template <class T, class Completions>
  concept receiver_of =
```

```
    receiver<T> &&
    requires (Completions* completions) {
        []<valid-completion-for<T>...Sigs>(completion_signatures<Sigs...>*)
        {}(completions);
    };
```

6.4. execution::set_value

Change [exec.set_value] as follows:

1. execution::set_value is used to send a *value completion signal* to a receiver.
2. The name execution::set_value denotes a customization point object. The expression execution::set_value(R, Vs...) for some subexpressions R and Vs... is expression-equivalent to:
    1. tag_invoke(execution::set_value, R, Vs...), if that expression is valid. If the function selected by tag_invoke does not send the value(s) Vs... to the receiver R's value channel, the behavior of calling execution::set_value(R, Vs...) is undefined.

        — *Mandates:* The tag_invoke expression above is not potentially throwing.
    2. Otherwise, execution::set_value(R, Vs...) is ill-formed.

6.5. Senders

Change [exec.snd] as follows:

1. A sender describes a potentially asynchronous operation. A sender's responsibility is to fulfill the receiver contract of a connected receiver by delivering one of the receiver completion-signals.
2. The sender concept defines the requirements for a sender type. The sender_to concept defines the requirements for a sender type capable of being connected with a specific receiver type.

```
~~template<template<template<class...> class, template<class...> class> class~~
~~  struct has-value-types; // exposition only~~

~~template<template<template<class...> class> class>~~
~~  struct has-error-types; // exposition only~~

~~template<class S>~~
```

```
  concept has-sender-types = // exposition only
    requires {
      typename has-value-types<S::template value_types>;
      typename has-error-types<S::template error_types>;
      typename bool_constant<S::sends_stopped>;
    };

template <class T, template <class...> class C>
  inline constexpr bool is-instance-of = false; // exposition only

template <class... Ts, template <class...> class C>
  inline constexpr bool is-instance-of<C<Ts...>, C> = true;

template <class Sigs, class E>
  concept valid-completion-signatures = // exposition only
    is-instance-of<Sigs, completion_signatures> ||
    (
      same_as<Sigs, dependent_completion_signatures<no_env>> &&
      same_as<E, no_env>
    );


template <class S, class E>
  concept sender-base = // exposition only
    requires { typename completion_signatures_of_t<S, E>; } &&
    has-sender-types<completion_signatures_of_t<S, E>>
    requires (S&& s, E&& e) {
      { get_completion_signatures(std::forward<S>(s), std::forward<E>(e)) }
        valid-completion-signatures<E>;
    };

template<class S, class E = no_env>
  concept sender =
    sender-base<S, E> &&
    sender-base<S, no_env> &&
    move_constructible<remove_cvref_t<S>>;

template<class S, class R>
  concept sender_to =
    sender<S, env_of_t<R>> &&
    receiver<R> &&
    receiver_of<R, completion_signatures_of_t<S, env_of_t<R>>> &&

    requires (S&& s, R&& r) {
      execution::connect(std::forward<S>(s), std::forward<R>(r));
    };
```

3. The `sender_of` concept defines the requirements for a sender type that on successful completion sends the specified set of value types.

```
template<class S, class E = no_env, class... Ts>
  concept sender_of =
    sender<S, E> &&
    same_as<
      type-list<Ts...>,
      typename completion_signatures_of_t<S, E>::template value_types<type-
      value_types_of_t<S, E, type-list, type_identity_t>
    >;
```

§ 6.6. `execution::completion_signatures_of_t`

Change [exec.sndtraitst]/p4 as follows:

4. `execution::get_completion_signatures` is a customization point object. Let `s` be an expression such that `decltype((s))` is `S`, and let `e` be an expression such that `decltype((e))` is `E`. Then `get_completion_signatures(s)` is expression-equivalent to `get_completion_signatures(s, no_env{})` and `get_completion_signatures(s, e)` is expression-equivalent to:

    1. `tag_invoke_result_t<get_completion_signatures_t, S, E>{}` if that expression is well-formed,

        — *Mandates:* `is-instance-of<Sigs, completion_signatures>` or `is-instance-of<Sigs, dependent_completion_signatures>`, where `Sigs` names the type `tag_invoke_result_t<get_completion_signatures_t, S, E>`.

    2. Otherwise, if `remove_cvref_t<S>::completion_signatures` is well-formed and names a type, then a value-initialized prvalue of type `remove_cvref_t<S>::completion_signatures`,

        — *Mandates:* `is-instance-of<Sigs, completion_signatures>` or `is-instance-of<Sigs, dependent_completion_signatures>`, where `Sigs` names the type `remove_cvref_t<S>::completion_signatures`.

    3. Otherwise, [...]

§ 6.7. `dependent_completion_signatures`

Change [exec.depsndtraits] as follows:

```
template <class E>  // arguments are not associated entities ([lib.tmpl-heads])
  struct dependent_completion_signatures {};
```

1. `dependent_completion_signatures` is a placeholder completion signatures descriptor that can be ~~used~~ returned from `get_completion_signatures` to report that a type might be a sender within a particular execution environment, but it isn't a sender in an arbitrary execution environment.

2. ~~If `decay_t<E>` is `no_env`, `dependent_completion_signatures<E>` is equivalent to:~~

   ```
   template <>
     struct dependent_completion_signatures<no_env> {
       template <template <class...> class, template <class...> class>
         requires false
       using value_types = /* unspecified */;

       template <template <class...> class>
         requires false
       using error_types = /* unspecified */;

       static constexpr bool sends_stopped = /* unspecified */;
     };
   ```

   ~~Otherwise, `dependent_completion_signatures<E>` is an empty struct.~~

2. When used as the return type of a customization of `get_completion_signatures`, the template argument `E` shall be the unqualified type of the second argument.

§ 6.8. `execution::connect`

Change [exec.connect]/p2 as follows:

2. The name `execution::connect` denotes a customization point object. For some subexpressions `s` and `r`, let `S` be `decltype((s))` and `R` be `decltype((r))`, and let `S'` and `R'` be the decayed types of `S` and `R`, respectively. If `R` does not satisfy `execution::receiver`, `execution::connect(s, r)` is ill-formed. Otherwise, the expression `execution::connect(s, r)` is expression-equivalent to:

1. `tag_invoke(execution::connect, s, r)`, if ~~that expression is valid and `S` satisfies `execution::sender`~~ the constraints below are satisfied . If the function selected by `tag_invoke` does not return an operation state for which `execution::start` starts work described by `s`, the behavior of calling `execution::connect(s, r)` is undefined.

   — *Constraints:*

   ```
   sender<S, env_of_t<R>> &&
   receiver_of<R, completion_signatures_of_t<S, env_of_t<R>>> &&
   tag_invocable<connect_t, S, R>
   ```

   — *Mandates:* The type of the `tag_invoke` expression above satisfies `operation_state`.

2. Otherwise, *`connect-awaitable`*`(s, r)` if [...]

   [...]

   ~~The operand of the *requires-clause* of *`connect-awaitable`* is equivalent to `receiver_of<R>` if *`await-result-type`*`<S, `*`connect-awaitable-promise`*`>` is `cv void`; otherwise, it is `receiver_of<R, `*`await-result-type`*`<S, `*`connect-awaitable-promise`*`>>`.~~
   Let `Res` be *`await-result-type`*`<S, `*`connect-awaitable-promise`*`>`, and let `Vs...` be an empty parameter pack if `Res` is `cv void`, or a pack containing the single type `Res` otherwise. The operand of the *requires-clause* of *`connect-awaitable`* is equivalent to `receiver_of<R, Sigs>` where `Sigs` names the type:

   ```
   completion_signatures<
     set_value_t(Vs...),
     set_error_t(exception_ptr),
     set_stopped_t()>
   ```

3. Otherwise, `execution::connect(s, r)` is ill-formed.

§ 6.9. `execution::just`

Change [exec.just] as follows:

1. `execution::just` is used to create a sender that propagates a set of values to a connected receiver.

   ```
   template<class... Ts>
   struct just-sender { // exposition only
   ```

```
  : completion_signatures<set_value_t(Ts...), set_error_t(exception_ptr)> {
  using completion_signatures =
    execution::completion_signatures<set_value_t(Ts...)>;

  tuple<Ts...> vs_;

  template<class R>
  struct operation_state {
    tuple<Ts...> vs_;
    R r_;

    friend void tag_invoke(start_t, operation_state& s) noexcept {
      try {
        apply([&s](Ts&... values_) {
          set_value(std::move(s.r_), std::move(values_)...);
        }, s.vs_);
      }
      catch (...) {
        set_error(std::move(s.r_), current_exception());
      }
    }
  };

  template<receiver_of<completion_signatures> R>
    requires receiver_of<R, Ts...> && (copy_constructible<Ts> &&...)
  friend operation_state<decay_t> tag_invoke(connect_t, const just-sender&
    return { j.vs_, std::forward<R>(r) };
  }

  template<receiver_of<completion_signatures> R>
    requires receiver_of<R, Ts...>
  friend operation_state<decay_t> tag_invoke(connect_t, just-sender&& j, R
    return { std::move(j.vs_), std::forward<R>(r) };
  }
};

template<movable-value... Ts>
  just-sender<decay_t<Ts>...> just(Ts &&... ts) noexcept(see-below);
```

2. *Effects*: [...]

§ 6.10. execution::just_error

Change [exec.just_error] as follows:

1. `execution::just_error` is used to create a sender that propagates an error to a connected receiver.

```
template<class T>
struct just-error-sender { // exposition only
  : completion_signatures<set_error_t(T)> {
  using completion_signatures =
    execution::completion_signatures<set_error_t(T)>;

  T err_;

  template<class R>
  struct operation_state {
    T err_;
    R r_;

    friend void tag_invoke(start_t, operation_state& s) noexcept {
      set_error(std::move(s.r_), std::move(err_));
    }
  };

  template<receiver_of<completion_signatures> R>
    requires receiver<R, T> && copy_constructible<T>
  friend operation_state<decay_t<R>> tag_invoke(connect_t, const just-error
    return { j.err_, std::forward<R>(r) };
  }

  template<receiver_of<completion_signatures> R>
    requires receiver<R, T>
  friend operation_state<decay_t<R>> tag_invoke(connect_t, just-error-sende
    return { std::move(j.err_), std::forward<R>(r) };
  }
};

template<movable-value T>
  just-error-sender<decay_t<T>> just_error(T&& t) noexcept(see-below);
```

2. *Effects*: [...]

§ 6.11. `execution::just_stopped`

Change [exec.just_stopped] as follows:

1. `execution::just_stopped` is used to create a sender that propagates a stopped signal to a connected receiver.

```
struct just-stopped-sender { // exposition only
  : completion_signatures<set_stopped_t()> {
  using completion_signatures =
    execution::completion_signatures<set_stopped_t()>;

  template<class R>
  struct operation_state {
    R r_;

    friend void tag_invoke(start_t, operation_state& s) noexcept {
      set_stopped(std::move(s.r_));
    }
  };

  template<receiver_of<completion_signatures> R>
  friend operation_state<decay_t<R>> tag_invoke(connect_t, const just-stopp
    return { std::forward<R>(r) };
  }
};

just-stopped-sender just_stopped() noexcept;
```

2. *Effects*: Equivalent to *just-stopped-sender*{}.

§ 6.12. execution::read

Change [exec.read]/p3 as follows:

3. *read-sender* is an exposition only class template equivalent to:

```
template <class Tag>
  struct read-sender { // exposition only
    template<class R>
      struct operation-state { // exposition only
        R r_;

        friend void tag_invoke(start_t, operation-state& s) noexcept try {
          auto value = Tag{}(get_env(s.r_));

          set_value(std::move(s.r_), std::move(value));
        } catch(...) {
```

```
        set_error(std::move(s.r_), current_exception());
      }
    };

  template <class Env>
      requires callable<Tag, Env>
    using completions = // exposition only
      completion_signatures<
        set_value_t(call-result-t<Tag, Env>), set_error_t(exception_ptr)>

  template<receiver R>
    requires callable<Tag, env_of_t<R>> &&
      receiver_of<R, call-result-t<Tag, env_of_t<R>>>
  template<class R>
    requires receiver_of<R, completions<env_of_t<R>>>
  friend operation-state<decay_t<R>> tag_invoke(connect_t, read-sender, R
    return { std::forward<R>(r) };
  }

  friend empty-env tag_invoke(get_completion_signatures_t, read-sender, a
  template<class Env>
    friend auto tag_invoke(get_completion_signatures_t, read-sender, Env)
      -> dependent_completion_signatures<Env>;

  template<class Env>
    requires (!same_as<Env, no_env>) && callable<Tag, Env>
    friend auto tag_invoke(get_completion_signatures_t, read-sender, Env)
      -> completion_signatures<
          set_value_t(call-result-t<Tag, Env>), set_error_t(exception_ptr
      -> completions<Env> requires true;
};
```

§ 6.13. execution::schedule_from

Replace [exec.schedule_from]/3.3, which begins with "Given an expression `e`, let `E` be `decltype((e))`," with the following:

3. Given subexpressions `s2` and `e`, where `s2` is a sender returned from `schedule_from` or a copy of such, let `S2` be `decltype((s2))` and let `E` be `decltype((e))`. Then the type of `tag_invoke(get_completion_signatures, s2, e)` shall be:

```
make_completion_signatures<
  copy_cvref_t<S2, S>,
  E,
  make_completion_signatures<
    schedule_result_t<Sch>,
    E,
    completion_signatures<set_error_t(exception_ptr)>,
    no-value-completions>>;
```

where *no-value-completions*<As...> names the type completion_signatures<> for any set of types As....

§ 6.14. execution::then

Replace [exec.then]/p2.3.3, which begins with "Given an expression e, let E be decltype((e))," with the following:

3. Let *compl-sig-t*<Tag, Args...> name the type Tag() if Args... is a template paramter pack containing the single type void; otherwise, Tag(Args...). Given subexpressions s2 and e where s2 is a sender returned from then or a copy of such, let S2 be decltype((s2)) and let E be decltype((e)). The type of tag_invoke(get_completion_signatures, s2, e) shall be equivalent to:

```
make_completion_signatures<
  copy_cvref_t<S2, S>, E, set-error-signature,
    set-value-completions>;
```

where *set-value-completions* is an alias for:

```
template <class... As>
  set-value-completions =
    completion_signatures<compl-sig-t<set_value_t, invoke_result_t<F, As...
```

and *set-error-signature* is an alias for completion_signatures<set_error_t(exception_ptr)> if any of the types in the *type-list* named by value_types_of_t<copy_cvref_t<S2, S>, E, *potentially-throwing*, *type-list*> are true_type; otherwise, completion_signatures<>, where *potentially-throwing* is the template alias:

```
template <class... As>
  potentially-throwing =
    bool_constant<is_nothrow_invocable_v<F, As...>>;
```

§ 6.15. execution::upon_error

Replace [exec.upon_error]/p2.3.3, which begins with "Given an expression `e`, let `E` be `decltype((e))`," with the following:

3. Let *`compl-sig-t`*`<Tag, Args...>` name the type `Tag()` if `Args...` is a template paramter pack containing the single type `void`; otherwise, `Tag(Args...)`. Given subexpressions `s2` and `e` where `s2` is a sender returned from `upon_error` or a copy of such, let `S2` be `decltype((s2))` and let `E` be `decltype((e))`. The type of `tag_invoke(get_completion_signatures, s2, e)` shall be equivalent to:

```
make_completion_signatures<
  copy_cvref_t<S2, S>, E, set-error-signature,
    default-set-value, set-error-completion>;
```

where *`set-error-completion`* is the template alias:

```
template <class E>
  set-error-completion =
    completion_signatures<compl-sig-t<set_value_t, invoke_result_t<F, E>>>
```

and *`set-error-signature`* is an alias for `completion_signatures<set_error_t(exception_ptr)>` if any of the types in the *`type-list`* named by `error_types_of_t<copy_cvref_t<S2, S>, E, potentially-throwing>` are `true_type`; otherwise, `completion_signatures<>`, where *`potentially-throwing`* is the template alias:

```
template <class... Es>
  potentially-throwing =
    type-list<bool_constant<is_nothrow_invocable_v<F, Es>>...>;
```

§ 6.16. execution::upon_stopped

Replace [exec.upon_stopped]/p2.3.3, which begins "Given some expression `e`, let `E` be `decltype((e))`," with the following:

3. Let *`compl-sig-t`*`<Tag, Args...>` name the type `Tag()` if `Args...` is a template paramter pack containing the single type `void`; otherwise, `Tag(Args...)`. Given subexpressions `s2` and `e` where `s2` is a sender returned from `upon_stopped` or a copy of such, let `S2` be `decltype((s2))` and let `E` be `decltype((e))`. The type of `tag_invoke(get_completion_signatures, s2, e)` shall be equivalent to:

```
make_completion_signatures<
  copy_cvref_t<S2, S>, E, set-error-signature,
    default-set-value, default-set-error, set-stopped-completions>;
```

where *`set-stopped-completions`* names the type `completion_signatures<`*`compl-sig-t`*`<set_value_t, invoke_result_t<F>>>`, and *`set-error-signature`* names the type `completion_signatures<set_error_t(exception_ptr)>` if `is_nothrow_invocable_v<F>` is `true`, or `completion_signatures<>` otherwise.

§ 6.17. `execution::bulk`

Replace [exec.bulk]/p2.4, which begins, "Given an expression `e`, let `E` be `decltype((e))`," with the following:

4. Given subexpressions `s2` and `e` where `s2` is a sender returned from `bulk` or a copy of such, let `S2` be `decltype((s2))` and let `E` be `decltype((e))`. The type of `tag_invoke(get_completion_signatures, s2, e)` shall be equivalent to:

```
make_completion_signatures<
  copy_cvref_t<S2, S>, E, completion_signatures<set_error_t(exception_ptr)>
```

§ 6.18. `execution::split`

Replace [exec.split]/p3.4, which begins, "Given an expression `e`, let `E` be `decltype((e))`," with the following:

4. Given subexpressions `s2` and `e` where `s2` is a sender returned from `split` or a copy of such, let `S2` be `decltype((s2))` and let `E` be `decltype((e))`. The type of `tag_invoke(get_completion_signatures, s2, e)` shall be equivalent to:

```
make_completion_signatures<
```

```
    copy_cvref_t<S2, S>, E, completion_signatures<set_error_t(exception_ptr)>
      value-signatures, error-signatures>;
```

where *value-signatures* is the alias template:

```
template <class... Ts>
  using value-signatures =
    completion_signatures<set_value_t(decay_t<Ts>&...)>;
```

and *error-signatures* is the alias template:

```
template <class E>
  using error-signatures =
    completion_signatures<set_error_t(decay_t<E>&)>;
```

§ 6.19. execution::when_all

Replace [exec.when_all]/p2.2.5, which begins, "Given some expression `e`, let `E` be `decltype((e))`," with the following:

5. Given subexpressions `s2` and `e` where `s2` is a sender returned from `when_all` or a copy of such, let `S2` be `decltype((s2))`, let `E` be `decltype((e))`, and let `Ss...` be the decayed types of the arguments to the `when_all` expression that created `s2`. If the decayed type of `e` is `no_env`, let `WE` be `no_env`; otherwise, let `WE` be a type such that `stop_token_of_t<WE>` is `in_place_stop_token` and `tag_invoke_result_t<Tag, WE, As...>` names the type, if any, of *`call-result-t`*`<Tag, E, As...>` for all types `As...` and all types `Tag` besides `get_stop_token_t`. The type of `tag_invoke(get_completion_signatures, s2, e)` shall be as follows:

    1. For each type S_i in `Ss...`, let S'_i name the type `copy_cvref_t<S2, `S_i`>`. If for any type S'_i, the type `completion_signatures_of_t<`S'_i`, WE>` names a type other than an instantiation of `completion_signatures`, the type of `tag_invoke(get_completion_signatures, s2, e)` shall be `dependent_completion_signatures<E>`.
    2. Otherwise, for each type S'_i, let $Sigs_i$`...` be the set of template arguments in the instantiation of `completion_signatures` named by `completion_signatures_of_t<`S'_i`, WE>`, and let C_i be the count of function types in $Sigs_i$`...` for which the return type is `set_value_t`. If any C_i is two or greater, then the type of

`tag_invoke(get_completion_signatures, s2, e)` shall be `dependent_completion_signatures<E>`.

3. Otherwise, let $Sigs2_i...$ be the set of function types in $Sigs_i...$ whose return types are *not* `set_value_t`, and let `Ws...` be the unique set of types in $[Sigs2_0..., Sigs2_1..., ... Sigs2_{n-1}..., set\_error\_t(exception\_ptr), set\_stopped\_t()]$, where n is `sizeof...(Ss)`. If any C_i is `0`, then the type of `tag_invoke(get_completion_signatures, s2, e)` shall be `completion_signatures<Ws...>`.
4. Otherwise, let $V_i...$ be the function argument types of the single type in $Sigs_i...$ for which the return type is `set_value_t`. Then the type of `tag_invoke(get_completion_signatures, s2, e)` shall be $completion\_signatures<Ws..., set\_value\_t(V_0..., V_1..., ... V_{n-1}...)>$.

§ 6.20. `execution::ensure_started`

Replace [exec.ensure_started]/p2.4 which begins, "Given an expression `e`, let `E` be `decltype((e))`," with the following:

4. Given subexpressions `s2` and `e` where `s2` is a sender returned from `ensure_started` or a copy of such, let `S2` be `decltype((s2))` and let `E` be `decltype((e))`. The type of `tag_invoke(get_completion_signatures, s2, e)` shall be equivalent to:

```
make_completion_signatures<
  copy_cvref_t<S2, S>,
  ensure-started-env,
  completion_signatures<set_error_t(exception_ptr&&)>,
  set-value-signature,
  error-types>
```

where *`set-value-signature`* is the alias template:

```
template <class... Ts>
  using set-value-signature =
    completion_signatures<set_value_t(decay_t<Ts>&&...)>;
```

and *`error-types`* is the alias template:

```
template <class E>
  using error-types =
    completion_signatures<set_error_t(decay_t<E>&&)>;
```

§ 6.21. `execution::start_detached`

Change [exec.start_detached]p2.3 as follows:

3. Otherwise:

    1. ~~Constructs a receiver `r`~~ Let `R` be the type of a receiver, let `r` be an rvalue of type `R`, and let `cr` be a lvalue reference to `const R` such that :
        1. ~~When `set_value(r, ts...)` is called, it does nothing.~~ The expression `set_value(r)` is not potentially throwing and has no effect,
        2. ~~When `set_error(r, e)` is called, it calls `std::terminate`.~~ For any subexpression `e`, the expression `set_error(r, e)` is expression-equivalent to `terminate()`,
        3. ~~When `set_stopped(r)` is called, it does nothing.~~ The expression `set_stopped(r)` is not potentially throwing and has no effect, and
        4. The expression `get_env(cr)` is expression-equivalent to *`empty-env`*`{}`.
    2. Calls `execution::connect(s, r)`, resulting in an operation state `op_state`, then calls `execution::start(op_state)`. The lifetime of `op_state` lasts until one of the receiver completion-signals of `r` is called.

§ 6.22. `this_thread::sync_wait`

Change [exec.sync_wait]/p4.3.3.1 as follows:

1. If `execution::set_value(r, ts...)` has been called, returns *`sync-wait-type`*`<S, `*`sync-wait-env`*`>{`*`decayed-tuple`*`<decltype(ts)...>{ts...}}`. If that expression exits exceptionally, the exception is propagated to the caller of `sync_wait`.

§ 6.23. `execution::receiver_adaptor`

Remove [exec.utils.rcvr_adptr]/p2, which begins, "This section makes use of the following exposition-only entities," and renumber all subsequent paragraphs.

Change [exec.utils.rcvr_adptr]/p4-6 (now p3-5) as follows:

3. `receiver_adaptor<Derived, Base>` is equivalent to the following:

```
template <
  class-type Derived,
  receiver Base = unspecified> // arguments are not associated entities ([l
class receiver_adaptor {
  friend Derived;
```

```
 public:
  // Constructors
  receiver_adaptor() = default;
  template <class B>
      requires HAS-BASE && constructible_from<Base, B>
    explicit receiver_adaptor(B&& base) : base_(std::forward<B>(base)) {}

 private:
  using set_value = unspecified;
  using set_error = unspecified;
  using set_stopped = unspecified;
  using get_env = unspecified;

  // Member functions
  template <class Self>
    requires HAS-BASE
  copy_cvref_t<Self, Base>&&decltype(auto) base(this Self&& self) noexcept
    return static_cast<Self&&>(self).base_;
    return (std::forward<Self>(self).base_);
  }

  // [exec.utils.rcvr_adptr.nonmembers] Non-member functions
  template <class D = Derived, class... As>
    friend void tag_invoke(set_value_t, Derived&& self, As&&... as) noexcep

  template <class E, class D = Derived>
    friend void tag_invoke(set_error_t, Derived&& self, E&& e) noexcept;

  template <class D = Derived>
  friend void tag_invoke(set_stopped_t, Derived&& self) noexcept;

  friend decltype(auto) tag_invoke(get_env_t, const Derived& self)
      noexcept(see below);

  template <forwarding-receiver-query Tag, class D = Derived, class... As>
      requires callable<Tag, BASE-TYPE(const Derived&), As...>
    friend auto tag_invoke(Tag tag, const Derived& self, As&&... as)
      noexcept(nothrow-callable<Tag, BASE-TYPE(const Derived&), As...>)
      -> call-result-t<Tag, BASE-TYPE(const Derived&), As...> {
      return std::move(tag)(GET-BASE(self), std::forward<As>(as)...);
    }

  [[no_unique_address]] Base base_; // present if and only if HAS-BASE is t
};
```

4. [*Note:* `receiver_adaptor` provides `tag_invoke` overloads on behalf of the derived class `Derived`, which is incomplete when `receiver_adaptor` is instantiated.]

5. [*Example:*

```
using _int_completion =
  execution::completion_signatures<execution::set_value_t(int)>;

template <execution::receiver_of<~~int~~_int_completion> R>
  class my_receiver : execution::receiver_adaptor<my_receiver<R>, R> {
    friend execution::receiver_adaptor<my_receiver, R>;
    void set_value() && {
      execution::set_value(std::move(*this).base(), 42);
    }
   public:
    using execution::receiver_adaptor<my_receiver, R>::receiver_adaptor;
  };
```

-- *end example*]

Replace section [exec.utils.rcvr_adptr.nonmembers] with the following:

```
template <class... As>
  friend void tag_invoke(set_value_t, Derived&& self, As&&... as) noexcept;
```

1. Let SET-VALUE be the expression `std::move(self).set_value(std::forward<As>(as)...)`.
2. *Constraints:* Either SET-VALUE is a valid expression or `typename Derived::set_value` denotes a type and *callable*`<set_value_t, `*BASE-TYPE*`(Derived), As...>` is `true`.
3. *Mandates:* SET-VALUE, if that expression is valid, is not potentially throwing.
4. *Effects:* Equivalent to:

   — If SET-VALUE is a valid expression, SET-VALUE;

   — Otherwise, `execution::set_value(`*GET-BASE*`(std::move(self)), std::forward<As>(as)...)`.

```
template <class E>
  friend void tag_invoke(set_error_t, Derived&& self, E&& e) noexcept;
```

1. Let SET-ERROR be the expression `std::move(self).set_error(std::forward<E>(e))`.

2. *Constraints:* Either `SET-ERROR` is a valid expression or `typename Derived::set_error` denotes a type and *`callable`*`<set_error_t, `*`BASE-TYPE`*`(Derived), E>` is `true`.
3. *Mandates:* `SET-ERROR`, if that expression is valid, is not potentially throwing.
4. *Effects:* Equivalent to:

   — If `SET-ERROR` is a valid expression, `SET-ERROR`;

   — Otherwise, `execution::set_error(`*`GET-BASE`*`(std::move(self)), std::forward<E>(e))`.

```
friend void tag_invoke(set_stopped_t, Derived&& self) noexcept;
```

1. Let `SET-STOPPED` be the expression `std::move(self).set_stopped()`.
2. *Constraints:* Either `SET-STOPPED` is a valid expression or `typename Derived::set_stopped` denotes a type and *`callable`*`<set_stopped_t, `*`BASE-TYPE`*`(Derived)>` is `true`.
3. *Mandates:* `SET-STOPPED`, if that expression is valid, is not potentially throwing.
4. *Effects:* Equivalent to:

   — If `SET-STOPPED` is a valid expression, `SET-STOPPED`;

   — Otherwise, `execution::set_stopped(`*`GET-BASE`*`(std::move(self)))`.

```
friend decltype(auto) tag_invoke(get_env_t, const Derived& self)
  noexcept(see below);
```

1. *Constraints:* Either `self.get_env()` is a valid expression or `typename Derived::get_env` denotes a type and *`callable`*`<get_env_t, `*`BASE-TYPE`*`(const Derived&)>` is `true`.
2. *Effects:* Equivalent to:

   — If `self.get_env()` is a valid expression, `self.get_env()`;

   — Otherwise, `execution::get_env(`*`GET-BASE`*`(self))`.
3. *Remarks:* The expression in the `noexcept` clause is:

   — If `self.get_env()` is a valid expression, `noexcept(self.get_env())`;

   — Otherwise, `noexcept(execution::get_env(`*`GET-BASE`*`(self)))`.

§ 6.24. `execution::completion_signatures`

Change [exec.utils.cmplsigs] as follows:

1. ~~`completion_signatures` is used to define a type that implements the nested `value_types`, `error_types`, and `sends_stopped` members that describe the ways a sender completes. Its arguments are a flat list of function types that describe the signatures of the receiver's completion-signal operations that the sender invokes.~~
   `completion_signatures` is used to describe the completion signals of a receiver that a sender may invoke. Its template argument list is a list of function types corresponding to the signatures of the receiver's completion signals.

2. [*Example:*

```
class my_sender {
  using completion_signatures =
    execution::completion_signatures<
      execution::set_value_t(),
      execution::set_value_t(int, float),
      execution::set_error_t(exception_ptr),
      execution::set_error_t(error_code),
      execution::set_stopped_t()>;
};

// completion_signatures_of_t<my_sender>
//       ::value_types<tuple, variant> names the type:
//   variant<tuple<>, tuple<int, float>>
//
// completion_signatures_of_t<my_sender>
//       ::error_types<variant> names the type:
//   variant<exception_ptr, error_code>
//
// completion_signatures_of_t<my_sender>::sends_stopped is true
// Declares my_sender to be a sender that can complete by calling
// one of the following for a receiver expression R:
//    execution::set_value(R)
//    execution::set_value(R, int{...}, float{...})
//    execution::set_error(R, exception_ptr{...})
//    execution::set_error(R, error_code{...})
//    execution::set_stopped(R)
```

   -- *end example*]

3. This section makes use of the following exposition-only concept:

```
template <class Fn>
  concept completion-signature = see below;
```

1. A type `Fn` satisfies *`completion-signature`* if it is a function type with one of the following forms:

   — `set_value_t(`*`Vs`*`...)`, where *`Vs`* is an arbitrary parameter pack.

   — `set_error_t(`*`E`*`)`, where *`E`* is an arbitrary type.

   — `set_stopped_t()`

2. Otherwise, `Fn` does not satisfy *`completion-signature`*.

4. ~~`template <`*`completion-signature`*`... Fns> // arguments are not associated ent`~~
   `struct completion_signatures {};`
   ~~`template <template <class...> class Tuple, template <class...> class Va`~~
   ~~`using value_types = `*`see below`*`;`~~

   ~~`template <template <class...> class Variant>`~~
   ~~`using error_types = `*`see below`*`;`~~

   ~~`static constexpr bool sends_stopped = `*`see below`*`;`~~
   ~~`};`~~

   1. ~~Let *`ValueFns`* be a template parameter pack of the function types in `Fns` whose return types are `execution::set_value_t`, and let V_n be a template parameter pack of the function argument types in the *n*-th type in *`ValueFns`*. Then, given two variadic templates *`Tuple`* and *`Variant`*, the type `completion_signatures<Fns...>::value_types<`*`Tuple, Variant`*`>` names the type *`Variant`*`<`*`Tuple`*`<`V_0`...>, `*`Tuple`*`<`V_1`...>, ... `*`Tuple`*`<`V_{m-1}`...>>`, where *m* is the size of the parameter pack *`ValueFns`*.~~

   2. ~~Let *`ErrorFns`* be a template parameter pack of the function types in `Fns` whose return types are `execution::set_error_t`, and let E_n be the function argument type in the *n*-th type in *`ErrorFns`*. Then, given a variadic template *`Variant`*, the type `completion_signatures<Fns...>::error_types<`*`Variant`*`>` names the type *`Variant`*`<`E_0`, `*`Error`*$_1$`, ... `*`Error`*$_{m-1}$`>`, where *m* is the size of the parameter pack *`ErrorFns`*.~~

   3. ~~`completion_signatures<Fns...>::sends_stopped` is `true` if at least one of the types in `Fns` is `execution::set_stopped_t()`; otherwise, `false`.~~

5. ```
   template<class S,
            class E = no_env,
            template <class...> class Tuple = decayed-tuple,
            template <class...> class Variant = variant-or-empty>
     requires sender<S, E>
   using value_types_of_t = see below;
   ```
   ```

— Let `Fns...` be a template parameter pack of the arguments of the `completion_signatures` instantiation named by `completion_signatures_of_t<S, E>`, let *`ValueFns`* be a template parameter pack of the function types in `Fns` whose return types are `execution::set_value_t`, and let $Values_n$ be a template parameter pack of the function argument types in the *n*-th type in *`ValueFns`*. Then, given two variadic templates *`Tuple`* and *`Variant`*, the type `value_types_of_t<S, E, Tuple, Variant>` names the type *`Variant`*`<`*`Tuple`*`<`$Values_0$`...>, `*`Tuple`*`<`$Values_1$`...>, ... `*`Tuple`*`<`$Values_{m-1}$`...>>`, where *m* is the size of the parameter pack *`ValueFns`*.

6.
```
template<class S,
         class E = no_env,
         template <class...> class Variant = variant-or-empty>
  requires sender<S, E>
using error_types_of_t = see below;
```

— Let `Fns...` be a template parameter pack of the arguments of the `completion_signatures` instantiation named by `completion_signatures_of_t<S, E>`, let *`ErrorFns`* be a template parameter pack of the function types in `Fns` whose return types are `execution::set_error_t`, and let $Error_n$ be the function argument type in the *n*-th type in *`ErrorFns`*. Then, given a variadic template *`Variant`*, the type `error_types_of_t<S, E, Variant>` names the type *`Variant`*`<`$Error_0$`, `$Error_1$`, ... `$Error_{m-1}$`>`, where *m* is the size of the parameter pack *`ErrorFns`*.

7.
```
template<class S, class E = no_env>
    requires sender<S, E>
inline constexpr bool sends_stopped = see below;
```

— Let `Fns...` be a template parameter pack of the arguments of the `completion_signatures` instantiation named by `completion_signatures_of_t<S, E>`. `sends_stopped<S, E>` is `true` if at least one of the types in `Fns` is `execution::set_stopped_t()`; otherwise, `false`.

§ 6.25. `execution::make_completion_signatures`

Change [exec.utils.mkcmplsigs] as follows:

1. `make_completion_signatures` is an alias template used to adapt the completion signatures of a sender. It takes a sender, and environment, and several other template arguments that apply modifications to the sender's completion signatures to generate a new instantiation of `execution::completion_signatures`.

2. [*Example:*

```
// Given a sender S and an environment Env, adapt a S's completion
// signatures by lvalue-ref qualifying the values, adding an additional
// exception_ptr error completion if its not already there, and leaving the
// other signals alone.
template <class... Args>
  using my_set_value_t =
    execution::completion_signatures<
      execution::set_value_t(add_lvalue_reference_t<Args>...)>;

using my_completion_signals =
  execution::make_completion_signatures<
    S, Env,
    execution::completion_signatures<execution::set_error_t(exception_ptr)>
    my_set_value_t>;
```

-- *end example*]

3. This section makes use of the following exposition-only entities:

```
template <class... As>
  using default-set-value =
    execution::completion_signatures<execution::set_value_t(As...)>;

template <class Err>
  using default-set-error =
    execution::completion_signatures<execution::set_error_t(Err)>;
```

4.
```
template <
  execution::sender Sndr,
  class Env = execution::no_env,
  class valid-completion-signatures<Env> AddlSigs = execution::completion_si
  template <class...> class SetValue = default-set-value,
  template <class> class SetError = default-set-error,
  bool SendsStopped = execution::completion_signatures_of_t<Sndr, Env>::sen
  valid-completion-signatures<Env> SetStopped =
      execution::completion_signatures<set_stopped_t()>>
    requires sender<Sndr, Env>
using make_completion_signatures =
  execution::completion_signatures</* see below */>;
```

— AddlSigs shall name an instantiation of the execution::completion_signatures class template.

— `SetValue` shall name an alias template such that for any template parameter pack `As...`, the type `SetValue<As...>` is either ill-formed ~~, `void` or an alias for a function type whose return type is `execution::set_value_t`~~ or else *`valid-completion-signatures`*`<SetValue<As...>, E>` is satisfied .

— `SetError` shall name an alias template such that for any type `Err`, `SetError<Err>` is either ill-formed ~~, `void` or an alias for a function type whose return type is `execution::set_error_t`~~ or else *`valid-completion-signatures`*`<SetError<Err>, E>` is satisfied .

Then:

— Let `Vs...` be a pack of the ~~non-`void`~~ types in the *`type-list`* named by `value_types_of_t<Sndr, Env, SetValue, type-list>`.

— Let `Es...` be a pack of the ~~non-`void`~~ types in the *`type-list`* named by `error_types_of_t<Sndr, Env, error-list>`, where *`error-list`* is an alias template such that *`error-list`*`<Ts...>` names *`type-list`*`<SetError<Ts>...>`.

— Let `Ss` ~~be an empty pack if `SendsStopped` is `false`; otherwise, a pack containing the single type `execution::set_stopped_t()`~~ name the type `completion_signatures<>` if `sends_stopped<Sndr, Env>` is `false`; otherwise, `SetStopped` .

Then:

7. ~~Let `MoreSigs...` be a pack of the template arguments of the `execution::completion_signatures` instantiation named by `AddlSigs`.~~

8. ~~If any of the above types are ill-formed, then `make_completion_signatures<Sndr, Env, AddlSigs, SetValue, SetDone, SendsStopped>` is an alias for `dependent_completion_signatures<Env>`.~~

9. ~~Otherwise, `make_completion_signatures<Sndr, Env, AddlSigs, SetValue, SetDone, SendsStopped>` names the type `completion_signatures<Sigs...>` where `Sigs...` is the unique set of types in `[Vs..., Es..., Ss..., MoreSigs...]`.~~

1. If any of the above types are ill-formed, then `make_completion_signatures<Sndr, Env, AddlSigs, SetValue, SetError, SetStopped>` is ill-formed,

2. Otherwise, if any type in `[AddlSigs, Vs..., Es..., Ss]` is not an instantiation of `completion_signatures`, then `make_completion_signatures<Sndr, Env, AddlSigs, SetValue, SetError, SetStopped>` is an alias for `dependent_completion_signatures<no_env>`,

3. Otherwise, `make_completion_signatures<Sndr, Env, AddlSigs, SetValue, SetError, SetStopped>` names the type `completion_signatures<Sigs...>` where `Sigs...` is the unique set of types in all the template arguments of all the `completion_signatures` instantiations in `[AddlSigs, Vs..., Es..., Ss]`.

§ 6.26. execution::as_awaitable

Change [exec.as_awaitable]/p1.2.1 as follows:

1. *awaitable-receiver* is equivalent to the following:

```
struct awaitable-receiver {
  variant<monostate, result_t, exception_ptr>* result_ptr_;
  coroutine_handle<P> continuation_;
  // ... see below
};
```

Let `r` be an rvalue expression of type *awaitable-receiver*, let `cr` be a `const` lvalue that refers to `r`, let ~~`v`~~ `vs...` be an ~~expression of type `result_t`~~ arbitrary function parameter pack of types `Vs...` , and let `err` be an arbitrary expression of type `Err`. Then:

1. ~~If `value_t` is `void`, then `execution::set_value(r)` is expression-equivalent to `(r.result_ptr_->emplace<1>(), r.continuation_.resume())`; otherwise, `execution::set_value(r, v)` is expression-equivalent to `(r.result_ptr_->emplace<1>(v), r.continuation_.resume())`.~~
   If `constructible_from<result_t, Vs...>` is satisfied, the expression `execution::set_value(r, vs...)` is not potentially throwing and is equivalent to:

```
try {
  r.result_ptr_->emplace<1>(vs...);
} catch(...) {
  r.result_ptr_->emplace<2>(current_exception());
}
r.continuation_.resume();
```

   Otherwise, `execution::set_value(r, vs...)` is ill-formed.
2. The expression `execution::set_error(r, err)` is not potentially throwing and is ~~expression~~ equivalent to ~~`(r.result_ptr_->emplace<2>(AS_EXCEPT_PTR(err)), r.continuation_.resume());`~~ :

```
r.result_ptr_->emplace<2>(AS_EXCEPT_PTR(err));
r.continuation_.resume();
```

   where *AS_EXCEPT_PTR*`(err)` is:
   1. `err` if `decay_t<Err>` names the same type as `exception_ptr`,

2. Otherwise, `make_exception_ptr(system_error(err))` if `decay_t<Err>` names the same type as `error_code`,
3. Otherwise, `make_exception_ptr(err)`.

3. The expression `execution::set_stopped(r)` is not potentially throwing and is ~~expression~~ equivalent to `static_cast<coroutine_handle<>>(r.continuation_.promise().unhandled_stopped()).resume()`.
4. `tag_invoke(tag, cr, as...)` is expression-equivalent to `tag(as_const(cr.continuation_.promise()), as...)` for any expression `tag` whose type satisfies *`forwarding-receiver-query`* and for any set of arguments `as...`.